DiploCloud: Efficient and Scalable Management of RDF Data in the Cloud

Marcin Wylot and Philippe Cudré-Mauroux

eXascale Infolab
Univeristy of Fribourg—Switzerland

{firstname.lastname}@unifr.ch

Abstract—Despite recent advances in distributed RDF data management, processing large-amounts of RDF data in the cloud is still very challenging. In spite of its seemingly simple data model, RDF actually encodes rich and complex graphs mixing both instance and schema-level data. Sharding such data using classical techniques or partitioning the graph using traditional min-cut algorithms leads to very inefficient distributed operations and to a high number of joins. In this paper, we describe DiploCloud, an efficient and scalable distributed RDF data management system for the cloud. Contrary to previous approaches, DiploCloud runs a physiological analysis of both instance and schema information prior to partitioning the data. In this paper, we describe the architecture of DiploCloud, its main data structures, as well as the new algorithms we use to partition and distribute data. We also present an extensive evaluation of DiploCloud showing that our system is often two orders of magnitude faster than state-of-the-art systems on standard workloads.

Index Terms—RDF, triplestores, Cloud Computing, BigData.

1 INTRODUCTION

The advent of cloud computing enables to easily and cheaply provision computing resources, for example to test a new application or to scale a current software installation elastically. The complexity of scaling out an application in the cloud (i.e., adding new computing nodes to accommodate the growth of some process) very much depends on the process to be scaled. Often, the task at hand can be easily split into a large series of subtasks to be run independently and concurrently. Such operations are commonly called *embarrassingly parallel*. Embarrassingly parallel problems can be relatively easily scaled out in the cloud by launching new processes on new commodity machines. There are however many processes that are much more difficult to parallelize, typically because they consist of sequential processes (e.g., processes based on numerical methods such as Newton's method). Such processes are called *inherently sequential* as their running time cannot be sped up significantly regardless of the number of processors or machines used. Some problems, finally, are not inherently sequential *per se* but are difficult to parallelize in practice because of the profusion of inter-process traffic they generate.

Scaling out structured data processing often falls in the third category. Traditionally, relational data processing is scaled out by partitioning the relations and rewriting the query plans to reorder operations and use distributed versions of the operators enabling *intra-operator* parallelism. While some operations are easy to parallelize (e.g., large-scale, distributed counts), many operations, such as distributed joins, are more complex to parallelize because of the resulting traffic they potentially generate.

While much more recent than relational data management, RDF data management has borrowed many relational techniques; Many RDF systems rely on hash-partitioning (on triple or property tables, see below Section 2) and on distributed selections, projections, and joins. Our own GridVine system [1], [2] was one of the first systems to do so in the context of large-scale decentralized RDF management. Hash partitioning has many advantages, including simplicity and effective load-balancing. However, it also generates much inter-process traffic, given that related triples (e.g., that must be selected and then joined) end up being scattered on all machines.

In this article, we propose DiploCloud, an efficient, distributed and scalable RDF data processing system for distributed and cloud environments. Contrary to many distributed systems, DiploCloud uses a resolutely non-relational storage format, where semantically related data patterns are mined both from the instance-level and the schema-level data and get co-located to minimize inter-node operations. The main contributions of this article are:

- a new hybrid storage model that efficiently and effectively partitions an RDF graph and physically co-locates related instance data (Section 3);
- a new system architecture for handling fine-grained RDF partitions in large-scale (Section 4);
- novel data placement techniques to co-locate semantically related pieces of data (Section 5);
- new data loading and query execution strategies taking advantage of our system's data partitions and indices (Section 6);
- an extensive experimental evaluation showing that our system is often two orders of magnitude faster than state-of-the-art systems on standard workloads (Section 7).

DiploCloud builds on our previous approach dipLODocus$_{[RDF]}$ [3], an efficient single node triplestore. The system was also extended in TripleProv [4], [5] to support storing, tracking, and querying provenance in RDF query processing.

2 RELATED WORK

Many approaches have been proposed to optimize RDF storage and SPARQL query processing; we list below a few of the most popular approaches and systems. We refer the reader to recent surveys of the field (such as [6], [7], [8], [9] or, more recently, [10]) for a more comprehensive coverage. Approaches for storing RDF data can be broadly categorized in three subcategories: triple-table approaches, property-table approaches, and graph-based approaches. Since RDF data can be seen as sets of *subject-predicate-object* triples, many early approaches used a giant triple table to

store all data. Hexastore [11] suggests to index RDF data using six possible indices, one for each permutation of the set of columns in the triple table. RDF-3X [12] and YARS [13] follow a similar approach. BitMat [14] maintains a 3-dimensional bit-cube where each cell represents a unique triple and the cell value denotes presence or absence of the triple. Various techniques propose to speed-up RDF query processing by considering structures clustering RDF data based on their *properties*. Wilkinson et al. [15] propose the use of two types of property tables: one containing clusters of values for properties that are often co-accessed together, and one exploiting the type property of subjects to cluster similar sets of subjects together in the same table. Owens et al. [16] propose to store data in three B+-tree indexes. They use SPO, POS, and OSP permutations, where each index contains all elements of all triples. They divide a query to basic graph patterns [17] which are then matched to the stored RDF data. A number of further approaches propose to store RDF data by taking advantage of its graph structure. Yan et al. [18] suggest to divide the RDF graph into subgraphs and to build secondary indices (e.g., Bloom filters) to quickly detect whether some information can be found inside an RDF subgraph or not. Ding et al. [19] suggest to split RDF data into subgraphs (*molecules*) to more easily track provenance data by inspecting blank nodes and taking advantage of a background ontology and functional properties. Das et al. in their system called gStore [20] organize data in adjacency list tables. Each vertex is represented as an entry in the table with a list of its outgoing edges and neighbours. To index vertices, they build an S-tree in their adjacency list table to reduce the search space. Brocheler et al. [21] propose a balanced binary tree where each node containing a subgraph is located on one disk page.

Distributed RDF query processing is an active field of research. Beyond SPARQL federations approaches (which are outside of the scope of this paper), we cite a few popular approaches below.

Like an increasing number of recent systems, The Hadoop Distributed RDF Store (HDRS)[1] uses MapReduce to process distributed RDF data. RAPID+ [22] extends Apache Pig and enables more efficient SPARQL query processing on MapReduce using an alternative query algebra. Their storage model is a nested hash-map. Data is grouped around a subject which is a first level key in the map i.e. the data is co-located for a shared subject which is a hash value in the map. The nested element is a hash map with predicate as a key and object as a value. Sempala [23] builds on top of Impala [24] stores data in a wide unified property tables keeping one star-like shape per row. The authors split SPARQL queries to simple Basic Graph Patterns and rewrite them to SQL, following they compute a natural join if needed. Jena HBase[2] uses the HBase popular wide-table system to implement both triple-table and property-table distributed storage. Its data model is a column oriented, sparse, multi-dimensional sorted map. *Columns* are grouped into *column families* and timestamps add an additional dimension to each cell. Cumulus RDF[3] uses Cassandra and hash-partitioning to distribute the RDF tiples. It stores data as four indices [13] (SPO, PSO, OSP, CSPO) to support a complete index on triples and lookups on named graphs (contexts). We recently worked on an empirical evaluation to determine the extent to which such noSQL systems can be used to manage RDF data in the cloud[4] [25].

Our previous GridVine [1], [2] system uses a triple-table storage approach and hash-partitioning to distribute RDF data over decentralized P2P networks. YARS2[5], Virtuoso[6] [26], 4store [27], and SHARD [28] hash partition triples across multiple machines and parallelize the query processing. Virtuoso [26] by Erlin et al. stores data as RDF quads consisting of the following elements: graph, subject, predicate, and object. All the quads are persisted in one table and the data is partitioned based on the subject. Virtuoso implements two indexes. The default index (set as a primary key) is GSPO (Graph, Subject, Predicate, Object) and an auxiliary bitmap index (OPGS). A similar approach is proposed by Harris et al. [27], where they apply a simple storage model storing quads of (model, subject, predicate, object). Data is partitioned as non-overlapping sets of records among segments of equal subjects; segments are then distributed among nodes with a round-robin algorithm. They maintain a hash table of graphs where each entry points to a list of triples in the graph. Additionally, for each predicate, two radix tries are used where the key is either subject or object, and respectively object or subject and graph are stored as entries (they hence can be seen as traditional P:OS and P:SO indices). Literals are indexed in a separate hash table and they are represented as (S,P, O/Literal). SHARD keeps data on HDFS as star-like shape centering around a subject and all edges from this node. It introduces a clause iteration algorithm [28] the main idea of which is to iterate over all clauses and incrementally bind variables and satisfy constrains.

Huang et al. [29] deploy a single-node RDF systems (RDF-3X) on multiple machines, partition the RDF graph using standard graph partitioning algorithms (METIS[7]), and use the Hadoop framework to synchronize query execution. Their approach collocates triples forming a subgraph (a star-like structure) on particular nodes. They aim to reduce the number of inter-node joins, and thus, the amount of data that is transferred over the network for intermediate results. Warp [30] is a recent approach extending [29] and using workload-aware partial replication of triples across partitions. Queries are decomposed into chunks executed in parallel and then reconstructed with MapReduce. The authors push of most of query processing to the triplestore while only the simplest part of query execution is processed through Hadoop.

Similar combination of Hadoop and RDF-3X was used by Lee and Liu in [31]. The authors of this paper build on a simple hash partitioning and hop-based triple replication. In addition, they filter-out certain edges which tend to appear rarely in a workload from hop-based partitioning and make use of the URI hierarchy to further increase data locality. Lee and Liu extend simple hash partitioning trough direction-based triple groups and replication in order to further limit inter-machine communication cost. Queries that cannot be executed without inter-nodes communication are decomposed into sub-queries. The intermediate results of all sub-queries are then stored on HDFS, and joined using Hadoop MapReduce.

Zeng et al. [32] build on top of Trinity (a key-value store) and implement an in-memory RDF engine storing data in a graph form. The data is stored as adjacency lists for a subject, though the authors also maintain lists for in- and out-going edges of a subgraph, thus taking the form of a bidirectional subgraph. The subgraphs are then partitioned. This approach avoids joins by applying graph exploration techniques.

