RDF Stores for Enhanced Living Environments: An Overview

Petteri Karvinen[1], Natalia Díaz-Rodríguez[2](✉), Stefan Grönroos[1], and Johan Lilius[1]

[1] Information Technologies Department, Åbo Akademi University, Turku, Finland
karvinen.petteri@gmail.com, research@stefang.net, jolilius@abo.fi
[2] ENSTA ParisTech and Inria Flowers, U2IS Department, Palaiseau, France
natalia.diaz@ensta-paristech.fr
http://flowers.inria.fr

Abstract. Handling large knowledge bases of information from different domains such as the World Wide Web is a complex problem addressed in the Resource Description Framework (RDF) by adding semantic meaning to the data itself. The amount of linked data has brought with it a number of specialized databases that are capable of storing and processing RDF data, called RDF stores. We explore the RDF store landscape with the aim of finding an RDF store that sufficiently meets the storage needs of an enhanced living environment, more concretely the requirements of a Smart Space platform aimed at running on a cluster set up of low-power hardware that can be run locally entirely at home with the purpose of logging data for a reactive assistive system involving, e.g., activity recognition or domotics. We present a literature analysis of RDF stores and identify promising candidates for implementation of consumer Smart Spaces. Based on the insights provided with our study, we conclude by suggesting different relevant aspects of RDF storage systems that need to be considered in Ambient Assisted Living environments and a comparison of available solutions.

Keywords: RDF store · RDF frameworks · Benchmark
Smart spaces · Ontologies · Ambient Assisted Living · Semantic Web
Publish/subscribe systems

1 Introduction

With the advent of the open Web and the large amounts of information that it has brought with it, a need for technologies that can handle large quantities of unstructured data in an automated fashion has arisen. Creating intelligent assumptions from the information pools that originate from widely different domains of knowledge is a labour intensive problem when using technologies popular today. A way of semantically representing data with the Resource Description Framework (RDF) and related semantic technologies has emerged

© The Author(s) 2019
I. Ganchev et al. (Eds.): Enhanced Living Environments, LNCS 11369, pp. 19–52, 2019.
https://doi.org/10.1007/978-3-030-10752-9_2

as a solution in order to mitigate some of the complexities involved when intelligently handling large amounts of knowledge. Storage and retrieval of information in the RDF format is most often performed by using specialised storage systems called RDF stores. The need for these storage systems capable of processing large amount of RDF data is evident by looking at the great effort that has been invested in a whole range of production system ready, RDF stores [24,32,42].

Smart Spaces are information sharing networks that are limited to the scale of rooms and buildings. Because of the cross-domain information sharing between actors, a Smart Space shares some of the same problems as the open Web when it comes to information processing. The information sharing between devices and users in the Smart Space, as well as a seamless device interoperability, and the need for reactive systems, are some of the motivation behind the use of RDF tools [58].

As more knowledge producers are introduced into a Smart Space environment efficient storage is needed in order to handle the growing amount of information constantly added to the Smart Space. The RDF store needs to provide fast data storage operations in order to enable the Smart Space to work smoothly. For Smart Spaces, the task of finding an RDF store is affected by the limited low-power hardware used in Smart Spaces environments. Therefore, a Smart Space needs an efficient RDF data storage that scales well, preferably in a distributed system.

Section 2.2 presents a brief overview of RDF frameworks, Sect. 2.3 presents some fundamental data storage techniques used in RDF stores, and this is followed by a run-trough of RDF store benchmarks suits in Sect. 2.4. Section 3 introduces the Smart-M3 platform with a definition of RDF data storage requirements for the system followed by a short analysis of the suitability of different RDF stores for the platform. In Sect. 4, the integration of a 4store storage option into the Smart-M3 platform and an evaluation of the implementation is outlined. Section 5 concludes the review and identifies future work.

2 Related Work: RDF Stores

RDF provides possibilities in knowledge processing that are not possible in other database models. The new way of thinking about information in these semantic technologies also presents their own challenges and new sets of tools. Even the most fundamental functionality of providing efficient storage and retrieval of information in an RDF data model is an issue that has created a new breed of information storage systems called the RDF stores. Besides providing storage and retrieval of information in the RDF format, RDF stores often consist of software solutions for a number of functionalities related to semantic technologies and information processing.

The RDF data model does not define the physical layout of the data itself, but instead it defines how the information should be presented to the user or the application when it is accessed from the RDF store. This abstraction of information has resulted in large differences in the underlying data structures used

for different RDF store. The data structures used for RDF stores range from off-the-shelf relational databases [15,41] to state-of-the-art advanced indexing schemes, which are specifically designed for the RDF data model [51]. As the underlying data structures greatly affect both the performance and the scalability of the storage system, this Section first presents the concepts that have shaped modern RDF stores. This presentation is then followed by a brief run-through of some of the most influential RDF stores. The Section concludes with a discussion of RDF store benchmarking software.

The data storage techniques in RDF stores range from mapping the RDF data model onto existing DBMS to custom DBMS where the data structures used are designed specifically for the RDF data model.

