Scalable Peer-to-Peer-based RDF Management

Christoph Böhm
Hasso Plattner Institute
Potsdam, Germany
christoph.boehm@
hpi.uni-potsdam.de

Daniel Hefenbrock*
Microsoft Corp.
Redmond, WA, USA
danielhe@microsoft.com

Felix Naumann
Hasso Plattner Institute
Potsdam, Germany
felix.naumann@
hpi.uni-potsdam.de

ABSTRACT

Handling web-scale RDF data requires sophisticated data management that scales easily and integrates seamlessly into existing analysis workflows. We present HDRS– a scalable storage infrastructure that enables online-analysis of very large RDF data sets. HDRS combines state-of-the-art data management techniques to organize triples in indexes that are sharded and stored in a peer-to-peer system. The store is open source and integrates well with Hadoop MapReduce or any other client application.

1. INTRODUCTION

We are witnessing an explosion in the amount of data to be processed and stored. To handle web-scale data sets new types of data management systems have been introduced; the key goals now are scalability across many servers and, due to frequent server failures, reliability. For instance, BigTable is a large scale storage system for structured data, in practice used as a replacement for relational systems [2]. To achieve scalability, typical relational properties, such as consistency, are relaxed, and SQL support is neglected. For analytical processing, MapReduce, a parallel data processing paradigm, has been introduced.

We argue that these trends for general-purpose structured data processing also apply to RDF: The Linked Open Data (LOD) initiative encourages the release and integration of RDF data into a large global data space. Triple stores, however, are currently limited in scaling out to truly web-scale data sets. As a result, solutions have emerged that employ MapReduce for large-scale RDF processing [6]. Though RDF data is highly structured, those solutions use low-level storage systems, such as the Hadoop Distributed File System HDFS to feed data into MapReduce, which requires handcrafted optimizations for each algorithm. This approach has several drawbacks: (1) Files in HDFS cannot be modified, only appended to. (2) There is no indexing, e.g., for filtering the input data, which results in additional MapReduce jobs to be chained for pre-processing. These shortcomings have been addressed by using HBase as storage system for RDF [4]. HBase runs on top of HDFS and is able to capture the structure of RDF while offering scalability. We show that our system, which is tailored to analytical RDF processing, outperforms HBase solutions.

We present a new scalable RDF storage infrastructure called Hadoop Distributed RDF Store (HDRS). HDRS allows seamless integration with the Hadoop MapReduce framework and can also be used in other application scenarios. HDRS offers a simple client interface to read and write triples, and thus hides its complexity as a distributed storage system. Its key benefits compared to existing solutions for RDF storage are:

- *Superior scalability* compared to non-distributed triple stores: HDRS scales vertically in storage capacity and read / write throughput as nodes are added.
- *Superior RDF data management capabilities* compared to HDFS-based solutions: The system is a triple store that indexes RDF data, and thus offers fast lookup and in-order scanning. It can be read and written in any order while HDRS is on-line.
- HDRS *is available* at `http://code.google.com/p/hdrs` and is easy to setup and configure.

Related work includes (but is not limited to) high performance triple storage systems, such as RDF-3X, Virtuoso, or Hexastore, usually work with indexed triple tables to reduce the cost of self-joins, which are common when executing SPARQL queries [3, 5, 7]. BigData is a distributed triple store that implements sorted indexes and dictionary encoding in a master-slave setup [1]. Indexes and the term dictionary are split into shards to be distributed. The latter, however, limits performance in cases where the term dictionary does not fit into main memory.

In the following, we outline the architecture of our system and then describe selected implementation details. Finally, we exemplary show performance measurements.

2. DESIGN AND IMPLEMENTATION

We now introduce the distributed architecture of HDRS. HDRS is a peer-to-peer system which is beneficial in terms of scalability and robustness. These benefits are mostly due to the absence of a master node, which could become a performance bottle neck as well as being a single point of failure. The peer-to-peer design implies all functionality to be implemented decentralized, presenting major challenges for

(*) Work done as a master student at HPI, Potsdam.

I-SEMANTICS 8th Int. Conf. on Semantic Systems, Sept. 5-7, 2012

consistency across the system.