Gurajada et al. propose a distributed shared-nothing RDF engine named TriAd [33]. The system combines join-ahead pruning via RDF graph summarization with a locality-based, horizontal partitioning of the triples into a grid-like, distributed index structure. TriAd uses traditional graph-based partitioning techniques

1. https://code.google.com/p/hdrs/
2. http://www.utdallas.edu/~vvk072000/Research/Jena-HBase-Ext/jena-hbase-ext.html
3. https://code.google.com/p/cumulusrdf/
4. http://ribs.csres.utexas.edu/nosqlrdf/
5. http://ostatic.com/yars-2
6. http://virtuoso.openlinksw.com/
7. http://glaros.dtc.umn.edu/gkhome/views/metis

(METIS) and stores distributed triples across the nodes. The multi-threaded and distributed execution of joins in TriAD is facilitated by an asynchronous Message Passing protocol which allows to run multiple join operators along a query plan in a fully parallel and asynchronous fashion.

3 Storage Model

Our storage system in DiploCloud can be seen as a hybrid structure extending several of the ideas from above. Our system is built on three main structures: RDF molecule clusters (which can be seen as hybrid structures borrowing both from property tables and RDF subgraphs), template lists (storing literals in compact lists as in a column-oriented database system) and an efficient key index indexing URIs and literals based on the clusters they belong to. Contrary to the property-table and column-oriented approaches, our system based on templates and molecules is more elastic, in the sense that each template can be modified dynamically, for example following the insertion of new data or a shift in the workload, without requiring to alter the other templates or molecules. In addition, we introduce a unique combination of physical structures to handle RDF data both horizontally (to flexibly co-locate entities or values related to a given instance) as well as vertically (to co-locate series of entities or values attached to similar instances).

Figure 1 gives a simple example of a few molecule clusters—storing information about students—and of a template list—compactly storing lists of student IDs. Molecules can be seen as *horizontal* structures storing information about a given instance in the database (like rows in relational systems). Template lists, on the other hand, store *vertical* lists of values corresponding to one attribute (like columns in a relational system). Hence, we say that DiploCloud is a *hybrid* system, following the terminology used for approaches such as Fractured Mirrors [34] or our own recent Hyrise system [35].

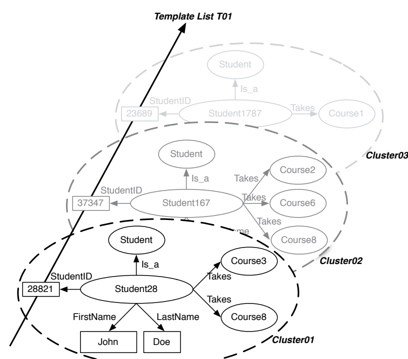

Figure 1: The two main data structures in DiploCloud: molecule clusters, storing in this case RDF subgraphs about students, and a template list, storing a list of literal values corresponding to student IDs.

Molecule clusters are used in two ways in our system: to logically group sets of related URIs and literals in the hash-table (thus, pre-computing joins), and to physically co-locate information relating to a given object on disk and in main-memory to reduce disk and CPU cache latencies. Template lists are mainly used for analytics and aggregate queries, as they allow to process long lists of literals efficiently.

3.1 Key Index

The Key Index is the central index in DiploCloud; it uses a lexicographical tree to parse each incoming URI or literal and assign it a unique numeric key value. It then stores, for every key and every template ID, an ordered list of all the clusters IDs containing the key (e.g., "key 10011, corresponding to a Course object [template ID 17], appears in clusters 1011, 1100 and 1101"; see also Figure 2 for another example). This may sound like a pretty peculiar way of indexing values, but we show below that this actually allows us to execute many queries very efficiently simply by reading or intersecting such lists in the hash-table directly.

The key index is responsible for encoding all URIs and literals appearing in the triples into a unique system id (key), and back. We use a tailored lexicographic tree to parse URIs and literals and assign them a unique numeric ID. The lexicographic tree we use is basically a prefix tree splitting the URIs or literals based on their common prefixes (since many URIs share the same prefixes) such that each substring prefix is stored once and only once in the tree. A key ID is stored at every leaf, which is composed of a type prefix (encoding the type of the element, e.g., $Student$ or $xsd: date$) and of an auto-incremented instance identifier. This prefix trees allow us to completely avoid potential collisions (caused for instance when applying hash functions on very large datasets), and also let us compactly co-locate both type and instance ids into one compact key. A second structure translates the keys back into their original form. It is composed of a set of inverted indices (one per type), each relating an instance ID to its corresponding URI / literal in the lexicographic tree in order to enable efficient key look-ups.

3.2 Templates

One of the key innovations of DiploCloud revolves around the use of *declarative storage patterns* [36] to efficiently co-locate large collections of related values on disk and in main-memory. When setting-up a new database, the database administrator may give DiploCloud a few hints as to how to store the data on disk: the administrator can give a list of triple patterns to specify the *root nodes*, both for the template lists and the molecule clusters (see for instance Figure 1, where "Student" is the root node of the molecule, and "StudentID" is the root node for the template list). Cluster roots are used to determine which clusters to create: a new cluster is created for each instance of a root node in the database. The clusters contain all triples departing from the root node when traversing the graph, until another instance of a root node is crossed (thus, one can join clusters based on their root nodes). Template roots are used to determine which literals to store in template lists.

Based on the storage patterns, the system handles two main operations in our system: i) it maintains a schema of triple templates in main-memory and ii) it manages template lists. Whenever a new triples enters the system, it associates template IDs corresponding to the triple by considering the type of the subject, the predicate, and the type of the object. Each distinct list of "(subject-type, predicate, object-type)" defines a new triple template. The triple templates play the role of an instance-based RDF schema in our system. We don't rely on the explicit RDF schema to define the templates, since a large proportions of constraints (e.g., domains, ranges) are often omitted in the schema (as it is for example the case for the data we consider in our experiments, see Section 7). In case a new template is detected (e.g., a new predicate is used), then the template manager updates its in-memory triple template schema and inserts new template IDs to reflect the new pattern it discovered. Figure 2 gives an example of a template. In case of very inhomogeneous data sets containing millions of different triple templates, wildcards can be used to regroup similar templates (e.g., "Student - likes - *"). Note that this is very rare in practice, since all the datasets we encountered so far (even those in

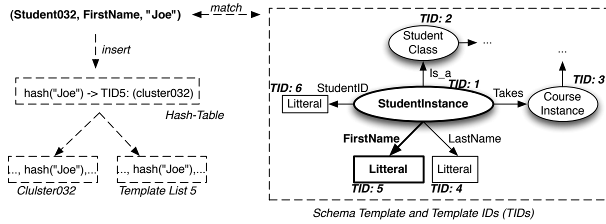

Figure 2: An insert using templates: an incoming triple (left) is matched to the current RDF template of the database (right), and inserted into the hash-table, a cluster, and a template list.

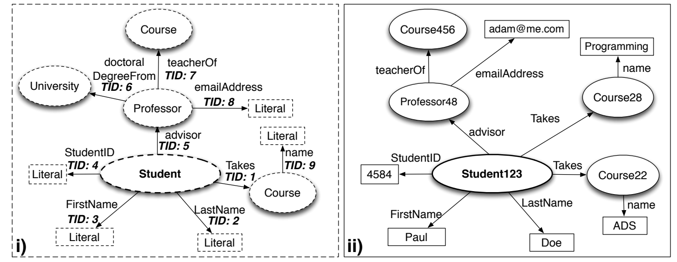

Figure 3: A molecule template (i) along with one of its RDF molecules (ii)

the LOD cloud) typically consider a few thousands triple templates at most.

Afterwards, the system inserts the triple in one or several molecules. If the triple's object corresponds to a root template list, the object is also inserted into the template list corresponding to its template ID. Templates lists store literal values along with the key of their corresponding cluster root. They are stored compactly and segmented in sublists, both on disk and in main-memory. Template lists are typically sorted by considering a lexical order on their literal values—though other orders can be specified by the database administrator when he declares the template roots. In that sense, template lists are reminiscent of *segments* in a column-oriented database system.

3.3 Molecules

DiploCloud uses physiological RDF partitioning and molecule patterns to efficiently co-locate RDF data in distributed settings. Figure 3 (ii) gives an example of molecule. Molecules have three key advantages in our context:

- Molecules represent the ideal tradeoff between co-location and degree of parallelism when partitioning RDF data. Partitioning RDF data at the triple-level is suboptimal because of the many joins it generates; Large graph partitions (such as those defined in [29]) are suboptimal as well, since in that case too many related triples are co-located, thus inhibiting parallel processing (see Section 7).
- All molecules are template-based, and hence store data extremely compactly;
- Finally, the molecules are defined in order to *materialize* frequent joins, for example between an entity and its corresponding values (e.g., between a student and his/her firstname), or between two semantically related entities (e.g., between a student and his/her advisor) that are frequently co-accessed.

When receiving a new triple the system inserts it in the corresponding molecule(s). In case the corresponding molecule does not exist yet, the system creates a new molecule cluster, inserts the triple in the molecule, and inserts the cluster in the list of clusters it maintains. Figures 3 gives a template example that co-locates information relating to Student instances along with an instance of a molecule for Student123.

Similarly to the template lists, the molecule clusters are serialized in a very compact form, both on disk and in main-memory. Each cluster is composed of two parts: a list of offsets, containing for each template ID in the molecule the offset at which the keys corresponding for the template ID are stored, and the list of keys themselves. Thus, the size of a molecule, both on-disk and in main-memory, is $\#TEMPLATES + (KEY_SIZE * \#TRIPLES)$, where KEY_SIZE is the size of a key (in bytes), $\#TEMPLATES$ is the number of templates IDs in the molecule, and $\#TRIPLES$ is the number of triples in the molecule (we note that this storage structure is much more compact than a standard list of triples). To retrieve a given information in a molecule, the system first determines the position of the template ID corresponding to the information sought in the molecule (e.g., "FirstName" is the sixth template ID for the "Student" molecule above in Figure 2). It then jumps to the offset corresponding to that position (e.g., 5th offset in our example), reads that offset and the offset of the following template ID, and finally retrieves all the keys/values between those two offsets to get all the values corresponding to that template ID in the molecule. The molecule depicted in Figures 3 (ii), for instance, contains 15 triples (including type information), and would hence require 45 URIs/literals to be encoded using a standard triple-based serialization. Our molecule, on the other hand, only requires to store 10 keys to be correctly defined, yielding a compression ratio of $1 : 4.5$.

