International Journal of Research
Available at https://edupediapublications.org/journals
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 04 Issue 07
June 2017

Efficient and Scalable control of information in the Cloud

1.K.SUKANYA Department of cse , sukanyamouni95@gmail.com

2.U.SANDHYA, ASSISTANT PROFESSOR, ummadisettysandhya@gmail.com

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.

I 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., largescale, 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 Grid- Vine 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 internode operations.

II 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 three-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 HBase2 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 RDF3 uses Cassandra and hash-partitioning to distribute the RDF tiples. It stores dataas 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 cloud4 [25].

III 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 agiven instance) as well as vertically (to co-locate series of entities or values attached to similar instances). 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 Diplo- Cloud is a hybrid system, following the terminology used for approaches such as Fractured Mirrors or our own recent Hyrise system . Molecule clusters are used in two ways in our system: to logically group sets of related URIs and literals in the hashtable

(thus, pre-computing joins), and to physically co-locate information relating to a given object on disk and in mainmemory 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.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.

3.2 Templates

One of the key innovations of DiploCloud revolves around the use of declarative storage patterns to efficiently collocate 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

3.3 Molecules

DiploCloud uses physiological RDF partitioning and molecule patterns to efficiently co-locate RDF data in distributed settings.. Molecules have three key advantages in our context: Molecules represent the ideal tradeoff between collocation 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) are suboptimal as well, since in that case too many related triples are co-located, thus inhibiting parallel processing . 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/heradvisor) that are frequently co-accessed.

IV DATA PARTITIONING AND ALLOCATION

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 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) consider unlabeled graphs mostly, and hence are totally agnostic tothe 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 use 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.

V COMMON OPERATIONS

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

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

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 logstructured (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-placeupdates, 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 the corresponding structures are flagged in the key index, and new structures are maintained in writeoptimized 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),y. 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 gives a high-level description of our distributed query execution process highlighting where particular operations are performed in our system.

Algorithm

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.

VI CONCLUSION

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 collocation 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). 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.

REFERENCES

[1] K. Aberer, P. Cudre-Mauroux, M. Hauswirth, and T. van Pelt, "GridVine: Building Internet-scale semantic overlay networks," in Proc. Int. Semantic Web Conf., 2004, pp. 107–121.

[2] P. Cudr_e-Mauroux, S. Agarwal, and K. Aberer, "GridVine: An infrastructure for peer information management," IEEE Internet Comput., vol. 11, no. 5, pp. 36–44, Sep./Oct. 2007.

[3] M. Wylot, J. Pont, M. Wisniewski, and P. Cudr_e-Mauroux. (2011). dipLODocus[RDF]: Short and long-tail RDF analytics for massive webs of data. Proc. 10th Int. Conf. Semantic Web - Vol. Part I, 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 Proc. 23rd Int. Conf. World Wide Web, 2014, pp. 455–466.

[5] M. Wylot, P. Cudr_e-Mauroux, and P. Groth, "Executing provenance- enabled queries over web data," in Proc. 24th Int.

Conf. World Wide Web, 2015, pp. 1275–1285.

[6] B. Haslhofer, E. M. Roochi, B. Schandl, and S. Zander. (2011). Europeana RDF store report. Univ. Vienna, Wien, Austria, Tech. Rep. [Online]. Available: 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 Proc. Int. Semantic Web Conf., 2004, pp. 274–288.

[8] Faye, O. Cure, and Blin, "A survey of RDF storage approaches," ARIMA J., vol. 15, pp. 11–35, 2012.

[9] B. Liu and B. Hu, "An Evaluation of RDF Storage Systems for Large Data Applications," in Proc. 1st Int. Conf. Semantics, Knowl. Grid, Nov. 2005, p. 59.

[10] Z. Kaoudi and I. Manolescu, "RDF in the clouds: A survey," VLDB J. Int. J. Very Large Data Bases, vol. 24, no. 1, pp. 67–91, 2015.

[11] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," Proc. VLDB Endowment, vol. 1, no. 1, pp. 1008–1019, 2008.

[12] T. Neumann and G. Weikum, "RDF-3X: A RISC-style engine for RDF," Proc. VLDB Endowment, vol. 1, no. 1, pp. 647–659, 2008.

[13] A. Harth and S. Decker, "Optimized index structures for querying RDF from the web," in Proc. IEEE 3rd Latin Am. Web Congr., 2005, pp. 71–80.

[14] M. Atre and J. A. Hendler, "BitMat: A main memory bit-matrix of RDF triples," in Proc. 5th Int. Workshop Scalable Semantic Web Knowl. Base Syst., 2009, p. 33.

[15] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds, "Efficient RDF Storage and Retrieval in Jena2," in Proc. 1st Int. Workshop Semantic Web Databases, 2003, pp. 131–150.. Query execution time on Amazon EC2 for 1,600 Universities from LUBM dataset.

[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. (2009). Efficient indices using graph partitioning in RDF triple stores. Proc.

[19] L. Ding, Y. Peng, P. P. da Silva, and D. L. McGuinness, "Tracking RDF graph provenance using RDF molecules," in Proc. Int. Semantic Web Conf., 2005, p. 42.

[20] S. Das, D. Agrawal, and A. El Abbadi, "G-store: A scalable data store for transactional multi key access in the cloud," pp. 163–174, 2010.

[21] M. Br€ocheler, A. Pugliese, and V. Subrahmanian, "Dogma: A diskoriented graph matching algorithm for RDF databases," in Proc. 8th Int. Semantic Web Conf., 2009, pp. 97–113.

[22] H. Kim, P. Ravindra, and K. Anyanwu, "From SPARQL to Map- Reduce: The journey using a Nested TripleGroup algebra," Proc. VLDB Endowment, vol. 4, no. 12, pp. 1426–1429, 2011.

[23] A. Sch€atzle, M. Przyjaciel-Zablocki, A. Neu, and G. Lausen, "Sempala: Interactive SPARQl query processing on Hadoop," in Proc. 13th Int. Semantic Web Conf., 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 Proc. 12th Int. Semantic Web Conf., 2013, pp. 310–325