Evaluation of Distributed Semantic Databases

Philipp Trucksäß
Advisor: Jan Seeger
Seminar Future Internet SS2018
Chair of Network Architectures and Services
Departments of Informatics, Technical University of Munich
Email: philipp.trucksaess@in.tum.de

ABSTRACT
In this paper, several implementations of RDF data stores are compared, analyzing their unique innovations and evaluating their benefits and shortcomings for various applications. The goal is to give an overview of popular implementations and identify potential for further research.

Keywords
RDF, SPARQL, Distributed Semantic Databases

1. INTRODUCTION
With the rising popularity of the Semantic Web, Semantic Databases are of increasing importance, as in the often cited DBPedia project, the Linked Open Data platform of Wikipedia [7]. The goal of the Semantic Web is to provide a way to describe data and relationships between data items, which makes it easier to process for machines. This is particularly relevant because modern systems have to process an ever increasing amount of information. For instance, IoT applications are becoming more prevalent with the growing acceptance of smart home appliances. Those make use of sensor networks which produce gigabytes of data every seconds, and with them comes a need for a standardized way to integrate data from different sources, and to store and process that data in an efficient way. Semantic databases can help with that feat.

The Resource Description Framework, usually known as RDF, standardizes the interchange of semantic data on the web. The term “Semantic Data” refers to the structure of the data as “Subject”, “Predicate” (or property), “Object”, which is analogous to that of an English sentence. Because of this focus on triples, the respective databases are called Triple Stores.

The subjects and objects can be thought of as nodes in a graph, where edges between those nodes represent predicates. Even more so than for a traditional relational database system, this graph structure calls for a distributed implementation.

While some approaches simply provide an API for RDF and its associated query language SPARQL on top of an established central database architecture, I present some of the more innovative distributed approaches and analyze their capabilities and unique features.

2. BACKGROUND
RDF as a standard is maintained by the W3C. For the purpose of comparing different RDF data stores it is helpful to have a basic understanding of the RDF schema and the SPARQL query language, which is most commonly used for examples and benchmarks by the presented stores.

2.1 RDF Schema
The most recent version of the W3C RDF Schema Recommendation (at the time of writing) specifies the vocabulary to be used for RDF data [16] [17]. It defines an extension to RDF, focused on describing groups and their relationships. At its core, the schema provides a framework to model relationships and properties of different kinds of resources, which are organized in classes. The term is used not unlike its meaning known from object-oriented programming languages in this context. In RDF triples, these resources can take the place of subject, property (predicate) or object. There are several different concrete syntaxes implementing the schema, such as the simple JSON RDF or Turtle. Those can be extended to impose entailment regimes, e.g. by OWL, the web ontology language which can describe complex logical contexts between resources. The only concrete syntax used in the standard is namespace: name.

Resources are specified by literal strings or numerals and pairs of resources can be related to each other via properties. Those properties implement hierarchical relationships, so a more specific property implies a more general property, that it is a sub-property of. Two properties are special; the range property refers to resources with a certain property, which themselves are instances of a number of classes. The domain property specifies, that resources with a specific property have to be instances of one or multiple classes. In the same vein, several other descriptors of hierarchical relationships and groupings are available.

Further, the RDF vocabulary also allows specifying labels and comments to provide human-readable descriptions, and set-theoretical groupings like rdf:Bag, as well as a syntax to describe RDF statements, e.g. rdf:subject, and more.

2.2 SPARQL
The “SPARQL Query Language for RDF” specifies a language and a protocol for interacting with RDF graph content [15]. SPARQL supports queries which use similar keywords to regular SQL queries, as seen in 1.

1.
PREDEFINED
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns/>
SELECT ?name
WHERE {
This example shows the use of triple patterns in the WHERE clause, where a ? is a variable, and literal values are bound to the query solution. The example also demonstrates how more complex relationships between multiple variables can be queried. Those relationships represent paths in the data graph. A common pattern in SPARQL is the introduction of several variables which occur in several places throughout the query, often connected by multiple predicates.