3.4 Auxiliary Indexes

While creating molecule templates and molecules identifiers, our system also take cares of two additional data gathering and analysis tasks. First, it inspects both the schema and instance data to determine all subsumption (subclass) relations between the classes, and maintains this information in a compact *type hierarchy*. We assign to every key the most specific type possible in order to avoid having to materialize the type hierarchy for every instance, and handle type inference at query time by looking-up types in the type hierarchy. In case two unrelated types are assigned to a given instance, the partition manager creates a new virtual type composed of the two types and assigns it to the instance. Finally, we maintain *statistics* on each templates, counting the number of instances for each vertex (instance / literal) and edge (property) in the templates.

For each type, DiploCloud also maintains a list of the keys belonging to that type (*type index*). In addition, it maintains a *molecule index* storing for each key the list of molecules storing that key (e.g., "key 15123 [Course12] is stored in molecule 23521 [root:Student543]"). This index is particularly useful to answer triple-pattern queries as we explain below in Section 6.

4 System Overview

Figure 4 gives a simplified architecture of DiploCloud. DiploCloud is a native, RDF database system. It was designed to run on clusters of commodity machines in order to *scale out* gracefully when handling bigger RDF datasets.

Our system design follows the architecture of many modern cloud-based distributed systems (e.g., Google's BigTable [37]), where one (Master) node is responsible for interacting with the clients and orchestrating the operations performed by the other (Worker) nodes.

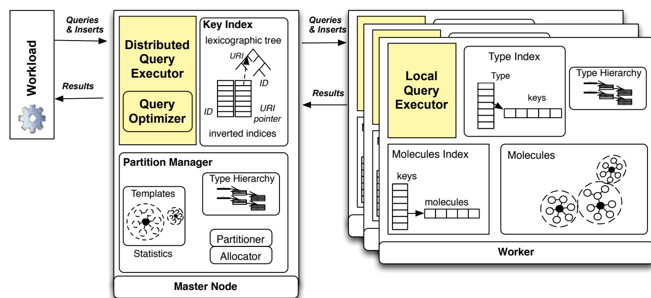

Figure 4: The architecture of DiploCloud.

4.1 Master Node

The Master node is composed of three main subcomponents: a key index (c.f. Section 3.1), in charge of encoding URIs and literals into compact system identifiers and of translating them back, a partition manager (c.f. Section 5), responsible for partitioning the RDF data into recurring subgraphs, and a distributed query executor (c.f. Section 6.3), responsible for parsing the incoming query, rewriting the query plans for the Workers, collecting and finally returning the results to the client. Note that the Master node can be replicated whenever necessary to insure proper query load-balancing and fault-tolerance. The Master can also be duplicated to scale out the key index for extremely large datasets, or to replicate the dataset on the Workers using different partitioning schemes (in that case, each new instance of the Master is responsible for one partitioning scheme).

4.2 Worker Nodes

The Worker nodes hold the partitioned data and its corresponding local indices, and are responsible for running subqueries and sending results back to the Master node. Conceptually, the Workers are much simpler than the Master node and are built on three main data structures: i) a type index, clustering all keys based on their types ii) a series of RDF *molecules*, storing RDF data as very compact subgraphs, and iii) a molecule index, storing for each key the list of molecules where the key can be found.

5 Data Partitioning & Allocation

As mentioned in Section 2, triple-table and property-table hash-partitionings are currently the most common partitioning schemes for distributed RDF systems. While simple, such hash-partitionings almost systematically implies some distributed coordination overhead (e.g., to execute joins / path traversals on the RDF graph), thus making it inappropriate for most large-scale clusters and cloud computing environments exhibiting high network latencies. The other two standard relational partitioning techniques, (tuple) round-robin and range partitioning, are similarly flawed for the data and setting we consider, since they would partition triples either at random or based on the subject URI / type, hence seriously limiting the parallelism of most operators (e.g., since many instances sharing the same type would end up on the same node).

Partitioning RDF data based on standard graph partitioning techniques (similarly to what [29] proposes) is also from our perspective inappropriate in a cloud context, for three main reasons:

Loss of semantics: standard graph partitioning tools (such as METIS[8], which was used in [29]) consider unlabeled graphs mostly, and hence are totally agnostic to the richness of an RDF graph including classes of nodes and edges.

Loss of parallelism: partitioning an RDF graph based, for instance, on a min-cut algorithm will lead to very *coarse* partitions where a high number of related instances (for instance linked to the same type or sharing links to the same objects) will be co-located, thus drastically limiting the degree of parallelism of many operators (e.g., projections or selections on certain types of instances).

Limited scalability: finally, attempting to partition very large RDF graphs is unrealistic in cloud environments, given that state-of-the-art graph partitioning techniques are inherently centralized and data/CPU intensive (as an anecdotal evidence, we had to borrow a powerful server and let it run for several hours to partition the largest dataset we use in Section 7 using METIS).

DiploCloud has been conceived from the ground up to support distributed data partitioning and co-location schemes in an efficient and flexible way. DiploCloud adopts an intermediate solution between tuple-partitioning and graph-partitioning by opting for a recurring, fine-grained graph-partitioning technique taking advantage of molecule templates. DiploCloud's molecule templates capture recurring patterns occurring in the RDF data naturally, by inspecting both the instance-level (*physical*) and the schema-level (*logical*) data, hence the expression *physiological*[9] partitioning.

5.1 Physiological Data Partitioning

We now define the three main molecule-based data partitioning techniques supported by our system:

Scope-k Molecules: the simplest method is to manually define a number of template types (by default the system considers all types) serving as root nodes for the molecules, and then to co-locate all further nodes that are directly or indirectly connected to the roots, up to given scope k. Scope-1 molecules, for example, co-locate in the molecules all root nodes with their direct neighbors (instances or literals) as defined by the templates. Scope-2 or 3 molecules concatenate compatible templates from the root node (e.g., $(student, takes, course)$ and $(course, hasid, xsd : integer)$) recursively up to depth k to materialize the joins around each root, at the expense of rapidly increasing storage costs since much data is typically replicated in that case (see Section 7). The scope of the molecules is defined in this case manually and involves data duplication. All data above Scope-1 is duplicated; this is the price to pay in order to benefit from pre-computed joins inside the molecules, which significantly increases query execution performance as we show in the following.

Manual Partitioning: Root nodes and the way to concatenate the various templates can also be specified by hand by the database administrator, who just has to write a configuration file specifying the roots and the way templates should be concatenated to define the generic shape of each molecule type. Using this technique, the administrator basically specifies, based on resource types, the exact path following which molecules should be physically extended. The system then automatically duplicates data following the administrator's specification and pre-computes all joins inside the molecules. This is typically the best solution for relatively stable datasets and workloads whose main features are well-known.

8. http://glaros.dtc.umn.edu/gkhome/views/metis

9. *physiological* characterizes in our context a process that work both on the physical and logical layers of the database, as the classical Aries recovery algorithm

Adaptive Partitioning: Finally, DiploCloud's most flexible partitioning algorithm starts by defining scope-1 molecules by default, and then adapts the templates following the query workload. The system maintains a sliding-window w tracking the recent history of the workload, as well as related statistics about the number of joins that had to be performed and the incriminating edges (e.g., missing co-location between students and courses causing a large number of joins). Then at each time epoch ϵ, the system: i) expands one molecule template by selectively concatenating the edges (rules) that are responsible for the most joins up to a given threshold for their maximal depth and ii) decreases (up to scope-1) all extended molecules whose extensions were not queried during the last epoch. In that way, our system slowly adapts to the workload and materializes frequent paths in the RDF graph while keeping the overall size of the molecules small. Similarly to the two previous techniques, when the scope of a molecule is extended, the system duplicates the relevant pieces of data and pre-computes the joins. The advantage of this method is that it begins with relatively simple and compact data structures and then automatically adapts to the dynamic workload by increasing and decreasing the scope of specific molecules, i.e., by adding and removing pre-computed paths based on template specifications. In the case of a very dynamic workload, the system will not adapt the structures in order to avoid frequent rewriting costs that would not by easily amortized by the improvement in query processing.

5.2 Distributed Data Allocation

Once the physiological partitions are defined, DiploCloud still faces the choice of how to distribute the concrete partitions (i.e, the actual RDF molecules defined from the molecule templates) across the physical nodes. Data allocation in distributed RDF systems is delicate, since a given allocation scheme has to find a good tradeoff between perfect load-balancing and data co-location. Our template manager implements three main allocation techniques:

Round-Robin: The round-robin allocation simply takes each new molecule it defines and assigns it to the next worker. This scheme favors load-balancing mostly.

Coarse Allocation: Coarse allocation splits the incoming data in W parts, where W is the number of workers, and assigns each part to a given worker. This allocation scheme favors data co-location mostly.

Semantic Co-location: The third allocation tries to achieve a tradeoff between load-balancing and co-location by grouping a small number of molecule instances (typically 10) that are semantically related through some connection (i.e., predicate), and then by allocating such groups in a round-robing fashion.

6 Common Operations

We now turn to describing how our system handles typical operations in distributed environments.

6.1 Bulk Load

Loading RDF data is generally speaking a rather expensive operation in DiploCloud but can be executed in a fairly efficient way when considered in bulk. We basically trade relatively complex instance data examination and complex local co-location for faster query execution. We are willing to make this tradeoff in order to speed-up complex queries using our various data partitioning and allocation schemes, especially in a Semantic Web or LOD context where isolated inserts or updates are from our experience rather infrequent.