2.1 RDF Store Taxonomy

As the storage techniques have a deterministic effect on the performance of RDF stores, the identification of the core data structures used in RDF stores becomes important for evaluating individual RDF stores. One of the defining features for the real-world performance of RDF stores is how well they can handle the prevalent conjunctive information retrieval requests of the RDF graphs. As a result of this, the performance of RDF stores is tightly bound to how well the index structure can handle the joins that graph pattern matching in queries. In order to grasp the different data structures that are used in RDF stores, this section presents the major data structures and indexing schemes that are an integral part of RDF stores.

A number of papers have been presented on the topic of classifying different types of RDF Stores. The classification is usually based on analyzing the underlying storage methods that are used to implement the RDF data model. The most extensive study on the topic was presented by Faye et al. [34], who surveyed the RDF store landscape and presented a taxonomy of RDF storage techniques and grouped the RDF stores in a tree structure shown in Fig. 1. The main separation is into two groups: *non-native* RDF stores, which are based on existing data storage solutions; and *native* RDF stores, which use data structures designed with the RDF data model in mind. A conscious omission in Faye et al.'s study is that distributed and peer-to-peer RDF stores were not at all considered. A literature survey from SYSTAP [65] includes a moderately extensive discussion on some distributed RDF stores. In the survey, the distributed RDF stores are grouped into *index based systems*, *key-value stores* extended with MapReduce and *main memory systems*. Peer-2-peer RDF stores are discussed in length in [35].

Defining an exact taxonomy of RDF stores, as presented in Fig. 1, and classifying each RDF store can be considered somewhat misleading as RDF stores can incorporate a combination of storage techniques. Some RDF store vendors do not publicize the details for the underlying data structure, and this makes the task even harder. Nevertheless, it is important for the database system administrator to be aware of the different techniques used in the available RDF stores and how they affect both the performance and the scalability of the RDF stores.

Below follows short descriptions of the main techniques used in RDF stores as presented in Fig. 1.

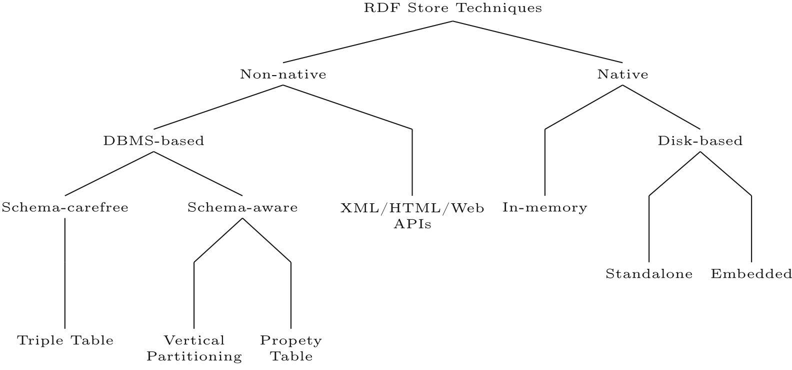

Fig. 1. RDF storage technique classification tree, as presented in [34]

Triple Table. The triple table can be considered the most straight forward way of storing RDF triples. In the triple table approach, the RDF data model is mapped directly onto a three-column wide table, in which each tuple contains the resources for the RDF statement subject, predicate and object. This can easily be implemented in any off-the-shelf RDBMS and it was a popular technique used in early RDF stores such as 3store [41], which maps the RDF graphs into a MySQL RDBMS triple table. A table representation on how the RDF data model could be implemented for a small example RDF graph in a triple table is presented in Table 1.

Table 1. Example of a triple table

Subject	Predicate	Object
place:City#London	rdf:type	place:City#
place:Region#England	rdf:type	place:England#
place:Country#UK	rdf:type	place:Country#
place:City#London	geo:isLocatedIn	place:Place#England
place:Place#England	geo:isPartOf	place:Country#UK
place:City#London	hasPopulation	8174000
place:Place#England	hasPopulation	53010000

A triple table representation as presented above, can be considered a rather naive solution that has some obvious disadvantages. This kind of single table representation will contain large amounts of unnecessary replication of information,

as the same resources will appear in several rows. The replication of information is also observable in Table 1 in which several of the subject and predicate fields are repeated. Additionally, this kind of naive triple table implementation will scale poorly since the number of triples in the table grows, as the query time will also grow linearly as the RDF graph grows. This is a limitation that makes the naive triple table infeasible for large datasets, a fact that was also noted in early RDF stores [41].

An improvement to the naive triple table approach is to build meaningful indices that covers the RDF statements. To cover all possible subject, predicate, object combinations, a total of six covering indices is needed. To provide an additional *context* resource for each triple in the RDF graph, the number of covering indices grows to 16. Most modern RDF stores that use a triple table also use some variation of covering indices [31,51].

Property Table. First introduced in the Jena framework in 2006 [66], the *property tables* is a step away from some of the scalability limits that persist in the triple table approach. The basic idea behind the property table is to discover clusters of triple subjects in the knowledge base that share the same properties and to group them into common tables. For each line in the property table one column contains the subject for the triple with one or more columns containing the property values for that subject. A property table grouping for the same example RDF graph as in Table 1 is illustrated in Table 2. As can be observed from Table 2, the triple predicates are not stored in the tables row data, but instead within the table meta data. The aim of this kind of structure is to take advantage of the regularities found in RDF graphs in order to reduce redundant writing of information, and in the process speed up some of the most commonly executed queries.