Figure 1 shows the architecture of HDRS. The upper part of the figure shows the logical perspective. The bottom part depicts the physical architecture. Logically, an HDRS instance stores a multi-set of triples, organized into a number of ordered indexes. Each index is self-contained, i.e., it is not dictionary-encoded, and stores all triples in a specific collation order (i.e. the combination of a triple's values). There are six *possible* collation orders for subject-predicate-object triples: SPO, SOP, PSO, POS, OSP, and OPS, i.e., a user may choose desired collation orders to be stored depending on the the use-case at hand: For instance , when planning to use many S-O-joins, one should use SPO and OPS.

Physically, indexes are divided into segments in order to be stored in a distributed manner. Each segment is assigned a physical node on the network. Client applications can query any node in the system to find out which node can serve a particular triple or range of triples. Triples are then accessed directly at the responsible node.

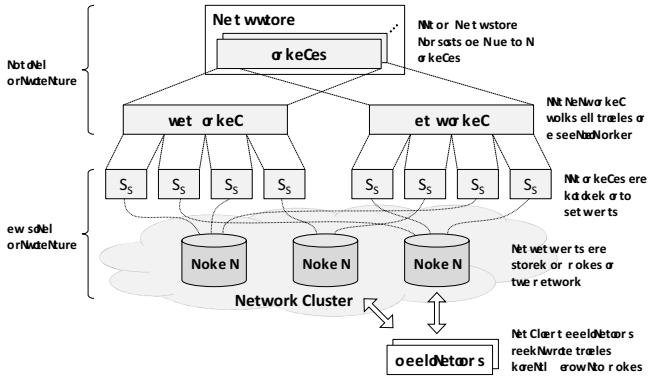

Figure 1: HDRS architecture overview.

Data Model and Operations. Instead of implementing set-semantics, HDRS stores a *multi-set* of triples from which client applications can add or remove triples. Thus, the store can keep track of triples that are added multiple times when, for instance, being used in a LOD web-crawl scenario where analysis takes place while writing into it simultaneously. Further, multi-set-semantics has several implementation advantages, e.g., it allows for an efficient index update approach. Internally, triples are persisted in up to six indexes. Each index enables fast lookups, in-order scanning, and pattern matching. For example, the pattern `(Berlin, * , *)` matches all triples having `Berlin` as subject. Since each index only persists one specific collation order, not every index can be used to match all patterns efficiently. Storing fewer indexes results in better write performance and less storage space, whereas more indexes offer higher flexibility for more efficient pattern matching. There are three operations supported on an index: 'write', 'delete', and 'read'; we model 'update' operations using 'delete' and 'write'. An **index** in HDRS is a pair (c, T), where $c \in \{SPO, SOP, PSO, POS, OSP, OPS\}$ is a collation order and T is a list of triples $(t_1, \ldots, t_n)$ that is sorted according to c. That is, $t_i \leq_c t_{i+1}$ for $i = 0, \ldots, n-1$.

Given an HDRS index $I = (c, T)$, the **multiplicity of a triple** t is the number of times t is present in T. As for the index operations mentioned above, the multiplicity is modified as follows: Writing a triple t increases the multiplicity of t by 1. Deleting a triple t decreases the multiplicity of t by 1. Each triple t is stored in all indexes of a HDRS store – possibly located on different nodes. Thus, keeping the multiplicity of t consistent across all nodes where t is stored is a challenge. Consider Fig. 2 as an example: There are two nodes N_1 and N_2; three writers w_1, w_2, and w_3; and one triple t. Here, N_1 stores the SPO index and N_2 stores the POS index; t is stored on both nodes with a multiplicity of 1. Now assume w_1 additionally writes t while, at the same time, w_2 and w_3 delete t. The overall outcome of this situation depends on the order in which the operations arrive at both nodes. Table 1 shows the resulting multiplicities for all operation orderings at both nodes. In four cases, the system would be in an inconsistent state. For robustness HDRS de-

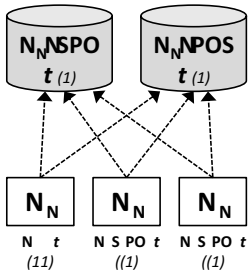

Figure 2: Two nodes and multiple writers.

Table 1: Combinations of write orders at nodes N_1 and N_2 and resulting multiplicities for triple t.