We assume that the data to be loaded is available in a shared space on the cloud. Bulk loading is a hybrid process involving both the Master—whose task is to encode all incoming data, to identify potential molecule roots from the instances, and to assign them to the Workers using some allocation scheme—and all the Workers—which build, store and index their respective molecules in parallel based on the molecule templates defined.

On the worker nodes, building the molecule is an n-pass algorithm (where n is the deepest level of the molecule, see Section 3) in DiploCloud, since we need to construct the RDF molecules in the clusters (i.e., we need to materialize triple joins to form the clusters). In a first pass, we identify all root nodes and their corresponding template IDs, and create all clusters. The subsequent passes are used to join triples to the root nodes (hence, the student clusters depicted in Figure 1 are built in two phases, one for the Student root node, and one for the triples directly connected to the Student). During this operation, we also update the template lists and the key index incrementally. Bulk inserts have been highly optimized in DiploCloud, and use an efficient page-manager to execute inserts for large datasets that cannot be kept in main-memory.

This division of work and the fact that the expensive operation (molecule construction) is performed in parallel enables DiploCloud to bulk load efficiently as we show in Section 7.

6.2 Updates

As for other hybrid or analytic systems, updates can be relatively complex to handle in DiploCloud, since they might lead to a partial rewrite of the key index and molecule indices, and to a reorganization of the physical structures of several molecules. To handle them efficiently, we adopt a lazy rewrite strategy, similarly to many modern read-optimized system (e.g., CStore or BigTable). All updates are performed on write-optimized log-structures in main-memory. At query time, both the primary (read-optimized) and log-structured (write-optimized) data stores are tapped in order to return the correct results.

We distinguish between two kinds of updates: in-place and complex updates. In-place updates are punctual updates on literal values; they can be processed directly in our system by updating the key index, the corresponding cluster, and the template lists if necessary. Complex updates are updates modifying object properties in the molecules. They are more complex to handle than in-place updates, since they might require a rewrite of a list of clusters in the key index, and a rewrite of a list of keys in the molecule clusters. To allow for efficient operations, complex updates are treated like updates in a column-store (see [38]): the corresponding structures are flagged in the key index, and new structures are maintained in write-optimized structures in main-memory. Periodically, the write-optimized structures are merged with the main data structures in an offline fashion.

6.3 Query Processing

Query processing in DiploCloud is very different from previous approaches to execute queries on RDF data, because of the three peculiar data structures in our system: a key index associating URIs and literals to template IDs and cluster lists, clusters storing RDF molecules in a very compact fashion, and template lists storing compact lists of literals. All queries composed of one Basic Graph Pattern (star-like queries) are executed totally in parallel, independently on all Workers without any central coordination thanks to the molecules and their indices.

For queries that still require some degree of distributed coordination—typically to handle distributed joins—we resort to adaptive query execution strategies. We mainly have two ways of executing distributed joins: whenever the intermediate result set is

small (i.e., up to a few hundred tuples according to our Statistics components), we ship all results to the Master, which finalizes the join centrally. Otherwise, we fall back to a distributed hash-join by distributing the smallest result set among the Workers. Distributed joins can be avoided in many cases by resorting to the distributed data partitioning and data co-location schemes described above.

Algorithm 1 gives a high-level description of our distributed query execution process highlighting where particular operations are performed in our system.

Algorithm 1 High Level Query Execution Algorithm

1: Master: divide query based on molecule scopes to obtain sub-queries
2: Master: send sub-queries to workers
3: Workers: execute sub-queries in parallel
4: Master: collect intermediate results
5: Master: perform distributed join whenever necessary

We describe below how a few common queries are handled in DiploCloud.

6.3.1 Basic Graph Patterns

Basic Graph Patterns represent queries retrieving triples sharing the same subject; they are relatively simple in DiploCloud: they are usually resolved by looking for a bound-variable (URI) in the key index or molecules index, retrieving the corresponding molecules numbers, and finally retrieving values from the molecules when necessary. Conjunctions and disjunctions of triples patterns can be resolved very efficiently in our system. Since the RDF nodes are logically grouped by molecules in the key index, it is typically sufficient to read the corresponding list of molecules in the molecules index. No join operation is needed since joins are implicitly materialized in molecules. The following query (query # 1 in Section 7), for instance:

```
?X a :GraduateStudent .
?X :takesCourse <GraduateCourse0> .
```

is first optimized by the Master based on the statistics it collected; a query plan is then sent to all Workers asking them to first look-up all molecules containing $GraduateCourse0$ (since it is the most selective pattern in the query) using their local molecule index. Each Worker can then contribute to the results independently and in parallel, by retrieving the molecule ids, filtering them based on the $GraduateStudent$ type (by simply inspecting the ids) and returning the resulting ids to the master node. If the template ID of $GraduateCourse0$ in the molecule is ambiguous (for example when a $GraduateStudent$ can both teach and take courses), then an additional filtering step is carried out locally at the end of the query plan by looking up molecules and filtering them based on their predicate (e.g., predicate linking $GraduateStudent$ to $GraduateCourse0$).

6.3.2 Molecule Queries

Molecule queries or queries retrieving many values/instances around a given instance (for example for visualization purposes) are also extremely efficient in our system. Those queries start with a shared subject and extend beyond scope-1. They represent an extended star-like query involving subject-object joins. In most cases, the key index is invoked to find the corresponding molecule (if the scope of the query matches the scope of a molecule), which contains then all the corresponding values. For bigger scopes (such as the ones we consider in our experimental evaluation below), our system can efficiently join clusters based on the various root nodes they contain.

6.3.3 Aggregates and Analytics

Aggregate and analytic queries can also be efficiently resolved by our system. Many analytic queries can be solved by first intersecting lists of clusters in the molecule index, and then looking up values in the remaining molecule clusters. Large analytic or aggregate queries on literals (such as our third analytic query below, returning the names of all graduate students) can be extremely efficiently resolved by taking advantage of template lists (containing compact and sorted lists of literal values for a given template ID), or by filtering template lists based on lists of molecule IDs retrieved from the key index. Typically those queries involves triple patterns consisting of type look-ups, or aggregate operations such as average, mean, literals operations, etc. operating on long series of similar instances.

6.3.4 Distributed Joins

That kind of queries regroups various flavors of joins (subject-object, object-object, triangular joins, etc.). We execute them by dividing them into molecule queries or basic graph patterns, depending on the scopes of the molecules in the configuration; following this, we execute each resulting subquery in parallel on worker nodes and then execute distributed joins to combine the results of the individual subqueries.

As a more complete example of query processing, we consider the following LUBM [39] query:

```
?Z is_a :Department .
?Y is_a :University .
?X is_a :GraduateStudent .
?Z :subOrganizationOf ?Y .              <— 1st
?X :undergraduateDegreeFrom ?Y .<—— 2nd
?X :memberOf ?Z .                        <— 3rd
```

We briefly discuss three possible strategies for dealing with this query below.

For the simplest and the most generic one (Algorithm 2), we divide the query into 3 basic graph patterns and we prepare intermediate results on each node; we then send them to the Master node where we perform the final join. In that way we retrieve elements meeting the 1st constraint (*Department subOrganizationOf University*), then the 2nd constraint (*GraduateStudent undergraduateDegreeFrom University*), and the 3rd constraint (*GraduateStudent memberOf Department*). Finally, we perform hash-joins for all those intermediate results on the Master node.

For the second method, we similarly divide the query into 3 basic graph patterns and we prepare, on each node, intermediate results for the 1st constraint, following we distribute them across the cluster, since in every molecule of type $GraduateStudent$, we have all information about the object instance (i.e. $undergraduateDegreeFrom$ and $memberOf$) for each $GraduateStudent$; having distributed intermediate results corresponding to the 1st constraint, we can perform the joint for the 2nd and 3rd constraints completely in parallel.

The third and most efficient strategy would be to increase the scope of the considered molecules, so that in every molecule, besides all information about the root ($GraduateStudent$), we would also store all information about $Department$ related to the root, and further $University$ related to the $Department$. To answer the query, we just need to retrieve data about the 2nd and the 3rd constraints in this case, and perform a check on the molecule to validate that a given $University$ from the 2nd constraint is the same as the one related to the $Department$ from the 3rd constraint, which indicates that the 1st constraint is met. The query is them executed completely in parallel on the worker nodes, without involving neither distributed nor centralized joins on the Master.

Algorithm 2 Query Execution Algorithm with Join on the Master Node

```
 1: procedure ExecuteQuery(a, b)
 2:   for all BGP in QUERY do          ▷ BGP - Basig Graph Pattern
 3:     if BGP.subject then
 4:       molecules ← GetMolecule(subject)
 5:     else if BGP.object then
 6:       molecules ← GetMolecules(object)
 7:     end if
 8:     for all molecules do
 9:                         ▷ check if the molecule matches the BGP
10:       for all TP in BGP do          ▷ TP - Triple Pattern
11:         if TP.subject != molecule.subject then
12:           nextMolecule
13:         end if
14:         if TP.predicate != molecule.predicate then
15:           nextMolecule
16:         end if
17:         if TP.object != molecule.object then
18:           nextMolecule
19:         end if
20:       end for
21:  ▷ the molecule matches the BGP, so we can retrieve entities
22:       resultBGP ← GetEntities(molecule,BGP)
23:     end for
24:     results ← resultBGP
25:   end for
26:   SendToMasterNode(results)
27: end procedure
28:                                     ▷ On the Master do Hash Join
```

7 Performance Evaluation

We have implemented a prototype of DiploCloud following the architecture and techniques described above. We note that in the current prototype we did not implement dynamic updates. Point updates are expensive in our system where related pieces of data are co-located. They could be implemented in a standard way by considering a write-optimized store, which is a common technique used for column-oriented database systems [38]. In our prototype, we support efficient bulk inserts or updates through batch operations. The following experiments were conducted for two scenarios: centralized and distributed. For each of them, we evaluated the performance of DiploCloud and we compared it with the state-of-the-art systems and techniques.

7.1 Datasets and Workloads

To compare the various systems, we used three different benchmarks.