Table 2. Example of a multi-value property table RDF graph representation

(a) Property table

Subject	Type	geo:hasPopulation	geo:isLocatedIn
place:City#London	place:City#	8174000	place:Region#England
place:Region#England	place:Region#	53010000	NULL
place:Country#UK	place:Country#	63230000	NULL

(b) Left-over triples

Subject	Predicate	Object
place:Region#England	geo:isPartOf	place:Country#UK

One of the major advantages of the property table compared to a triple table is that the number of join operations is reduced for certain types of queries. For example, for queries that needs two or more single-value properties for a subject, all properties can be found on the same tuple row, eliminating the tuple joins that would have otherwise been needed if a triple table had been used. An

additional feature which is not possible in a triple table is the possibility to do *attribute typing*, i.e. defining the datatype formats for individual properties in the column schema.

There are numerous ways to group RDF graphs into different property tables. It has been shown that the selected property table scheme dramatically affect the performance of the RDF store [66]. If the property table groupings lead to wide and sparsely populated tables, the tables will be filled by a large amount of NULL values, which in turn can dominate the storage space [18]. The property table approach is also less flexible than the triple table approach as the clustered properties might need rearranging as the data changes to maintain good performance. Furthermore, the query performance is negatively affected when performing queries on RDF graphs for which the property triple match is unknown on a property table since all property tables then must be evaluated.

Vertical Partitioning. The third way to map the RDF data model onto an RDBMS solution is by using *vertical partitioning*. This approach was first introduced in SW-store [18] with the basic principle being that all triples are placed into n two-column tables, for which n is the total number of unique properties in the knowledge base. The first column is used to store the subjects of the triple that have the defined property for the table, and the second column contains the object values for those subjects. The tables are usually sorted by subject, allowing for *self joins* (combination of records in the same table) to be performed faster. A simple example representation of a property table is shown in Table 3.

Table 3. Example of a vertical partitioning (binary table) representation

(a) rdf:type

place:City#London	place:City#
place:Country#UK	place:Country#
place:Place#England	place:Region#

(b) geo:isLocatedIn

place:City#London	place:Place#England

(c) goe:isPartOf

place:Place#England	place:Country#UK

(d) geo:hasPopulation

place:City#London	8174000
place:Place#England	53010000

The use of a vertical partitioning data structure in datasets containing large numbers of predicates will lead to a large number of tables being created. The large number of tables is specifically problematic when executing queries for which several properties for each subject are requested. These types of queries requires self joins each subject that have several of the requested properties. Both

relational [18,31] and non-relational column stores have been used in implementations of vertically partitioned RDF stores.

Native RDF Stores. RDF stores that do not rely on existing RDBMS systems, but instead use a custom DBMS system, which use data structures that are tailored to the RDF data model, or store data in main memory, are called *native* RDF stores. Native RDF stores can be divided into *main memory* and *disk* RDF stores depending on the medium where the RDF graphs are stored. *Main memory* RDF stores such as Jena TDB and RYA store the entire RDF graph in the main memory, while disk-based RDF stores, like Virtuoso or 4store, use file systems in combination with custom DBMSs for storing the triple data in secondary or tertiary memories. The main memory approach relies on the fast access times of RAM memory to provide fast query response times, while modern disk-based systems make heavy use of cache techniques to serve frequently occurring queries.

A lot of effort has also gone into the creation RDF storage systems that use peer-to-peer technologies for the storage and retrieval of RDF data. An extensive study of RDF storage systems that take advantage of the peer-to-peer communication model is in [35]. In the study, Filali et al. identify the decentralization, the scalability, and the fault-tolerance provided by peer-to-peer systems as the leading factors that motivated the design of the RDF data stores that were covered.

RDF storage solutions capable of performing queries on large RDF datasets mapped onto distributed file systems and queried using a MapReduce engine have been presented in a number of research papers. SHARD [56], which was one of the earliest peer-to-peer RDF stores presented, uses Hadoop and the HDFS. The RDF graph is grouped into SHARDs that are directly mapped to the HDFS file system. Queries are evaluated by performing triple patterns match MapReduce operations on the SHARDs in a sequential order. Another Hadoop based RDF query system is CliqueSquare [36], which reduces the network traffic by exploiting the built-in replication in the HDFS and a clique-based algorithm to find connected subgraphs to speed up query processing.

RDF storage systems that store the RDF graph in a cluster of individual RDF stores and are queried using a MapReduce system have also been presented. An example of such a system is presented [56], which distributes the RDF graph into overlapping sub-graphs and placing the sub-graphs into individual RDF stores. All SPARQL queries in the system are processed into smaller MapReduce sub-query chunks in a master node that can be processed in parallel by the individual RDF store nodes. This approach can be considered better suited for querying large RDF that do not change since the graph partitioning performed in the master node limits the scalability of the system.

An inherent advantage of peer-to-peer RDF stores is that they offer a direct way to distribute the RDF graphs over hardware nodes. This enables vertical scalability by adding new nodes to the network, and therefore there is no need

to redesign of the system in order to achieve scale-out features. Even though the peer-to-peer RDF storage solutions have some advantages, problems relating the overhead caused by the traffic between the nodes, and in that they do not solve the underlying problems caused by splitting up the RDF graphs have been cited as problems are yet to be solved in current peer-to-peer RDF stores [65].