A problem which all presented data stores try to solve one way or another, is that SPARQL queries with many interrelated variables tend to be quite expensive if the underlying data storage model is based on a relational database. The relational data model was designed to retrieve information about indexable data items. In the semantic data model, quick access to any of the stored resources is needed, so, as shown later, a need for extensive index structures exists. Each node in the query graph, in which two edges meet, implies the need for a table join. These joins are unavoidable in relational databases, but they do in general not scale well, as they are so called “pipeline breakers”, so they should be kept to a minimum. The number of joins that are required even for simple SPARQL queries like 1 is significantly higher than usually seen in SQL queries of comparable complexity. Therefore, an alternative approach to relational modelling of RDF data, which avoids large numbers of joins is presented e.g. in 3.2.1. The method used there is graph exploration, which lends itself to distributed parallel execution while only requiring one final join phase in the end, but it requires a more complex message passing system.

3. OVERVIEW OF RELEVANT DISTRIBUTED RDF SYSTEMS

With the diversity of implementations, which have been proposed focusing on optimizing various, but not necessarily mutually exclusive, parts of the RDF architecture, it is not straightforward to draw comparisons between them. A problem which all presented data stores try to solve one way or another, is that SPARQL queries with many interrelated variables tend to be quite expensive if the underlying data storage model is based on a relational database. The relational data model was designed to retrieve information about indexable data items. In the semantic data model, quick access to any of the stored resources is needed, so, as shown later, a need for extensive index structures exists. Each node in the query graph, in which two edges meet, implies the need for a table join. These joins are unavoidable in relational databases, but they do in general not scale well, as they are so called “pipeline breakers”, so they should be kept to a minimum. The number of joins that are required even for simple SPARQL queries like 1 is significantly higher than usually seen in SQL queries of comparable complexity. Therefore, an alternative approach to relational modelling of RDF data, which avoids large numbers of joins is presented e.g. in 3.2.1. The method used there is graph exploration, which lends itself to distributed parallel execution while only requiring one final join phase in the end, but it requires a more complex message passing system.

A problem which all presented data stores try to solve one way or another, is that SPARQL queries with many interrelated variables tend to be quite expensive if the underlying data storage model is based on a relational database. The relational data model was designed to retrieve information about indexable data items. In the semantic data model, quick access to any of the stored resources is needed, so, as shown later, a need for extensive index structures exists. Each node in the query graph, in which two edges meet, implies the need for a table join. These joins are unavoidable in relational databases, but they do in general not scale well, as they are so called “pipeline breakers”, so they should be kept to a minimum. The number of joins that are required even for simple SPARQL queries like 1 is significantly higher than usually seen in SQL queries of comparable complexity. Therefore, an alternative approach to relational modelling of RDF data, which avoids large numbers of joins is presented e.g. in 3.2.1. The method used there is graph exploration, which lends itself to distributed parallel execution while only requiring one final join phase in the end, but it requires a more complex message passing system.

In the following, representatives across the spectrum are evaluated in more depth. In particular, the maturity of the implementation, the type of architecture with respect to unique design decisions, and the reasoning tasks that are supported, are given special attention.

3.1 Cloud Based Triple Stores

Approaches to utilize cloud architectures by building on top of a distributed file system, such as Hadoop’s HDFS used by Hammoud et al. [4] are among the earliest proposed distributed RDF systems. The approach is rather straightforward, as it builds on a preexisting cloud infrastructure, with built in resilience and elasticity, as well as data partitioning backed up by redundancies. The core technology behind these straightforward cloud based triple stores is the MapReduce programming model [2].

The MapReduce implementation brings their functionality to key-value pairs stored in distributed clusters. In the context of e.g. SPARQL queries, the mapping phase produces matches for a query from all relevant partitions, which are then reduced i.e. combined incrementally into intermediate query solutions up to the final answer.

The partitions are simply represented as individual files. An individual file can be used to represent various levels of abstraction over the triples. Each subject can be stored as one line in a file, containing all properties associated with the subject which is sometimes known as horizontal partitioning [7]. To execute a query on such a data structure...
may require a full scan of the data. Alternatively, extensive indexing can be used to represent any constellation of the triples and facilitate direct access for any query, but not without massive data duplication. A popular scheme is the storage of all six possible permutations of subject, predicate, object, each as the key of a key-value store without associated value as used by Zheng et al. [19]. Another popular ordering is the partitioning of one file per property, so all edges of a certain kind can be found in one place without redundant data. Such a vertical partitioning scheme is employed by Husain et al. [5] among others. For the purpose of stability, each partition is usually replicated over several storage points.