- the Lehigh University Benchmark (LUBM) [39]
- the BowlognaBench Benchmark [40]
- the DBPedia dataset with five queries [41]

LUBM is one of the oldest and most popular benchmarks for Semantic Web data. It provides an ontology describing universities together with a data generator and fourteen queries. We generated the following datasets:

- 10 universities: 1’272’814 triples [226 MB]
- 100 universities: 13’876’209 triples [2.4 GB]
- 400 universities: 55 035 263 triples [9.4 GB]
- 800 universities: 110 128 171 triples [19 GB]
- 1600 universities: 220 416 262 triples [38 GB]

We compared the runtime execution for LUBM queries and for three analytic queries inspired by BowlognaBench [40]. LUBM queries are criticized by some for their reasoning coverage; this was not an issue in our case, since we focused on RDF DB query processing rather than on reasoning capabilities. We keep an in-memory representation of subsumption trees in DiploCloud and rewrite queries automatically to support subclass inference for the LUBM queries. We manually rewrote inference queries for the systems that do not support such functionalities.

The three additional analytic/aggregate queries that we considered are as follows: 1) a query returning the professor who supervises the most students 2) a query returning a big molecule containing all triples within a scope of 2 of Student0 and 3) a query returning all graduate students.

For BowlognaBench, we used two different datasets generated with the BowlognaBench Instance Generator:

- 1 departments: 1.2 million triples [273MB]
- 10 departments: 12 millions triples [2.7GB]

For both datasets we set 4 fields per department and 15 semesters. We run the 13 queries of BowlognaBench to compare the query execution time for RDF systems.

Additionally, we also used a dataset extracted from DBPedia (which is interesting in our context as it is much more *noisy* than the LUBM and BowlognaBench data) with five queries [41]. From the original DBpedia 3.5.1, we extracted a subset of:

- 73 731 354 triples [9.3 GB]

All inference queries were implemented by rewriting the query plans for DiploCloud and the systems that did not support such queries.

7.2 Methodology

As for other benchmarks (e.g., tpc-*x*[10] or our own OLTP-Benchmark [42]) we include a warm-up phase before measuring the execution time of the queries. We first run all the queries in sequence once to warm-up the systems, and then repeat the process ten times (i.e., we run in total 11 batches containing all the queries in sequence for each system). We report the mean values for each query and each system below. We assumed that the maximum time for each query should not exceed 2 hours (we stopped the tests if one query took more than two hours to be executed). We compared the output of all queries running on all systems to ensure that all results were correct.

We tried to do a reasonable optimization job for each system, by following the recommendations given in the installation guides for each system. We did not try to optimize the systems any further, however. We performed no fine-tuning or optimization for DiploCloud.

We avoided the artifact of connecting to the server, initializing the DB from files and printing results for all systems; we measured instead the query execution times only.

7.3 Systems

We chose those systems to have different comparison points, and because they were either freely available on the Web, or possible to implement with relatively little effort. We give a few details about each system below.

AllegroGraph [43] We used AllegroGraph RDFStore 4.2.1 AllegroGraph unfortunately poses some limits on the number of triples that can be stored for the free edition, such that we couldn’t load the big data set. For AllegroGraph, we prepared a SPARQL Python script using libraries provided by the vendor.

BigOWLIM [44] We used BigOWLIM 3.5.3436. OWLIM provides us with a java application to run the LUBM benchmark, so we used it directly for our tests.

10. *http://www.tpc.org/*

	DiploCloud	AllegroGraph	BigOwlim	Virtuoso	RDF-3X	Jena
Load Time [s]	31	13	50	88	16	98
size [MB]	87	696	209	140	66	118

Table 1: Load times and size of the databases for the 10 universities LUBM data set.

	DiploCloud	BigOwlim	Virtuoso	RDF-3X	Jena
Load Time [s]	427	748	914	214	1146
size [MB]	913	2012	772	694	1245

Table 2: Load times and size of the databases for the 100 universities LUBM data set.

Jena [16] We used Jena-2.6.4 and the TDB-0.8.10 storage component. We created the database by using the "tdbloader" provided by Jena. We created a Java application to run and measure the execution time of each query.

Virtuoso [26] We used Virtuoso Open-Source Edition 6.1.3. Virtuoso supports ODBC connections, and we prepared a Python script using the PyODBC library for our queries.

RDF-3X [12] We used RDF-3X 0.3.5. We slightly modified the system to measure the execution time of the queries only, without taking into account the initialization of the database and turning off the print-outs.

4store [27] is a well-known distributed, native RDF system based on property tables and distributing triples (or quads, actually) based on a hash-partitioning of their subject. We used 4store revision v1.1.4., with eight segments per node, and the provided tools to load and query.

SHARD [28] stores RDF triples directly in HDFS and takes advantage of Hadoop for all distributed processes. We slightly modified the system in order to measure the execution time of the queries only, without taking into account the initialization of the database and by turning off the print-outs.

RDF-3X GraphPartitioning : we re-implemented the base approach described in [29] by using RDF-3X and by partitioning the RDF data using METIS. Rather than using Hadoop for the distributed coordination, we implemented all distributed joins in Java, following the same design as for our own prototype.

7.4 Centralized Environment

7.4.1 Hardware Platform

All experiments were run on a HP ProLiant DL360 G7 server with two Quad-Core Intel Xeon Processor E5640, 6GB of DDR3 RAM and running Linux Ubuntu 10.10 (Maverick Meerkat). All data were stored on recent 1.4 TB Serial ATA disk.

7.4.2 Results

Relative execution times for all queries and all systems are given below, in Figure 5 (log-scale) for 10 universities and in Figure 6 (log-scale) for 100 universities. The Tables 1 and 2 shows the loading time in seconds and the storage consumption in MB for respectively 10 and 100 universities of the LUBM benchmark.

We observe that DiploCloud is generally speaking very fast, both for the bulk inserts, for the LUBM queries and especially for the analytic queries. DiploCloud is not the fastest system for inserts, and produces slightly larger databases on disk than some other systems (like RDF-3X), but performs overall very-well for all the queries. Our system is on average 30 times faster than the fastest RDF data management system we have considered (i.e., RDF-3X) for the LUBM queries, and on average 350 times faster than the fastest system (Virtuoso) on the analytic queries. Is is also very scalable (both the bulk insert and the query processing scale gracefully from 10 to 100 universities). We can see (Tables 3 and 4) that Virtuoso takes more time to load and index the dataset but the size of the indices scales better than for the other systems. The fastest system is 4Store which also has the biggest indices. Both RDF-3X and Virtuoso achieve a good compression.

Figures 7 (log-scale) and 8 (log-scale) report the experimental results for the BowlognaBench datasets consisting of 1 and 10 departments respectively. The values indicate query execution times for each query of the BowlognaBench benchmark. We note that query 4 could not be run on RDF-3X and DiploCloud as they do not provide support for pattern matching. The Tables 3 and 4

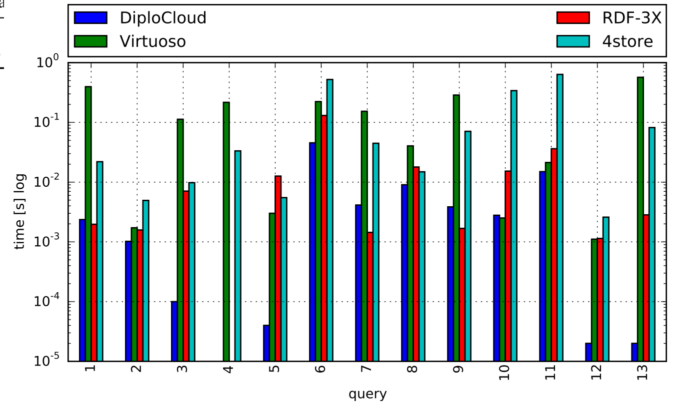

Figure 7: Query execution time for the 1 department BowlognaBench data set.

	DiploCloud	Virtuoso	RDF-3X	4store
Load Time [s]	18.3503	31.71	11.94	6.25
size [MB]	92.0000	108.00	60.00	192.00

Table 3: Load times and size of the databases for the 1 department BowlognaBench data set.

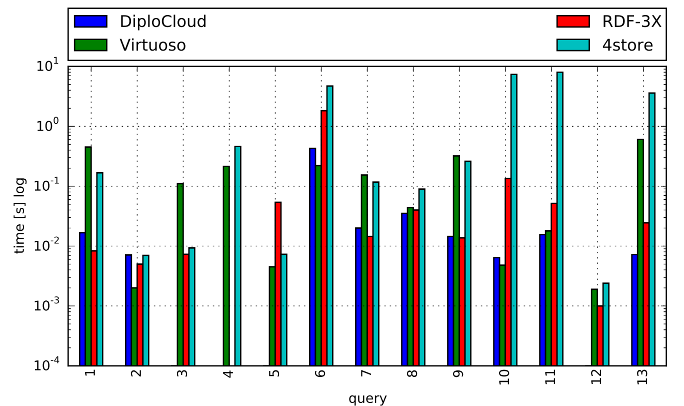

Figure 8: Query execution time for the 10 department BowlognaBench data set.

	DiploCloud	Virtuoso	RDF-3X	4store
Load Time [s]	526.652	363.24	139.55	69.65
size [MB]	920.000	616.00	618.00	1752.00

Table 4: Load times and size of the databases for the 10 department BowlognaBench data set.