2.2 RDF Frameworks

In addition to individual RDF stores such as BitMat [21], MonetDB [63], TriAD [38], AdPart [40], H2RDF [55], there are a number of RDF-centric frameworks that provide interfaces to third party RDF storage implementations or implement their own internal RDF storage solutions. A listing of the major RDF frameworks with an accompanying description is presented below.

Apache Jena [2] RDF framework originated from the Hewlett Packard Labs and includes a whole range of RDF specific tools from parsers, RDF stores, reasoners and query systems. The libraries support both internal RDF stores and provide libraries to access a number of independent RDF stores. The libraries are Java-based, but bindings for the most common languages are provided.

Developed by Aduna, the OpenRDF Sesame framework [6] is similar to the Apache Jena framework in that it provides a de-facto standard tool set for processing RDF data in Java APIs. Access to most modern RDF stores is provided through the SAIL API part of the framework. The framework has been developed for over ten years and is used by companies in several different industries. The OWLIM platform [24] is a high-performance Java-based semantic repository that is packaged as an implementation of Sesame's SAIL API. Besides serving as an interface to the OWLIM RDF store, the platform also supports reasoning for RDFS, OWL Horst, OWL 2 QL and OWL 2 RL semantics.

Developed by Dave Beckett, the Redland RDF libraries [12], written in C, provide tools for parsing, querying and storage on RDF data. The Redland *storage* library supports a limited number of RDF stores, the default being a custom storage solution based on the Oracles Berkeley DB database. Besides the C API, the libraries have bindings to Ruby, PHP, Python and Perl.

The PerlRDF libraries [11] are a set of Perl libraries similar to the Redland RDF libraries with the aim of providing a Perl interface for RDF tools, in a similar fashion to what OWL API implementaion in Java does, to provide an OWL 2 syntax API.

2.3 Individual RDF Stores

Since the first RDF stores appeared in the early 2000s, a number of surveys and evaluations have been presented that both evaluate the state of RDF stores and discuss the techniques used. An incomplete list of recent studies include: an evaluation of RDF database solutions from 2009 [64], a report over RDF stores done for the European project 2011 [44], a discussion of interesting RDF stores in a literature survey of RDF storage approaches [34]. The RDF stores covered in the studies vary largely based on the aim of the studies. RDF stores briefly

covered in this chapter were included on the basis of having either shaped the evolution of RDF stores or being considered a major player in the current state of RDF stores. Below follows a short introduction to these RDF stores.

YARS2. Released in 2004, Yet Another RDF Store (YARS) was one of the first distributed RDF store released to the public. The improved version, YARS2 [43] released in 2006, improved the scalability of the system. YARS2 represents RDF statements as quads, in which the fourth position of the statement is defined as the source of the triple, functioning in a similar fashion as the RDF NAMED GRAPH [4]. The store uses six, alternatively ordered, covering indices: SPOC, POCS, OCSP, CPSO and OSPC. In these indices, S, P and O are the triple *subject*, *predicate* and *object*, and C stands for the *context* of the statement. YARS2 uses an *in memory sparse index* data structure that refers to sorted and blocked data files on disk. In order to save space, only the first two elements of each quad are stored in the sparse index. By doing so, the indexing structure sacrifices insertion speed for better query performance, as all six indices must be calculated in a specific order when inserting new triples. Huffman encoding is used in the data blocks in order to save storage space. The entire lexical value of the triple is indexed in order to speed up the queries that contain FILTER operations.

Virtuoso. Virtuoso is defined as "a general purpose relational/federated database system and application platform" [32] developed by OpenLink Software, and can be considered a full-featured RDF solution with interfaces for the Jena framework, the Sesame and Redland libraries, a limited OWL inference engine, full-text search, relational data analytics and Multi Version Concurrency Control (MVCC) for transaction handling. Virtuoso was originally a relational database that was later extended in order to support RDF data. The software first used a row-wise transaction scheme [33], but the latest version of the software, Virtuoso 7, uses column-wise compressed storage with a vectored execution [31]. The software is provided under both an open-source license for single machines and a commercial license for software that supports federated (distributed) storage and other additional functionality.

Virtuoso uses a quadruplet structure for modelling RDF triples and by doing so extends the subject S, predicate P and object O with a G graph node column representing the graph IRI ID. The earlier versions of Virtuoso used only two covering indices, $<GSPO>$ and $<OGPS>$, for each statement. The index structure was motivated by the assumption that most triples are queried using either the subject or the object. Virtuoso 6 and 7 extended the covering indices to include the optional covering indices $<PSOG>$ and $<POGS>$ as well as the additional indices $<OP>$, $<SP>$ and $<GS>$ for distinct projections. All SPARQL queries in Virtuoso are transformed into SQL statements that are then handled inside Virtuoso's SQL query engine in a similar fashion as Oracle's RDF_Match table function (see Sect. 2.3). The latest version of the software is marketed as scaling up to datasets over a trillion triples.