### 3.1.1 H2RDF+

An example of a MapReduce based distributed RDF store was introduced with H2RDF+ by Papailiou et al. [11]. It builds upon the NoSQL key-value store HBase, which in turn employs HDFS for data storage. All six permutations of subject, predicate, object per triple are stored as keys with empty values. A separate store is used as a dictionary to map string labels to IDS with different lengths, where more common values receive shorter IDS, utilizing byte level variable length encoding. The partitioning of the data is left to the underlying HBase implementation, which distributes its input into several ranges of key-value mappings. When using MapReduce for RDF queries, results are produced by joining triple patterns over a join variable which they share. H2RDF+ focuses on two efficient merge join algorithms, one for multi-way merge join and one for sort-merge join. For the sorted content of the index table the "MapReduce Merge Join Algorithm" joins multiple triple patterns which have a variable in common. The biggest partition produces its share of the queried range via a map-only job and the other partitions are merge-joined by local scanners.

Intermediate results are unsorted, or rather not ordered by the variable to join over, therefore they are handled by the "MapReduce Sort-Merge Join Algorithm". The biggest partition over the join variable is used to generate a global ordering for the reducers, which can then operate on sorted ranges. If only intermediate results are joined, hash partitioning can be used to perform a hash join instead. The joins can be performed in a distributed or a centralized version, depending on the workload.

To minimize query execution time, the optimal join order is decided for every step iteratively by an online planner which calculates costs based on statistics from previous queries. Only joins of a certain size profit from distributed execution, smaller ones are ran centrally. The planner runs on a master node, but centralized joins can also be executed on any slave node. The merits of the adaptive join execution are more measurable for big inputs and complex queries where it yields performance benefits over simpler approaches. Especially for less selective queries, the use of sorted ranges is useful and it also enables various range queries. Complex reasoning tasks are possible, but they may require an inflationary number of joins, so they can quickly become unfeasible.

### 3.1.2 EAGRE

For distributed queries in a cloud based system, I/O operations need to be coordinated with care, or they represent a significant bottleneck. A MapReduce based implementation with the aim to target that issue is EAGRE ("Entity-Aware Graph compREssion") [19]. EAGRE strives to incorporate semantic and structural data into its data format in a more sophisticated way than a simple key-value store of previous cloud-based RDF databases such as Trinity which will be described in 3.2.1. Queries are answered with the MapReduce method, with a focus on minimizing disk I/O and network traffic or rather on a desirable trade off between network traffic costs and the benefit of distributed I/O. To limit the expenses of MapReduce, intermediate results are estimated using a Bloom Filter and the special Consulting protocol which facilitates the exchange of information about the specific value ranges of each compute node, so the search space can be limited in the scheduling phase. To leverage this range based optimization, the data needs to be partitioned while keeping its inherent order. Zhang et al. [19] limit the evaluation of SPARQL to SELECT queries, but for those the partitioning is favorable for range and order constraints.

EAGRE’s approach to modeling RDF data focuses on the subjects and describes them as Entities, each with a key-value collection of its properties and respective objects. Based on shared description keys, the entities are categorized into Entity Classes, by which they can then be grouped to generate a so called "Compressed RDF Entity Graph". The grouping is first performed in a randomized distributed fashion, and subsequently the resulting structure is partitioned using METIS while preserving the data locality as well as possible. Within each compute node the layout of the entity classes is determined using the Space Filling Curve algorithm for high dimensional data, to effectively place the connected entities together and save unnecessary disk I/O. This rather involved partitioning scheme revolves around I/O optimization, but because of its complexity, it has its downsides when used with dynamic data, as updates of the data set become expensive.

![Table 1: An overview of distributed RDF databases](image-url)
The evaluation of a query then utilizes the **Consulting** to minimize the variable ranges to be read so only a minimum amount of data blocks needs to be read. The final **MapReduce** is postponed until the most beneficial variable set has been determined, so it operates on a minimal data set and network payload is as small as possible.

While disk I/O speeds have increased since EAGRE was first introduced, network communication continues to motivate elaborate scheduling optimization [1].

### 3.1.3 S2RDF

The last implementation to be discussed here, which is based on a cloud store, is **S2RDF** [14]. Under the realization that the query patterns usually encountered with cloud-based triple stores are in practice more limited than what the **SPARQL** specification allows, because the full range of possible queries is often undesirably slow to execute, the relational partitioning schema **ExtVP**, which strives to minimize the input sizes of any kind of query pattern.