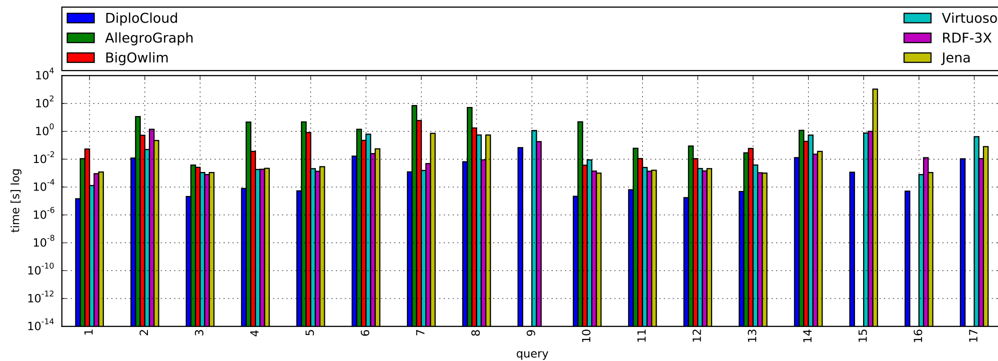

Figure 5: Query execution time for the 10 universities LUBM data set

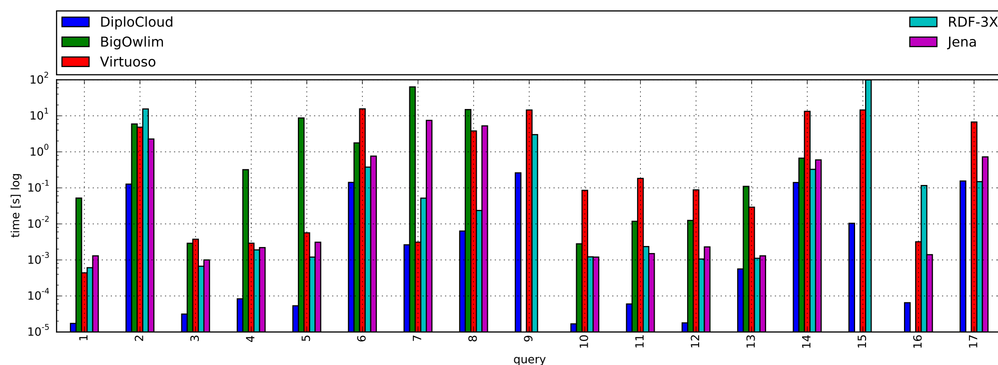

Figure 6: Query execution time for the 100 universities LUBM data set

shows the loading time in seconds and the storage consumption in MB for respectively 1 and 10 departments.

As we can observe, the query execution time for the BowlognaBench analytic queries strongly vary for different systems. DiploCloud is slightly slower for the queries 1 and 7 than RDF-3X, and it is outperformed by Virtuoso for the queries 2 and 10. We can observe the slower performance of 4Store for 10 out of 13 queries as compared with the other systems: for some queries (e.g. 10) the execution times took more than 7 seconds. Specifically, longest query executions can be observed for the queries 6, 10, and 11. The slowest is the path query which involves several joins. For all those queries DiploCloud performs very well. We can see that the query 8 is not easy to be efficiently answered for all the systems. The queries 3 and 11 are also challenging because of the several joins involved, though DiploCloud handles them without any problem (especially the query 3). Instead, the count queries (i.e., 1 and 2) can be performed quite efficiently. One difference that we can observe for the bigger dataset of BowlognaBench as compared with the smaller dataset is the good result of Virtuoso: it performed faster than RDF-3X on 10 out of 13 queries. We can also observe that DiploCloud scales very well, whilst the competitors for some cases have issues handling the big dataset (e.g. 4store query 8, RDF-3X query 6). In general, we can again observe that DiploCloud outperforms the competitors for most of the queries for the both datasets and that it scales gracefully.

The impressive performance in centralized environments can be explained by several salient features of our system, including: its extremely compact structures based on molecule templates to store related pieces of data, its physically redundant structures to optimize different types of operations (e.g., aggregates), and its way of pre-materializing joins in the data structures following the administrator's decisions or shifts in the query workload. This high performance is counterbalanced by relatively complex and expensive inserts, which can however be optimized if considered in bulk.

7.5 Distributed Environment

In the previous section, we empirically evaluated and discussed the advantages of our storage model in a single-node scenario. We showed that the techniques we introduced represent an efficient way for storing RDF in centralized environments, and how our physical model and indices allow to execute queries efficiently. Now, we turn to investigating the performance of our approach in a distributed environment. In the following, we evaluate the behavior of our system on live, distributed deployments on clusters of commodity machines and in the cloud. We demonstrate how our partitioning, co-location, and distributed query processing techniques are leveraged in distributed settings, minimizing the data transfers across the network while parallelizing query execution.

7.5.1 Hardware Platform

All experiments (except the EC2 experiments) were run in three cloud configurations of 4, 8, and 16 nodes. Worker nodes were commodity machines with Quad-Core Intel i7-2600 CPUs @ 3.40GHz, 8GB of DDR3-1600 RAM, 500GB Serial ATA HDD, running Ubuntu 12.04.2 LTS. The Master node was similar, but with 16GB RAM.

7.5.2 Results

We start by comparing query execution times for DiploCloud deployed in its simplest configuration i.e., partitioning with Scope-1 molecules, and allocating molecules in a round-robin fashion.

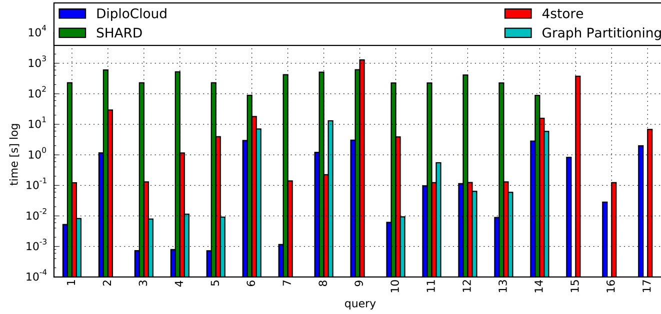

Figure 9: Query execution time for 4 nodes and 400 universities LUBM data set

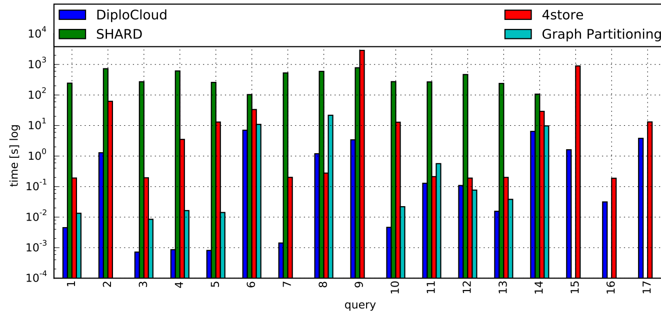

Figure 10: Query execution time for 8 nodes and 800 universities LUBM data set

Figures 9, 10, and 11 (log-scale) give the results for the LUBM datasets for 400, 800, and 1600 universities executed respectively on 4, 8, and 16 servers. Note that several queries timed-out for GraphPartitioning (2, 7, 9, 15, 16, 17) (mostly due to the very large number of generated intermediate results, and due to the subsequent distributed joins). On the biggest deployment, DiploCloud is on average 140 times faster than 4store, 244 times faster than SHARD, and 485 times faster than the graph partitioning approach using RDF-3X (including the time-out values for the timed-out queries). Figures 12, 13, and 14 (log-scale) give the results for the DPBedia dataset. DiploCloud achieves sub-second latencies on most queries, and is particularly efficient when deployed on larger clusters. We explain some of those results in more detail below.

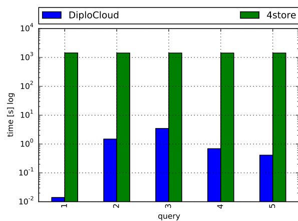

Figure 12: Query execution time for DBPedia running on 4 nodes

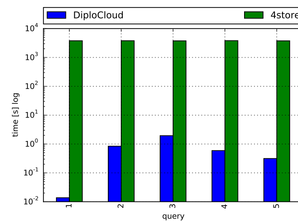

Figure 13: Query execution time for DBPedia running on 8 nodes

Data Partitioning & Allocation: We now turn to our adaptive partitioning approach. We implemented our adaptive partitioning approach, keeping all the queries in the history, considering a max-depth of 2, and switching to a new time epoch after each query batch. The results are available in Figures 15, 16, and 17 (log-scale) for respectively 4, 8, and 16 nodes. Only the deepest (in terms of RDF paths) LUBM queries are shown on the graphs (the other queries behave the same for both partitioning schemes). By co-locating all frequently queried elements, query execution using the adaptive partitioning is on average more than 3 times faster than the simple partitioning for those queries. Note that scope-2 molecules would behave like the adaptive scheme in that case, but take much more space (see Table 6).

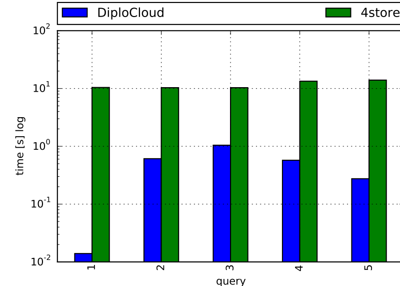

Figure 14: Query execution time for DBPedia running on 16 nodes

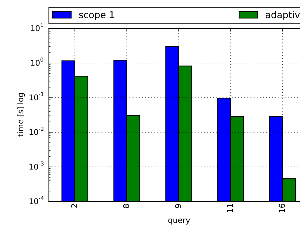

Figure 15: Scope-1 and adaptive partitioning on the most complex LUBM queries for 4 nodes.

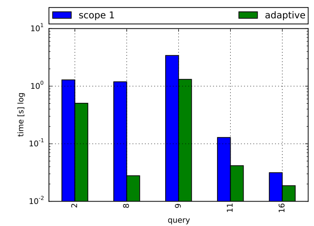

Figure 16: Scope-1 and adaptive partitioning on the most complex LUBM queries for 8 nodes.

Join Analysis: In order to better understand the above results, we made a small query execution analysis (see Table 5) on the LUBM workload, counting the number of joins for DiploCloud (counting the number of joins between molecules for scope-1 / adaptive molecules), 4store (by inspecting the query plans given by the system), and RDF-3X GraphPartitioning (using EXPLAINs). For the RDF-3X GraphPartitioning approach, we report both distributed joins (first number) and local joins (second number). We observe that DiploCloud avoids almost all joins even for complex queries.