4store. 4store [42] is an open-source RDF store that was originally developed by Garlik in order to be used in the company's personal data protection products. As Garlik moved on to their new clustered RDF-store, 5store, 4store has been maintained by the 4store user community.

Even if 4store is labelled as the logical successor of 3store [41], it shares very little code with its predecessor. The main feature that has remained is the mapping of RDF resources as integers. The data structure used in 4store resembles the property table used in Jena SDB rather than the triple table used in 3store. RDF statements are defined as quadruplets or *quads* consisting of a *subject*, a *predicate*, an *object* and a *model* that is used analogously with the RDF NAMED GRAPH. The indexing and distribution of the RDF graph to nodes in 4store is based on hashing algorithms.

For the query optimization, 4store executes *bind* operations in a descending order based on a selectivity factor evaluated on the basis of statistical predicate frequency tables. The resulting bindings are combined in the master node, and as such produce the final query results. The evaluation of FILTER operations is delayed towards the end of the query execution in order to limit the cost of transforming the lexical values of RDF resources. The indexing structure gives good performance for most queries, but queries with unknown predicates on a knowledge base with many unique predicates will be at a disadvantage due to the large number of tables required by the property table-like structure used in 4store.

4store does possess some clear deficiencies compared to other leading RDF stores. The major deficiencies are an incomplete SPARQL 1.1 support, and a lack of both transaction handling and built-in inference engines. However, a separate version of 4store that supports backward inferencing on a *minimal RDFS* set [60] has been developed, but is yet to be included in the official 4store release.

SYSTAP, BigData. BigData is a RDF platform targeting the Semantic Web and it has been developed by Systap LLC since 2006. BigData was initially released in a single node version that is currently named *journal*. BigData has later been extended to a clustered version of the software called *federation*. The *journal* version is in principal a main memory RDF store, while the federation version uses a "dynamic horizontal partitioning architecture" that is inspired by BigTable [27]. The *journal* version is aimed for smaller knowledge bases that can fit into the main memory of a single node, whereas the federation is aimed at handling large knowledge bases that do not fit onto a single node. Like Virtuoso, the BigData software is published under both an open-source and a commercial license [13]. At the time of writing, the open-sourced version of the software can be used without a commercial license for knowledge bases less than 50 million triples. Transaction support using an MVCC system is available for both the *journal* and the *federation* versions of the software.

Jena TDB. The original Jena RDF store [66] (now called Jena SDB) was developed by the Hewlett Packard labs. The software system was initially an RDBMS

mapped onto a property table indexing scheme. The current RDF storage provided by the Jena framework, the Jena TDB, has diverged from the original version of Jena and can now be considered as a single node main memory RDF store. The change is motivated by the significantly better performance offered by keeping the RDF graph in the main memory compared to the initial disk based system, for which further development has been discontinued. Another of the selling points for the Jena TDB RDF storage solution is the extensive tools provided with the Jena framework.

Allegrograph. Developed by Franz Inc, Allegrograph [1] is a commercial grade graph database for RDF data, containing a range of RDF tools. Allegrograph represents triples in *assertions* and each triple is mapped into a *subject*, a *predicate*, an *object*, a *graph* and a *triple-id* assertion. The triple-id is mainly used for graph extension when performing direct graph reification for RDF graphs. The system uses a combination of dictionaries, seven different indices and a cache handling system in order to provide the storage and retrieval of the RDF data.

Knowledge bases in Allegrograph can be queried using both the SPARQL language and a specialized Prolog instruction set. The system supports full RDFS and partial OWL reasoning through its RacerPro [39] software, which is built on *tableau* calculus. Rather unique features found in Allegrograph are the possibilities for doing *geospatial inferencing* as well as *temporal reasoning*. Federation and ACID compliant transactions are also supported[1]. The system can be accessed by the number of programming languages or through the Jena platform.

OWLIM. OWLIM [24] is a family of semantic repositories that provides storage, inference and novel data-access features for RDF data. The software comes in three different versions: a main memory RDF store for datasets up to 100 million statements called OWLIM-Lite (previously SwiftOWLIM), a file system based RDF store for larger data volumes called OWLIM-SE (previously BigOWLIM) and a replication cluster RDF store called OWLIM-Enterprise.

All the OWLIM versions are accessible through the package interface layer in the Sesame SAIL platform. The query engine for OWLIM-Lite relies on the Sesame framework, while the other versions use their own built-in query engines. In addition to the SPARQL 1.1 language support, OWLIM-SE and OWLIM-Enterprise also support *full text search* through the Lucene [3] text search engine. OWLIM uses an embedded reasoning engine developed at Ontotext, which performs reasoning based on forward-chaining of the entailment rules over the RDF triple patterns with variables. A relatively unique feature for OWLIM-SE is the possibility for the user to receive notifications on changes in triples by using a publish/subscribe mechanism.

[1] The ACID properties of a DBMS that allow safe sharing of data are Atomicity, Consistency, Isolation, and Durability.