The underlying in-memory cluster computing system **Spark** runs on top of **Hadoop** data sources. It uses **Resilient Distributed Datasets** to store the data, which allows parallel operations on the contained elements and provides fault tolerance. Like **MapReduce** the **data parallel computation** model operates on multiple records of data in parallel, and on top of this, **Spark** comes with multiple utilities like the Catalyst optimizer, which is originally intended for heavily optimized computations in a relational database interface. The data is stored in a column layout, with the benefit of performance improving compression while the table schema is preserved. The **ExtVP** **extended** vertical partitioning scheme was designed to minimize I/O, as well as the number of necessary join operations, while not catering to any particular query shape (such as specifically star-shaped queries). The basic vertical partitioning uses one table per property, with one column each for the occurring subjects and objects. This leads to rather unbalanced tables and many intermediate results which are discarded later. To avoid a good part of these, **ExtVP** uses precomputed joins for all pairs of property tables, which was found to be still more space efficient than e.g. storing all six permutations of tuples, as presented in 3.1.1. The possible benefit is greatest, if the join is of high selectivity, which also leads to a memory efficient representation, which provides a good heuristic for cases that justify precomputation.

Queries are processed as algebraic trees, which are traversed bottom-up. The joins are ordered to provide maximum selectivity, i.e. triples with common variables are preferably joined, to limit necessary network traffic. Schätzle et al. [14] found **S2RDF** to be significantly faster than previous frameworks for common query shapes. They also found the speed up for the less common query shapes, which motivated the development in the first place, to be even more pronounced. That also means that complex inference tasks can be performed efficiently, as the precomputed joins reduce the necessary workload. **S2RDF** is a good example, how synergies between meticulously optimized traditional relational database engines and RDF systems can be utilized, in this case in the form of **Spark**.

### 3.2 Graph based triple stores

Many of newer additions to the plethora of RDF data management systems are found in the family of graph based variants. They are usually based on cloud architectures as well, but they use less of their native features, and utilize custom implementations of e.g. index structures instead, as in the work of Zou et al. [20]. They are motivated by the demand for graph operations, which are not trivially supported in **MapReduce** systems, where the connections between the vertices have to be derived from joins as discussed by Zheng et al. [18]. In graph-oriented storage schemes, the nouns of the triples readily provide all edges connected to them by means of adjacency lists. This way, less duplication of data is required and especially more complex queries can be executed faster.

The partitioning is usually done by **METIS**, a software package for graph partitioning where the number of adjacent vertices distributed to separate compute nodes is minimized [8]. Independently of the partitioning scheme, a nonzero amount of fragment boundary crossing edges can not be avoided in general. One possible way of dealing with this is to replicate the n-hop neighboring nodes of a partition’s border nodes for each partition. In a federated method, a query then will have to be able to crawl through any number of partitions, crossing between them when an edge spanning fragment boundaries is followed. However, the federated approach has only limited potential for parallelization [7]. A more efficient use of clustered computing power is achieved by the **Partial Evaluation** method, where matches are first computed for every partition independently, accepting also sub-matches of only parts of the query, and then combining them to reach a final solution. The combination of the partial matches can either be performed on a central server, or in a distributed fashion, which makes better use of the available computing power and also puts less contention on the network connections, but not without an overhead in scheduling and synchronizing network communication [10].

### 3.2.1 Trinity.RDF

**Trinity.RDF** was first presented by Zeng et al. [18] as an attempt to store graph data in its native graph form instead of a set-based approach, in order to support the full range of operations only available for graphs, such as reachability queries. The graph is stored in memory, distributed in a cloud to make random accesses feasible.