Queries and Results Analysis: The queries in Table 5 can be classified into three main categories:

- relatively simple queries with a small output, which do not exhibit any significant difference when changing the kind

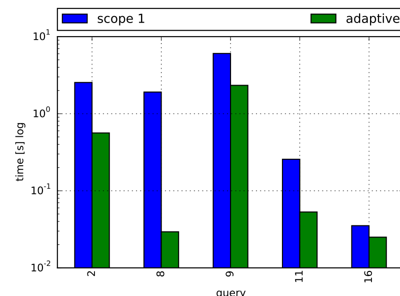

Figure 17: Scope-1 and adaptive partitioning on the most complex LUBM queries for 16 nodes.

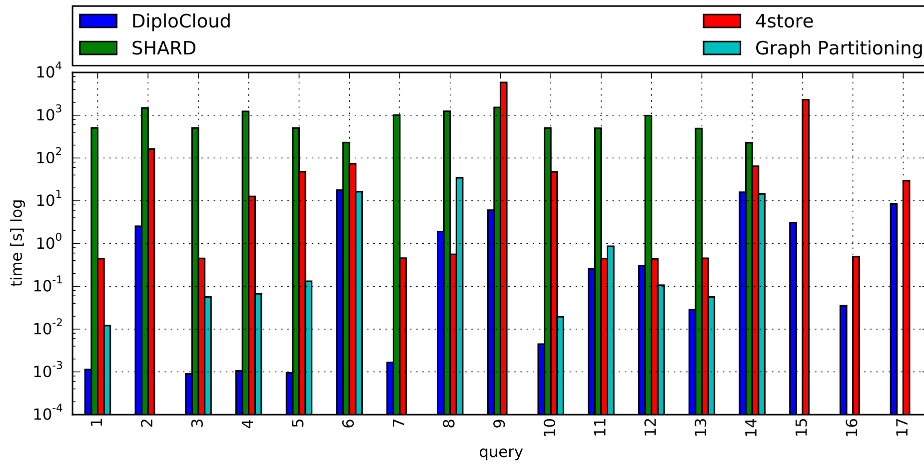

Figure 11: Query execution time for 16 nodes and 1600 universities LUBM data set

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
DiploCloud	0	1/0	0	0	0	0	1	1/0	1/0	0	1/0	1	0	0	0	1/0	0
4store	1	5	1	5	2	1	7	5	11	2	2	2	2	0	3	2	1
RDF3X Part.	0+1	2+5	0+1	0+5	0+2	0+1	2+5	1+5	2+9	0+2	1+2	1+3	0+2	0	-	-	-

Table 5: Joins analysis for several system on the LUBM workload (Distributed Environment). For DiploCloud scope-1/adaptive molecules.

of partitioning (e.g., queries 1,3,10,13); for those kinds of queries DiploCloud significantly outperforms other solutions because of our template and indexing strategies. Those queries are executed on Workers independently, fully in parallel, and results are sent to the Master.

- queries generating a big result set, where the main factor then revolves around transferring data to the master node (e.g., queries 6,14,17); for those queries, DiploCloud is often closer to the other systems and suffers from the (potentially) high network latency associated with cloud environments.
- queries which typically require a distributed join, and for which the partitioning plays a significant role; DiploCloud performs very well on those queries (since most joins can be pre-computed in our molecules), with the exception of query 8, which is also characterized by a big output. For such queries, we differentiate two kinds of joins as briefly evoked above:
  - distributed joins (where we distribute intermediate results among the Workers and then process local joins in parallel); for that kind of queries the influence of the partitioning is not significant, though the collocation of molecules on the same node speedups the exchange of intermediate results, and hence the resulting query execution times
  - centralized joins; when a distributed join is too costly, the intermediate results are shipped to the master node where the final join is performed. We note that for queries 11 and 12, which are based on molecules indirectly related through one particular object, all work is performed on one node, where the particular object is located; that is the reason why this partitioning performs slower for those queries.

As presented above, DiploCloud often outperforms the other solutions in terms of query execution time, mainly thanks to the fact that related pieces of data are already collocated in the molecules. For example for query 2, DiploCloud has to perform only one join (or zero if we adapt the molecules) since all data related to the elements queried (e.g. GraduateStudent or Department) are located on one Worker and are in addition directly collocated in memory; The only thing DiploCloud has to do in this case is to retrieve the list of elements on each Worker and to send it back to the Master, where it either performs a distributed hash-join (if we have molecules of scope-1), or directly takes the result as is (if molecules are adapted). We have similar situations for queries 8, 9, 11, and 16. For query 7, we cannot take advantage of the pre-computed joins since we store RDF data as a directed graph and this particular query traverses the graph in the opposite direction (this is typically one kind of query DiploCloud is not optimized for at this stage). For the remaining queries, we do not require to perform any join at all, and can process the queries completely in parallel on the Workers and send back results to the Master, while the other systems have to take into account the intermediate joins (either locally or in a distributed fashion). Another group of queries for which DiploCloud should be further optimized are queries with high numbers of returned records, like the queries 6 or 14. In some cases we still outperform other systems for those queries, but the difference is not as significant.

			4 workers	8 workers	16 workers
DiploCloud	master	memory (GB)	3.2	3.2	3.2
		loading time (sec)	1285	296	296
	per worker	memory (GB)	3.1	1.6	0.82
		loading time (sec)	28	14	7
4store		loading time (sec)	537	1284	1313

Table 7: Load times and size of the databases for the DBPedia data set (Distributed Environment).

Data Loading: Table 6 gives the loading times for 4store and DiploCloud using the LUBM datasets and different partitioning strategies. We observe that the size taken by the deeper molecules (scope 2) rapidly grows, though the adaptive molecules strike a good balance between depth and size (we loaded the data according to the final version of the adaptive partitioning in that case in order to have comparable results for all variants). Using our parallel batch-loading strategies and adaptive partitioning, DiploCloud is more than 10 times faster than 4store at loading

			4 workers			8 workers			16 workers		
		molecules configuration	scope-1	scope-2	adaptive	scope-1	scope-2	adaptive	scope-1	scope-2	adaptive
DiploCloud	master	memory (GB)	3.1	3.1	3.1	6.2	6.2	6.2	12.4	12.4	12.4
		loading time (sec)	157	154.8	158	372	374	371.83	786	796	784
	per worker	memory (GB)	2.32	6.06	3.35	2.41	6.27	3.42	2.7	6.45	4
		loading time (sec)	11.72	43	26.38	12	66	37.5	39	115	85
4store		loading time (sec)		226			449			893	

Table 6: Load times and size of the databases for the LUBM data set (Distributed Environment).

data for the biggest deployment. Table 7 reports the corresponding numbers for the DBPedia dataset.

EC2 Deployment: Finally, to evaluate how DiploCloud performs in bigger cloud environments, we deployed it on Amazon EC2 instances[11]. We picked an M3 Extra Large Instance for the Master, and M1 Large Instances for the Workers, and loaded the LUBM 1600 dataset on 32 and 64 nodes. The results (see Figures 18) are comparable to those obtained on our own cluster, though slower, due to the larger network latency on EC2 (hence emphasizing once more the importance of minimizing distributed operations in the cloud, as DiploCloud does).

We also tested out adaptive partitioning approach on the EC2 infrastructure. The results are available in Figures 19 and 20 (log-scale). Here again we show that by co-locating all frequently queried elements we can significantly increase the performance. Co-location is especially important in environments where the network is not reliable so that we can minimize the amount of transferred data. We performed a small analysis of the network latency in that case. We measured the time spent by the Workers and Master on query execution only and discovered that the network overhead represents between 40% and 70% of the total execution time.

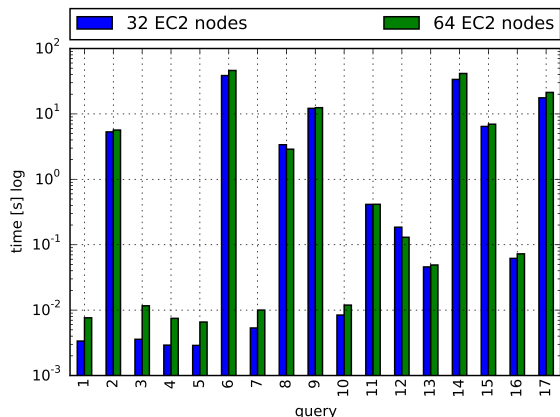

Figure 18: Query execution time on Amazon EC2 for 1600 Universities from LUBM dataset.

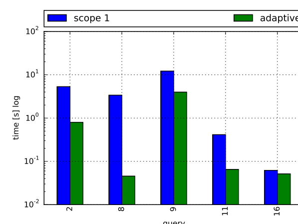

Figure 19: Scope-1 and adaptive partitioning on Amazon EC2 (32 Nodes) for 1600 Universities from LUBM dataset.

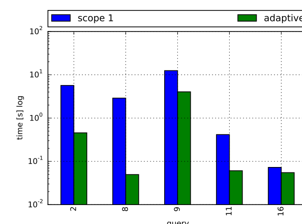

Figure 20: Scope-1 and adaptive partitioning on Amazon EC2 (64 Nodes) for 1600 Universities from LUBM dataset.

8 Conclusions

DiploCloud is an efficient and scalable system for managing RDF data in the cloud. From our perspective, it strikes an optimal balance between intra-operator parallelism and data co-location by considering recurring, fine-grained physiological RDF partitions and distributed data allocation schemes, leading however to potentially bigger data (redundancy introduced by higher scopes or adaptive molecules) and to more complex inserts and updates. DiploCloud is particularly suited to clusters of commodity machines and cloud environments where network latencies can be high, since it systematically tries to avoid all complex and distributed operations for query execution. Our experimental evaluation showed that it very favorably compares to state-of-the-art systems in such environments. We plan to continue developing DiploCloud in several directions: First, we plan to include some further compression mechanisms (e.g., HDT [45]). We plan to work on an automatic templates discovery based on frequent patterns and untyped elements. Also, we plan to work on integrating an inference engine into DiploCloud to support a larger set of semantic constraints and queries natively. Finally, we are currently testing and extending our system with several partners in order to manage extremely-large scale, distributed RDF datasets in the context of bioinformatics applications.

11. http://aws.amazon.com/ec2/instance-types/

Acknowledgment

This work is supported by the Swiss National Science Foundation under grant number PP00P2_153023.