	N_2:W D D	N_2:D W D	N_2:D D W
N_1: W D D	0/0	0/0	**0/1**
N_1: D W D	0/0	0/0	**0/1**
N_1: D D W	**1/0**	**1/0**	1/1

liberately does not have a master node enforcing operation orders. Instead, it assumes that all operations are *commutative*, i.e., the outcome of a set of operations is always the same, regardless of their order. This strategy allows each node to define its own ordering while guaranteeing consistency across nodes. To achieve commutativity, HDRS allows the multiplicity of a triple to be *negative*. Then, deleting a triple with multiplicity 0 does have an effect: It becomes -1. Thus the outcome in the previous scenario is always 0/0.

A multiplicity of ≤ 0 for a triple t is interpreted as t not being present in the store. We are aware of the drawback that it can lead to unexpected behavior, say when writing a triple expected to have a multiplicity of 0 while actually having a negative multiplicity. Nonetheless, we deliberately trade off semantics and scalability to achieve consistency without costly global ordering coordination.

Distributed Index Storage. In the following we elaborate on how an index is stored in a distributed manner on distinct nodes in the system: Each index is split into disjoint parts called *segments*. Each segment covers a successive triple range and is assigned to one node. This strategy is known for its scalability; it is also referred to as sharding or shared-nothing. Given an index $I = (c, T)$, a **segment** $S = (I, T', R)$ is a tuple, where I is the corresponding index of S, and $T' \subseteq T$ is a list of triples $(t_n, \ldots, t_m)$ contained in S. R is the range $[t_n, t_{m+1})$ of S, with t_n being the first triple of S, and t_{m+1} being smallest triple (according to

c) of the next segment of I. T' is sorted according to the collation order c. The range of segments is important for determining the appropriate segment for a triple. HDRS uses hashing for assigning segments to nodes. Specifically, the first triple (subject, predicate and object) of a segment is hashed to determine the responsible node. This strategy does not allow adjusting the load-balance by offloading triples when the system is online. Achieving good balance relies on a hash function that distributes segments evenly across all nodes. To further maintain the load-balance, segments are split into two parts once they grow beyond a specified limit, by default 64MB. Figure 3 illustrates a segment split of a segment with four triples. S_1 stays at the same node since its first triple t_1 is also the first triple of the parent segment S. Child segment S_2, however, is likely to be assigned to another node as its first triple is t_3.

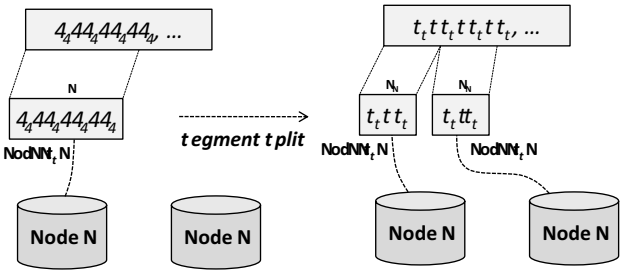

Figure 3: Segment split into segments S_1 and S_2.

Segment Implementation. Segments form the basic triple storage containers in HDRS. Their implementation is inspired by tablets in Bigtable [2]: They consist of a number of sorted *triple files* and a *triple buffer*. The latter collects segment modifications and flushes to disk, i.e., it creates a new triple file when reaching a predefined size. Index files are merged when a predefined number of files has been reached. To read a segment, the (sorted) files and the (sorted) buffer are combined. Files are divided into compressed blocks and a block index. The multiplicity of triples allows immutable triple files and sequential disk I/O. That is, for triple deletions, HDRS adds a triple with multiplicity -1 and simply sums up multiplicity values when merging segment contents.

When requesting a triple, HDRS needs to determine the segment in which it resides: Each node maintains an index segment map that is kept up-to-date using a weakly-consistent *anti-entropy* protocol. A client can thus be forwarded to the node where the segment with the requested triple resides. When writing triples, each triple must be written to *all* of its indexes. HDRS implements the *Two-Phase Commit Protocol* to achieve atomic writes across nodes. The client library provided with HDRS contains the transaction manager which buffers transactions and issues them in batches.