Oracle Spatial and Graph 10g-12c, and Oracle NoSQL. RDF and semantic inference support were initially introduced to the Oracle RD-BMS in 2005 [50]. Version 11g of the Oracle RDBMS that was released in 2007 provided native RDF storage that scales up to billions of triples. Also included in the release of version 11g was OWL inference and the integration to prominent RDF technologies such as Jena, Sesame and Protégé [22]. The RDF graphs in Oracle 11g are modelled using relational tables and views that are optimized for semantic data. RDF graphs can be accessed in the system by using mixed SQL and SPARQL queries. Nevertheless, all SPARQL queries are translated in runtime into table/join structures that are executed by the underlying DBMS[2].

A separate software product provided by the Oracle Corporation that has RDF support is the Oracle NoSQL database [9]. The underlying storage is based on the key-value store, Oracle Java Berkeley DB (previously SleepyCat DB). Oracle NoSQL supports SPARQL queries as well as inferencing, and it can be accessed through the Jena interface.

RDF-3X. Introduced in 2010 by the Plank Institute, RDF-3X [51] is an academic effort intended to improve the RDF storage architecture. The RDF-3X RDF store uses a RISC-style [7] architecture with a streamlined indexing structure combined with a streamlined query optimisation approach.

In RDF-3X, all triples are stored in a single triple table, and each triple is sorted lexicographically into one compressed B+ tree. In order to compress the storage of triples and to simplify the processing of queries, triple literals are replaced with identifiers using a mapping dictionary. When querying the knowledge base, the triple patterns are translated into string identifiers and the resulting literals get translated back to strings using a direct mapping index.

All six possible permutations of the covering indices are built for each triple and inserted into clustered B+ trees. This index structure ensures that single index lookups are possible for every triple pattern. For each tuple in the B+ tree leaf nodes value, byte-level compression is performed based on the delta difference of the preceding tuples. Similarities between neighbouring tuples in the B+ tree are exploited in order to gain a high level of compression. Additional aggregate indices (*SP*, *PS*, *SO*, *OS*, *PO*, *OP*, *S*, *P*, *O*) are also built in order to speed up the SPARQL queries that include partial triple patterns.

SPARQL queries are transformed and performed using tuple calculus. This is motivated by the fact that it eliminates a large part of the merge joins that are prevalent in property table approaches. The query optimization within RDF-3X is based on identifying the lowest-cost execution plan based either on the selectivity of the calculations for executing frequent join paths or alternatively on a through and specialized histogram of data when the join path data is not available.

The first version of the RDF-3X software was mainly optimized for retrieving information from RDF graphs. Later versions of the software include a compact

[2] http://download.oracle.com/otndocs/tech/semantic_web/pdf/oradb_semantic_overview.pdf.

differential indices and an integrated versioning, which enables the deferral of changes to the RDF graph that can be merged with the main RDF graph in batch operations. The additions enabled online updates to the knowledge base and provided time travel queries that offer both flexibility and consistency through a transaction concurrency control system [52]. The index structure in combination with the query optimisations used give RDF-3X a good performance in many types of queries, although it has been noted that the performance of RDF-3X degrades for unbound queries and queries where the selectivity factor is low [52].

Trinity.RDF. Trinity.RDF [67] is a main-memory RDF graph database based on the Trinity [62] distributed graph system. The system was designed to handle large Web scale data. Trinity.RDF introduces graph database specific features that are not available in other RDF stores like random walks and reachability that can be used for data analytics and data mining purposes.

The defining feature for Trinity.RDF is that queries are performed using graph exploration instead of relational joins common in many other RDF stores. This graph exploration is claimed to provide especially good performance compared to current solutions for graph walking queries, which, at the time of writing, shows superior performance compared to state-of-the-art systems in a number queries.

2.4 RDF Store Benchmarks

Since the introduction of RDF stores, a number of benchmark suites ([19,25,37], Waterloo SPARQL Diversity Test Suite (WatDiv) [20], Bio2RDF [26], Yago2 and 3 [46]) have been presented in order to measure RDF stores. The methods used in the benchmark differ somewhat from each other, but most of the benchmarks include at least the measurements *load time* for inserting datasets to the RDF store and the measurements on *query performance* using either synthetic or real world datasets. The accuracy of how well the benchmark measures real world performance has been questioned [30], and therefore, the benchmarks are included in this chapter as they provide a general impression of both the scalability and performance of RDF stores.

One of the earliest RDF benchmarks was the Leigh University Benchmark (LUBM) [37] released in 2005. The benchmark aimed to evaluate the reasoning capabilities and the query performance of RDF storage solutions by using OWL knowledge bases. The datasets used in the benchmarks are generated using an ontology dataset generator, which replicates university setting with triple data relating to professors, students and courses. The test suite provides 14 evaluation queries that can be used in order to evaluate the semantic inference and the reasoning capabilities of RDF stores while at the same time providing execution times for the aforementioned queries. The LUBM group does not provide updated experimental results for RDF stores, but the benchmarking suite has been used by several RDF store developers to compare the performance of their RDF stores against other RDF stores.

Another benchmark suite using synthetic data is the Berlin SPARQL Benchmark (BSBM) [25] that was first presented in 2009. Like LUBM, BSBM measures RDF store query speed using a number of SPARQL queries, but the evaluation is focused on explore and update scenarios in a business intelligence use case. The test suite includes multi-client benchmarks that are performed through a HTTP SPARQL front end in the RDF store.