References

[1] K. Aberer, P. Cudre-Mauroux, M. Hauswirth, and T. van Pelt, "GridVine: Building Internet-Scale Semantic Overlay Networks," in *International Semantic Web Conference (ISWC)*, 2004.

[2] P. Cudré-Mauroux, S. Agarwal, and K. Aberer, "GridVine: An Infrastructure for Peer Information Management," *IEEE Internet Computing*, vol. 11, no. 5, 2007.

[3] M. Wylot, J. Pont, M. Wisniewski, and P. Cudré-Mauroux, "dipLODocus[RDF]: short and long-tail RDF analytics for massive webs of data," in *Proceedings of the 10th international conference on The semantic web - Volume Part I*, ser. ISWC'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 778–793. [Online]. Available: http://dl.acm.org/citation.cfm?id=2063016.2063066

[4] M. Wylot, P. Cudre-Mauroux, and P. Groth, "TripleProv: Efficient Processing of Lineage Queries in a Native RDF Store," in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW '14. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2014, pp. 455–466.

[5] M. Wylot, P. Cudré-Mauroux, and P. Groth, "Executing Provenance-Enabled Queries over Web Data," in *Proceedings of the 24rd International Conference on World Wide Web*, ser. WWW '15. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2015.

[6] B. Haslhofer, E. M. Roochi, B. Schandl, and S. Zander, "Europeana RDF Store Report," in *University of Vienna, Technical Report*, 2011, http://eprints.cs.univie.ac.at/2833/1/europeana_ts_report.pdf.
[7] Y. Guo, Z. Pan, and J. Heflin, "An evaluation of knowledge base systems for large OWL datasets," in *In International Semantic Web Conference*. Springer, 2004, pp. 274–288.
[8] Faye, O. Cure, and Blin, "A survey of RDF storage approaches," *ARIMA Journal*, vol. 15, pp. 11–35, 2012.
[9] B. Liu and B. Hu, "An Evaluation of RDF Storage Systems for Large Data Applications," in *Semantics, Knowledge and Grid, 2005. SKG '05. First International Conference on*, nov. 2005, p. 59.
[10] Z. Kaoudi and I. Manolescu, "Rdf in the clouds: A survey," *The VLDB Journal*, pp. 1–25, 2014.
[11] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," *Proceeding of the VLDB Endowment (PVLDB)*, vol. 1, no. 1, pp. 1008–1019, 2008.
[12] T. Neumann and G. Weikum, "RDF-3X: a RISC-style engine for RDF," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 1, no. 1, pp. 647–659, 2008.
[13] A. Harth and S. Decker, "Optimized Index Structures for Querying RDF from the Web," in *IEEE LA-WEB*, 2005, pp. 71–80.
[14] M. Atre and J. A. Hendler, "BitMat: a main memory bit-matrix of RDF triples," in *The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*. Citeseer, 2009, p. 33.
[15] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds, "Efficient RDF Storage and Retrieval in Jena2," in *SWDB'03*, 2003, pp. 131–150.
[16] A. Owens, A. Seaborne, N. Gibbins *et al.*, "Clustered TDB: a clustered triple store for Jena," 2008.
[17] E. PrudHommeaux, A. Seaborne *et al.*, "SPARQL query language for RDF," *W3C recommendation*, vol. 15, 2008.
[18] Y. Yan, C. Wang, A. Zhou, W. Qian, L. Ma, and Y. Pan, "Efficient Indices Using Graph Partitioning in RDF Triple Stores," in *Proceedings of the 2009 IEEE International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1263–1266. [Online]. Available: http://portal.acm.org/citation.cfm?id=1546683.1547484
[19] L. Ding, Y. Peng, P. P. da Silva, and D. L. McGuinness, "Tracking RDF Graph Provenance using RDF Molecules," in *International Semantic Web Conference*, 2005.
[20] L. Zou, J. Mo, L. Chen, M. T. Oezsu, and D. Zhao, "gStore: Answering SPARQL Queries via Subgraph Matching," *PVLDB*, vol. 4, no. 8, 2011.
[21] M. Bröcheler, A. Pugliese, and V. Subrahmanian, "Dogma: A disk-oriented graph matching algorithm for rdf databases," in *The Semantic Web-ISWC 2009*. Springer, 2009, pp. 97–113.
[22] H. Kim, P. Ravindra, and K. Anyanwu, "From SPARQL to MapReduce: The Journey Using a Nested TripleGroup Algebra," *PVLDB*, vol. 4, no. 12, pp. 1426–1429, 2011.
[23] A. Schätzle, M. Przyjaciel-Zablocki, A. Neu, and G. Lausen, "Sempala: Interactive SPARQL Query Processing on Hadoop," in *The Semantic Web–ISWC 2014*. Springer, 2014, pp. 164–179.
[24] M. Kornacker and J. Erickson, "Cloudera Impala: real-time queries in Apache Hadoop, for real," 2012.
[25] P. Cudr-Mauroux, I. Enchev, S. Fundatureanu, P. Groth, A. Haque, A. Harth, F. L. Keppmann, D. Miranker, J. Sequeda, and M. Wylot, "NoSQL Databases for RDF: An Empirical Evaluation," in *International Semantic Web Conference*, 2013.
[26] O. Erling and I. Mikhailov, "RDF Support in the Virtuoso DBMS," in *Networked Knowledge-Networked Media*. Springer, 2009, pp. 7–24.
[27] S. Harris, N. Lamb, and N. Shadbolt, "4store: The design and implementation of a clustered RDF store," in *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, 2009, pp. 94–109.
[28] K. Rohloff and R. E. Schantz, "Clause-iteration with mapreduce to scalably query datagraphs in the shard graph-store," in *Proceedings of the fourth international workshop on Data-intensive distributed computing*. ACM, 2011, pp. 35–44.
[29] J. Huang, D. J. Abadi, and K. Ren, "Scalable SPARQL Querying of Large RDF Graphs," *PVLDB*, vol. 4, no. 11, pp. 1123–1134, 2011.
[30] K. Hose and R. Schenkel, "WARP: Workload-Aware Replication and Partitioning for RDF," in *DESWEB*, 2013.
[31] K. Lee and L. Liu, "Scaling queries over big rdf graphs with semantic hash partitioning," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1894–1905, 2013.
[32] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, "A distributed graph engine for web scale RDF data," in *Proceedings of the VLDB Endowment*, vol. 6, no. 4. VLDB Endowment, 2013, pp. 265–276.
[33] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald, "Triad: A distributed shared-nothing rdf engine based on asynchronous message passing," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 289–300.
[34] R. Ramamurthy, D. J. DeWitt, and Q. Su, "A case for fractured mirrors," in *Proceedings of the 28th international conference on Very Large Data Bases*, ser. VLDB '02. VLDB Endowment, 2002, pp. 430–441. [Online]. Available: http://portal.acm.org/citation.cfm?id=1287369.1287407
[35] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudré-Mauroux, and S. Madden, "HYRISE - A Main Memory Hybrid Storage Engine," *PVLDB*, vol. 4, no. 2, pp. 105–116, 2010.
[36] P. Cudré-Mauroux, E. Wu, and S. Madden, "The Case for RodentStore, an Adaptive, Declarative Storage System," in *Biennial Conference on Innovative Data Systems Research (CIDR)*, 2009.
[37] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: a distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 15–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267308.1267323
[38] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-Store: A Column Oriented DBMS," in *International Conference on Very Large Data Bases (VLDB)*, 2005.
[39] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *Web Semant.*, vol. 3, pp. 158–182, October 2005. [Online]. Available: http://dx.doi.org/10.1016/j.websem.2005.06.005
[40] G. Demartini, I. Enchev, M. Wylot, J. Gapany, and P. Cudré-Mauroux, "BowlognaBenchBenchmarking RDF Analytics," in *Data-Driven Process Discovery and Analysis*. Springer, 2012, pp. 82–102.
[41] C. Becker, "RDF store benchmarks with DBpedia," 2008, http://wifo5-03.informatik.uni-mannheim.de/benchmarks-200801/.
[42] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux, "OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases," *PVLDB*, vol. 7, no. 4, pp. 277–288, 2013.
[43] J. Aasman, "Allegro graph: RDF triple database," Technical report. Franz Incorporated, 2006. ur l: http://www. franz. com/agraph/allegrograph/(visited on 10/14/2013)(cited on pp. 52, 54), Tech. Rep., 2006.
[44] A. Kiryakov, D. Ognyanov, and D. Manov, "OWLIM–a pragmatic semantic repository for OWL," in *Web Information Systems Engineering–WISE 2005 Workshops*. Springer, 2005, pp. 182–192.
[45] M. A. Martnez-Prieto, M. Arias, and J. D. Fernndez, "Exchange and Consumption of Huge RDF Data," in *The Semantic Web: Research and Applications*. Springer, 2012, pp. 437–452.

Marcin Wylot is a PhD student at the University of Fribourg in Switzerland. Since 2011 he is a member of the eXascale Infolab led by Professor Philippe Cudré-Mauroux. He received his MSc in computer science at the University of Lodz in Poland in 2010, doing part of his studies at the University of Lyon in France. During his studies he was also gaining professional experience working in various industrial companies. His main research interests revolve around databases for Semantic Web, provenance in RDF data, and Big Data processing.
Webpage: http://mwylot.net

Philippe Cudré-Mauroux is a Swiss-NSF Professor and the director of the eXascale Infolab at the University of Fribourg in Switzerland. Previously, he was a postdoctoral associate working in the Database Systems group at MIT. He received his Ph.D. from the Swiss Federal Institute of Technology EPFL, where he won both the Doctorate Award and the EPFL Press Mention in 2007. Before joining the University of Fribourg, he worked on distributed information and media management for HP, IBM Watson Research (NY), and Microsoft Research Asia. He was Program Chair of the International Semantic Web Conference in 2012 and General Chair of the International Symposium on Data-Driven Process Discovery and Analysis in 2012 and 2013. His research interests are in next-generation, Big Data management infrastructures for non-relational data.
Webpage: http://exascale.info/phil