Hadoop Integration. HDRS can be run on a Hadoop MapReduce cluster, side-by-side with HDFS. When serving a MapReduce job, segments are treated as input splits that are read by the mappers in parallel. Since the range of each segment is known prior to reading and since triples are sorted, segments can be grouped into logical splits. Thus HDRS can guarantee that subjects are not scattered across multiple Hadoop input splits.

3. EVALUATION

For performance measurements we used a cluster of 10 old commodity machines and the Billion Triple Challenge 2010 data (BTC), which contains roughly 3.2 billion triples of web-crawled data[1].

To demonstrate **horizontal scalability**, we measured the write-throughput for clusters of $N = 1$, 2, 6, and 10 nodes by loading up to one billion triples without interruption. For each size $N \times 100$ million triples were loaded. We compared our throughput to HBase (v. 0.90.3, configured equally and fair) running on the same cluster, plus one HBase master and HDFS Namenode. In HBase, triples are stored in a single table with one column family. For each subject, there is a row containing one column qualifier per predicate. Objects are stored as values; multi-values are modeled using the versioning feature of HBase. Our HDRS was configured to store only the SPO index. We measured the read-throughput by running a simple MapReduce job that counts all triples in the store by scanning through them (Hadoop v. 0.20.2). We expect our throughput measurements to behave similarly for more then one index, since in both cases, HBase and HDRS, resources would have to be shared for maintaining the different index segment files on disk.

For HDRS, it takes $14-36$min to load $1-10$ nodes. A full scan of the content requires less then 5min in all cases. Figure 4 compares load- and scan-throughputs for both setups. Table 2 breaks down the load-throughput results for HDRS in particular. The HDRS batch-write-throughput increases from 25 MB/s (1 node) to 100 MB/s (10 nodes). HBase achieves 6 to 24 MB/s. For HDRS, this amounts to approximately 465,000 triples per second that can be loaded on our 10-node cluster. However, both systems reached a speedup of 4 in write-throughput for 10 nodes (taking 1 node as baseline). Scanning scales almost perfectly in HDRS, from 73 MB/s to 719 MB/s aggregated throughput, which is a speedup of 10. HBase achieves 173 MB/s with 10 nodes.

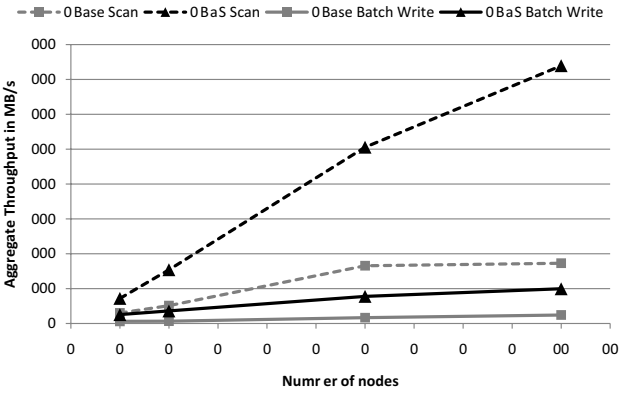

Figure 4: HDRS and HBase throughput results.

We further evaluated how **index segmentation** influences the performance, as the distribution of segments across multiple nodes in the cluster is our approach to achieve horizontal scalability. Given the number n of nodes in the cluster, the probability of a child segment transfer is $(n-1)/n$, because the hashing of the first triple may result in any of the nodes (out of which $n-1$ are not the one where the parent

[1] `http://challenge.semanticweb.org`

#nodes	triples	total time	throughput	triples/sec
1	100 M	14:03 min	25.43 MB/s	118,694
2	200 M	19:50 min	35.99 MB/s	167,997
6	600 M	27:30 min	77.82 MB/s	363,543
10	1000 M	35:53 min	99.61 MB/s	464,494

Table 2: HDRS throughput for batch-loading the BTC data set.

segment resides). We found that this estimate holds in practice, which implies a segment transfer for each segment split for large clusters. Figure 5 depicts the required transfer time for segment splits per respective segment sizes when loading three billion triples ($N \times 300$ million triples). The figure shows two clusters: The first contains transfers of segments smaller than 60MB, which can typically be moved over the network in less than two seconds. These small segments result from a scatter phase that we introduced to populate the nodes on the network with initial data when starting with an empty store. This scatter phase ensure a quick distribution of the load across the nodes in the cluster. Without the scatter phase (effectively a lower maximum segment size) the system would first load individual nodes up to the actual limit before distributing the data. The second cluster in comprises segments that are mostly around 260MB. These result from a maximum segment size of 512MB and have a transfer time varying around 15 seconds.