The SPARQL Performance Benchmark (SP^2Bench) [61] introduced in 2009 uses SPARQL construct operator constellations and broader data access patterns in order to evaluate RDF stores in non-application specific use cases. The SP^2Bench suit uses artificially generated datasets related to publications, with a benchmark end goal to cover a large range of use cases. The SP^2Bench suite uses a total of 17 SPARQL queries in order to benchmark RDF store performance.

DBpedia SPARQL Benchmark [49] is a project that aims to provide a generic SPARQL benchmark creation methodology by using real world datasets. In 2011 [49], Morandi et al. present methods for how they extracted sample data subsets from the DBpedia dataset, and how they from the resulting dataset derive 25 unique SPARQL queries that can be used in order to benchmark the performance of RDF stores.

A table of the different RDF benchmark suites experiments is presented below. From the table, one can note that there are several orders of magnitude differences between the dataset sizes in the different benchmark experiments. The BSBM testing suite experiments are the largest in scale and can therefore be considered the most extensive. In addition to the experiment done by the benchmark creators mentioned above, both individual RDF store developers and independent sources have performed experiments using the different benchmarking suites. As the results of these other experiments are hard to compare in the scope of this chapter, they are not included (Table 4).

Table 4. Comparison of different RDF benchmarks, modified from [49].

	LUBM	SP^2Bench	BSBM v3.0	BSBM v3.1	DBPSD
RDF stores tested	DLDB-OWL, Sesame OWL-JessKB	ARQ, Redland, SDB, Sesame, Virtuoso	Virtuoso, 4store, Jena-TDB, Jena-SDB	BigData, BigOWLIM, Jena-TDB, Virtuoso 6 & 7	Virtuoso, Jena-TDB, BigOWLIM, Sesame
Test data	Syntetic	Syntetic	Syntetic	Syntetic	Real
Dataset size (millions of triples)	0.1–6.9	0.01–1	100, 200	10–150000	14–300
Use case	Universities	DBLP	E-commerce	E-commerce	DBpedia
Classes	43	8	8	8	239 + 300K
Properties	32	22	51	51	1200

In the BSBM v3.0 experiment run in 2010, Virtuoso 6 and 4store were shown to have the best performance of the tested RDF stores with a nearly equal

performance in most queries performed. Virtuoso 7 showed the best results in the BSBM v3.1 benchmarks experiment, with Virtuoso 7 showing an order of magnitude better performance compared to the other RDF stores regarding both scalability and query execution time.

One obvious conclusion that can be drawn from the benchmark experiments is that there is a large difference in query execution times and scalability between RDF stores. The performance improvement shown in the Virtuoso 7 compared to other RDF stores and previous versions of Virtuoso can be interpreted as a sign that there is yet much optimization to be done for RDF stores in the future.

3 RDF Stores in the Context of Smart Spaces

The concept of smart spaces was introduced to enable an intelligent interaction of information between entities in both the physical and the virtual environment of an enclosed space. The vision was to build smart spaces that can be seen as a small version of the broader "Internet of things" concept. Considering that a smart space can contain a plethora of different actors producing information in varying domains, the choice of using semantic technologies has become a logical for the implementation of the smart space concept.

Even if the use of semantic technologies can be considered a rather novel feature for the Smart-M3 platform we will use, it also raises a need for efficient RDF data storage and retrieval in order to enable the information sharing inside the smart space environment. The insufficiency of the currently available RDF stores in the Smart-M3 is one of the motivation behind the task of the work done in this chapter; to improve on the storage solution currently used in the Smart-M3 software. The end goal being that the platform can become a viable alternative for use in real world applications. To evaluate the suitability of RDF stores for the smart spaces, we start with an introduction to the Smart-M3 platform followed by the defining of the storage requirement for smart spaces. Lastly, an analysis of how well different RDF stores suit the defined requirements is given.

3.1 RDF Storage in Smart-M3

Smart-M3[3] is an implementation of the smart space concept that originated from a collaboration between the Nokia Corporation and the VTT technical Research Centre of Finland starting in 2006.

The motivation behind the Smart-M3 is to create a **M**ulti-device, **M**ulti-domain and **M**ulti-vendor platform for information sharing between devices and people in smart spaces. Even if the concepts of intelligent rooms or buildings is not in itself a novel concept, most implementation of intelligent spaces on the market are bound to vendor specific devices or are limited to specific types of devices. The idea behind the Smart-M3 platform is to let any device or user

[3] https://github.com/smart-m3, https://sourceforge.net/projects/smart-m3.

belonging to the smart space, regardless of the vendor of the device, to join the smart space and to add to the common information pool. The devices and people, or *knowledge processor* (KP) as they are called in Smart-M3, can share information through a central service in the smart space called a *semantic information broker* (SIB). A depiction of the logical layout of a Smart-M3 environment is presented in Fig. 2 below.

