ISSN No: 2348-4845
International Journal & Magazine of Engineering, Technology, Management and Research
A Peer Reviewed Open Access International Journal

An Efficient Evaluation Scalable Management of RDF Data in the Cloud

Swapna Gangapuram
HOD,
Department of CSE,
Siddhartha Institute of Technology & Sciences.

B.Pragathi
M.Tech Student,
Department of CSE,
Siddhartha Institute of Technology & Sciences.

Abstract:
Despite recent advances in distributed Resource Description Frame work (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. Sharing 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 Diplo Cloud, an efficient and scalable distributed RDF data management system for the cloud. Contrary to previous approaches, Diplo Cloud runs a physiological analysis of both instance and schema information prior to partitioning the data. In this paper, we describe the architecture of Diplo Cloud, its main data structures, as well as the new algorithms we use to partition and distribute data. We also present an extensive evaluation of Diplo Cloud showing that our system is often two orders of magnitude faster than state-of-the-art systems on standard workloads.

Keywords:
RDF, triple stores, cloud computing, Big data

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) and on distributed selections, projections, and joins. Our own Grid- Vine system was one of the first systems to do so in the context of large-scale decentralized RDF management.

ISSN No: 2348-4845
International Journal & Magazine of Engineering, Technology, Management and Research
A Peer Reviewed Open Access International Journal

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; · a new system architecture for handling fine-grained RDF partitions in large-scale;
- novel data placement techniques to co-locate semantically related pieces of data; new data loading and query execution strategies taking advantage of our system's data partitions and indices;
- an extensive experimental evaluation showing that our system is often two orders of magnitude faster than state-of-the-art systems on standard workloads . DiploCloud builds on our previous approach diplodocus ½RDF, an efficient single node triple store. The system was also extended in TripleProv to support storing, tracking, and querying provenance in RDF query processing.

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 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 suggests to index RDF data using six possible indices, one for each permutation of the set of columns in the triple table. RDF-3X and YARS follow a similar approach. BitMat 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. 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. 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 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. 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. 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 organize data in adjacency list tables. Each vertex is represented as an entry in the table with a list of its outgoing edges and neighbors. To index vertices, they build an S-tree in their adjacency list table to reduce the search space.

Brocheler et al. propose a balanced binary tree where each node containing a sub graph 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) uses Map Reduce to process distributed RDF data. RAPID+ extends Apache Pig and enables more efficient SPARQL query processing on Map Reduce 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 builds on top of Impala 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 data as four indices (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 . Our previous Grid Vine system uses a triple-table storage approach and hash-partitioning to distribute RDF data over decentralized P2P networks. YARS2,5 Virtuoso6 , 4store , and SHARD hash partition triples across multiple machines and parallelize the query processing. Virtuoso 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., 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 the main idea of which is to iterate over all clauses and incrementally bind variables and satisfy constrains.

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.

ISSN No: 2348-4845
International Journal & Magazine of Engineering, Technology, Management and Research
A Peer Reviewed Open Access International Journal

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). 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.

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], 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. 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 URI sand 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.

Templates:

One of the key innovations of DiploCloud revolves around the use of declarative storage patterns [36] 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 (see for instance Fig. 1, where "Student" is the root node of the molecule, and "Student ID" 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).

ISSN No: 2348-4845
International Journal & Magazine of Engineering, Technology, Management and Research
A Peer Reviewed Open Access International Journal

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. 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 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. A 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 sub lists, 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.

System Overview

Diplo-Cloud 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 Big Table), where one (Master) node is responsible for interacting with the clients and orchestrating the operations performed by the other (Worker) nodes.

Master Node:

The Master node is composed of three main sub components: a key index (c.f.), in charge of encoding URIs and literals into compact system identifiers and of translating them back, a partition manager (c.f.), responsible for partitioning the RDF data into recurring subgraphs, and a distributed query executor (c.f.), 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).

Worker Nodes:

The Worker nodes hold the partitioned data and its corresponding local indices, and are responsible for running sub queries 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 sub graphs, and iii) a molecule index, storing for each key the list of molecules where the key can be found.

DATA PARTITIONING AND ALLOCATION

Triple-table and property-table hash-partitioning are currently the most common partitioning schemes for distributed RDF systems. While simple, such hash-partitioning 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 is also from our perspective inappropriate

ISSN No: 2348-4845
International Journal & Magazine of Engineering,
Technology, Management and Research
A Peer Reviewed Open Access International Journal

in a cloud context, for three main reasons: Loss of semantics: standard graph partitioning tools 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 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. Diplo Cloud'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 physiological9 partitioning.

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.,ð student; takes; courseÞ and ðcourse; hasid; xsd : integerÞ) 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. 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.

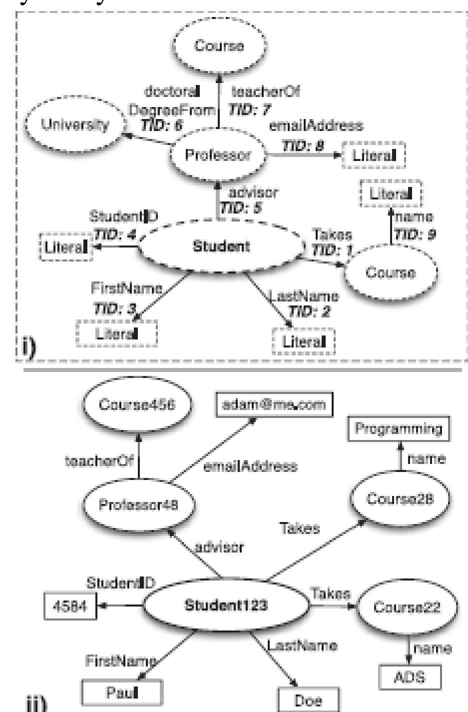

Fig.. A molecule template (i) along with one of its RDF molecules (ii).

ISSN No: 2348-4845
International Journal & Magazine of Engineering, Technology, Management and Research
A Peer Reviewed Open Access International Journal

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. Adaptive partitioning. Finally, Diplo Cloud'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 , 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.

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 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). We plan to work on an automatic templates discovery based on frequent patterns and un-typed 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.

Future Enhancement:

Many RDF systems rely on hash-partitioning and on distributed selections, projections, and joins. Our own Grid Vine system was one of the first systems to do so in the context of large-scale decentralized RDF management.

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.

ISSN No: 2348-4845
International Journal & Magazine of Engineering, Technology, Management and Research
A Peer Reviewed Open Access International Journal

[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.