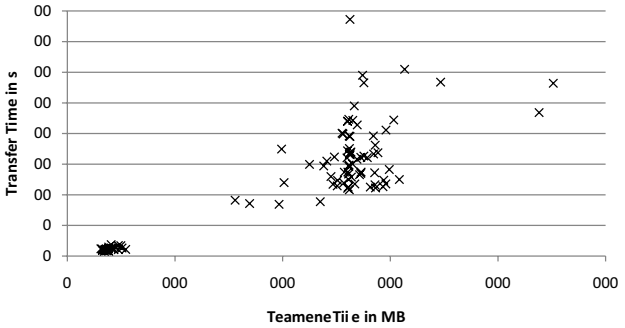

Figure 5: Segment size and transfer time of segment transfers for loading three billion triples into an HDRS store with 10 nodes.

Segment transfers lower the write-throughput, because transfers can block all write operations, as writes are performed in large batches of triples (performed by the transaction manager in the client library). A batch write can only succeed if all segments involved can be written – as required by the transactional design of HDRS.

To determine this overhead, we measured the time the client application was blocked during the batch-write in the previous experiment with three billion triples. Figure 6 depicts the time required for segment transfers as well as the time the system was blocked due to these transfers. The accumulated increase in the batch-load time (the topmost graph in the figure) is the total overhead required for distributing the segments across the nodes. In theory, if there was no overhead, this number would be 0 since the amount of data loaded is proportional to the number of nodes in the experiment. Apparently, roughly 50% of the total overhead is caused by the client application being stalled during segment transfers. The figure also shows that the client application is stalled longer than the actual segment transfer time. This further delay is because the routing mechanism first needs to learn about the new segment location before it can continue the batch. Of course, this overhead of ≈50min (out of ≈110min) for loading 3 billion triples into a cluster of ten nodes cannot be neglected. However, HDRS scales almost perfectly for reading triples which clearly supports our system design decisions.

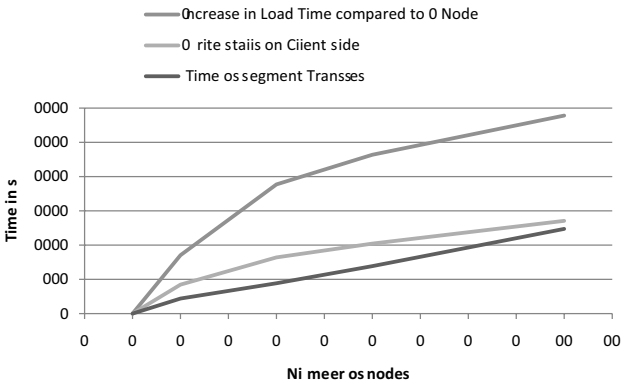

Figure 6: Overhead of segment transfers by stall and transfer time as well as accumulated load time increase (topmost graph).

4. CONCLUSION

We presented HDRS – a new scalable storage infrastructure for very large RDF data sets. The peer-to-peer architecture of HDRS stores a set of sorted indexes that are eventually consistent. Scalability is achieved through its decentralized design. We plan to extend HDRS with reliability features, such as replication, to minimize the probability of data loss. We also plan to add triple context support (quads). The community is welcome to contribute at `http://code.google.com/p/hdrs`.

5. REFERENCES

[1] Bigdata architecture whitepaper. Technical report, SYSTAP, `http://www.bigdata.com`, 2009.

[2] 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 *OSDI*, 2006.

[3] O. Erling and I. Mikhailov. RDF support in the Virtuoso DBMS. In *Conf. on Social Semantic Web*, 2007.

[4] C. Franke, S. Morin, A. Chebotko, J. Abraham, and P. Brazier. Distributed semantic web data management in HBase and MySQL cluster. *CoRR*, 2011.

[5] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J*, 19, 2010.

[6] J. Urbani, S. Kotoulas, E. Oren, and F. Harmelen. Scalable distributed reasoning using mapreduce. In *ISWC*, 2009.

[7] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *VLDB*, 1, 2008.