The “Sideways Information Passing” technique or **SIP** is employed in conjunction with graph exploration for dynamic optimization of parallel execution. It lets filters on identifiers be shared between compute nodes processing similar such identifiers. The main assumption of Zeng et al. [18] is that graph exploration can be performed more efficiently than equivalent joins in a table-based storage format. The data is partitioned by randomly hashing the nodes across a cluster, where each machine receives a disjoint partition of the graph. The data is committed to **Trinity** key-value stores. One machine is called a “proxy” and represents a master node which coordinates the query plan and assembles the final result from the slaves holding the data partitions. A special server holds mappings from literal strings to unique IDs, so the **Trinity** stores only need to retain fixed-size identifiers and the partitions are more memory efficient. Every **RDF** entity is represented by a node whose identifier is the key for the key-value store, and is used to retrieve adjacency lists of all incoming and outgoing edges, which represent the predicates and respective subjects or objects. In the context of **Trinity.RDF** this key-value store is con-
sidered a “local predicate index”, where predicates can be found for any subject or object. On top of that, the “global predicate index” of each machine maps each predicate to all occurring subjects and objects with that predicate. Because the random partitioning may incur massive costs in network traffic, the adjacency lists are further split up depending on the machine owning the edge’s receiver node, so the internal nodes on one machine can be explored first, and the collective information leading the exploration to the next machine can be sent as a block. *Trinity.RDF* was found to be more memory efficient than other more heavily indexed implementations, but it is important to know that the assumption that graph exploration outperforms join-based query processing proved to be correct only for rather specific complex queries.

### 3.2.2 *TriAD*

Introduced by Gurajada et al. [3], the *TriAD* (for “Triple Asynchronous Distributed”) *RDF* engine puts a strong focus on parallelization as its *shared-nothing* architecture enables several nodes in a cluster, as well as several cores on the same machine to operate on part of a query completely autonomously. It is presented as a direct successor to *Trinity.RDF*, trying to avoid its perceived shortcomings. The essential proposition to enable this high level of scalability is *TriAD’s* custom asynchronous message passing protocol, which allows strongly multithreaded query execution without a big synchronization overhead. Still, the compute nodes are hierarchically ordered, with a master node whose purpose it is to receive and optimize all queries. The master node is also keeping track of the summary graph, a simplified mapping of the RDF data graph, where each summarized node contains enough information about a set of data nodes to prune the nodes which do not contain relevant triples from a query plan.

The slave nodes contain indices for all six permutations of the triples, so all possible point queries can be answered directly via the most appropriate index structure. To partition the graph between the slave nodes, triples are hashed and distributed horizontally, depending on the summarized super-node they belong to, to allow for pruning. The horizontal partitioning is suited for graph exploration, which is used to answer queries starting from opportune nodes and collecting data while traversing potentially many of the graph partitions. Practically, the range of exploration is still limited enough to be efficient because of the locality-preserving partitioning scheme and the join-ahead pruning algorithm. However, there is only direct support for point queries, and range queries may require a scan of the entire graph, as the ranges may spread over several partitions.

### 3.2.3 *gStore*

Another interesting graph based representation of RDF data is *gStore* [20]. More so than other implementations presented here, *gStore* stores data in custom data structures specifically tailored to the needs of RDF graphs. At their core is the *vertex signature* representation of the entities and classes present in the graph. This binary representation works similarly to a *Bloom Filter* over the adjacent edges and neighboring vertices. Each vertex is identified by an adjacency list, where edges and neighbors are hashed and the results are combined via bitwise OR. This encoding is used to create a signature graph. The query graph is encoded analogously, and a *VS*-tree, a vertex signature tree, is used to answer the query. The *VS*-tree summarizes the signature graph at different resolutions, so while it is matched against the query down to the leaf nodes, the search space can be pruned efficiently on each level. Eventually the positive matches generate small materialized aggregates. For caching purposes all transactions are collected in a transaction database managed by a trie structure called *T-index*. The queried predicates are ordered by frequency in order to minimize the number of materialized views that have to be maintained.

While the problem of updating the used data structures efficiently is addressed, there is no way to avoid updating entire paths in both the *VS*-tree and the *T-index* when a single triple changes, which makes maintenance of highly dynamic data rather expensive.

In its original form *gStore* is not a distributed system but an addition to *gStore* has been introduced by Peng et al. [12] with the distributed *Local Partial Match* technique, although it could also be employed in conjunction with other graph-based query engines. It is motivated by the problem of finding matches which cross compute node boundaries in an efficient way.

The idea is that, given a complete query graph, each node explores its local data set and maximizes the sub-graph matching the query. The partitions are disjoint, so nodes do not share any vertices, but crossing edges between the partitions are available at both ends. If an incomplete match is actually a fragment of a solution to the query, any edge where only one end has a match in the current partition must be a crossing edge. Based on this proposition, all compute nodes produce maximal local partial matches in parallel. Special NULL values which match anything are introduced for crossing edges.

Once the partial matches have been found, the assembly phase is started. Assembly can be performed either centrally or in a distributed manner, employing the *Bulk Synchronous Parallel* model. If the final assembly is performed centrally by a master node, an iterative pairwise join of local partial matches is performed. First, matches are partitioned into sets where each set only contains matches which cannot be joined with each other to reduce the amount of possible joins to be considered. Then, crossing edges need to be closed with matches from other sets until a complete match for the query is found, or the partial match can be refuted because there is no viable join partner left. Distributed assembly works similarly, but each compute node only evaluates joins for a subset of the matches and then sends the results to other nodes for which they are relevant, indicated by shared crossing edges. Distributed assembly still implies that queries are scheduled by a master node and executed by slaves, but introduces a higher level of peer to peer communication.

This approach shifts the bulk of the network communication away from graph exploration, as the boundaries between partitions are only crossed once the maximally available local information has been computed already. The authors benchmarked this method in comparison to previous implementations and found the greatest speed gain when used for complex queries. Their benchmark results show though, that the scalability of *Local Partial Match strongly depends on the cost of communication during the assembly phase, so highly selective queries perform better due to the reduced
amount of information to be transmitted.

3.2.4 DREAM
The “Distributed RDF Engine with Adaptive Query Planner and Minimal Communication” by Hammoud et al. [4] presents a paradigm shift compared to the other systems mentioned here. Instead of partitioning the RDF data set, it simply replicates it over distributed compute nodes and partitions incoming queries instead in an attempt to avoid communication overhead and data passing as well as scheduling issues. By the time of DREAM’s introduction, it was the first system to employ this approach of partitioning, motivated by the fact that the largest RDF data sets did not exceed 2.5 TB in size, which makes replication of the entire set feasible. The unique feature of DREAM is its query planner, which is run on a central master node, while the slave nodes are kept agnostic as to which storage system is deployed to them locally.

The query planner first generates a query graph and then partitions it into basic sub-graphs, which may be exclusive or shared depending on the query as a property of their join vertices, where one join vertex represents an exclusive sub-graph, and multiple of them a shared one. From the sets of join vertices a directed set graph is generated, where each node represents a set of join vertices, and considering the direction of their associated edges. On the directed set graph the most opportune query plan is determined using a cost function. The join vertex sets of the query plan are then sent to one slave node each, where the query is started in parallel. Once the candidates for a join vertex set are known on one machine, only their IDs need to be broadcast to the other machines who share them, as every one of them holds the full data graph. The cost function deciding the lowest cost plan takes the interdependencies into account and strives to minimize the idle time of slaves waiting for auxiliary data from other sub-graphs. The query planner can adapt the number of slave nodes deployed for any particular query to the involved workload, bounded by the number of join vertices. The DREAM approach is most suited for minimizing query response times and adaptive work-balancing, but does not put much emphasis on memory efficiency, and the join vertex approach is not a good fit for range queries.

4. CONCLUSION
All of the presented systems address different issues in the organization of RDF data. While it may seem like there is a trend towards graph based systems, that may be caused by them leaving more room for variation, because there is more development from the ground up. Future development will likely be directed by technological progress, e.g. work by Mittal et al. [9] indicates that the widespread availability of Storage Class Memory will have a strong impact on the architecture of database systems in general, and especially systems like triAD, which are already designed as in-memory stores, can definitely profit from such hardware advances. On the front of network communication cost, traditional RDBMS have already paid attention to faster interconnection technologies like InfiniBand [13], which require software changes to reap their full potential benefits, which might become more relevant for RDF systems as well.

There is also the sector of IoT applications, where different technological parameters require a stronger focus on memory efficiency as presented by Jiang et al. [6], where an architecture for IoT data storage based on Hadoop is introduced. Those memory constraints imply that implementations like DREAM are not the best match. IoT also puts different constraints on network data exchange, so optimizations like those in EAGRE are quite relevant for the sector. It appears that future research in the field of RDF data storage systems has to focus on adapting to hardware paradigms which seem to be yet unexplored for triple stores. Possibly, synergies with relational database systems can be used further to the advantage of RDF systems, as discussed in the context of S2RDF, since those tend to be on the forefront of technological advances.

5. REFERENCES