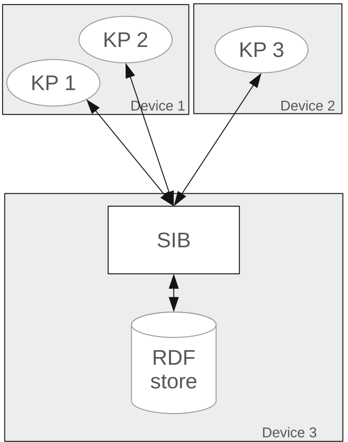

Fig. 2. Smart-M3 overview

The SIBs act as mediators of communication between KPs through the rules and syntax specified in the Smart Space Access Control (SSAP) [8] protocol. The SSAP protocol defines the following operations that a KP can perform on a SIB: *join* the SIB, *leave* the SIB, *subscribe* to changes to certain pieces of information, *unsubscribe* from an active subscription, *add* triple/triples to the SIB, *remove* triple/triples, *update* a triple and *query* the SIB. The most novel of the operations mentioned above are subscribe and unsubscribe[4], which provide the smart space with a publish/subscribe paradigm. This paradigm works through users defining persistent triple matching subscription queries that are triggered whenever a change has occurred in the corresponding triples for the query in the knowledge base. When a subscription is triggered, the SIB notifies the KP that produced the subscription about the changes that has happened related to the subscription since the last notification. In a sense, one can look at it as the SIB notifying the KP that an event has happened in the smart space.

The early use cases for the Smart-M3 were related to intelligent homes [29, 58, 59], in which interaction between the devices and the users in the home were made easier by automating events and centralizing how functions of devices were accessible. Since then, other use cases ranging from home entertainment

[4] The OWLIM-SE software suite offers a somewhat similar notification system.

systems [57] to person health monitoring security [45] or bioimaging [28] purposes have been proposed.

As was briefly mentioned earlier, one of the major factors affecting the performance of the Smart-M3 platform is the underlying RDF store that is used in the SIB. Most of the activity in smart space environments involves either adding pieces of information to the knowledge base or accessing the information in the knowledge base. When considering how prevalent these operations are in combination with the large difference in performance of the different RDF stores that were discussed in Sect. 2.3, the choice of storage solution for the Smart-M3 does have impact the performance of the system as a whole.

The earliest versions of the Smart-M3 platform used a RDF store that relied on an embedded MySQL database that was accessed using a specialized domain modeling language. The most recent version of the Smart-M3 SIB, RedSIB, uses the libraries found in the Redland framework [12] for all its data storage needs. Consequently, the Redland storage library gives Smart-M3 access to storage modules that use embedded RDF stores, in-memory RDF stores or native RDF stores. All the RDF storage modules are listed in Table 5 alongside some of the functional and non-functional features for each module.

The default storage module in Redland is the embedded Berkeley DB (BDB) with an enabled hash indexing option. This module is also the default storage option used in the RedSIB software. In the BDB storage module, the triples are mapped to the BDB key-value store with the help of three indices: *SP2O*, *SO2P* and *PO2S*, for which the (S, P, O) resources to the left of the *2* build the *key* and the resource on the right side represents the *value*. The BDB storage solution performs well when small RDF graphs are used, but the scalability of this storage solution is limited by the indexing scheme used. The indexing scheme leads to vast amounts of storage space needed to store large RDF graphs, rendering the module unusable in Smart-M3 environments that contain large knowledge bases. Of the other modules listed in Table 5, Virtuoso is the only full-featured RDF store capable of handling datasets over tens of millions of triples. The Virtuoso storage module includes the option to use an internal query engine for handling SPARQL queries.

3.2 Problems Related to the Existing RDF Stores in RedSIB

In the latest version of RedSIB, the publish/subscribe functionality was implemented with the help of two separate RDF store instances that keep track of triples that have been added and removed. As triples are added to the storage instances, they get matched with the active subscriptions in the SIB in order to evaluate if the subscription should trigger. Caching features are used to limit the overhead that the publish/subscribe functionality causes. Nevertheless, the insertion and the removal of triples become linearly slower with every subscription that is added to the RedSIB [48].

One feature missing in all of the alternatives for the Redland libraries is the lack of scale-out capabilities. As one of the aims has been to create a scalable

Table 5. A feature run down on RDF storage modules provided by the Redland storage library

Module	Storage type	Persistent storage	Scalability	Transaction support	Named graphs	Additional notes
Berkeley DB	Key-value store	Yes	Tested up to 10MT	Yes	Yes	Large disk usage when the indexed option is used
MySQL 3 an 4	RDBMS	Yes	Larger data models	Yes	No	-
PostgreSQL	RDBMS	Yes	Larger data models	Yes	No	Indexed but not optimized
SQLite	RDBMS	Unknown	Unknown	No	Unknown	-
Virtuoso 6	Row-wise RDBMS	Yes	1B+ triples	No	Yes	-
Memory	In-memory	No	Poor	No	Optional	Fast with small models
File	In-memory/file storage	Yes	Poor	No	No	Uses the memory module on a file
URI	In-memory	No	Poor	No	No	Uses the memory module on a file
3store	Triple table	Yes	A few million triples	No	No	Alpha quality support

solution that can be extended when the need for more storage space and processing power is needed, this was concluded to be a desirable function for the RDF store used in RedSIB.

3.3 Previous RDF Store Evaluations for Smart Spaces

The use of RDF stores for smart spaces was explored during the DIEM project in 2011 [53]. In the study, Allegrograph, OWLIM-SE, Virtuoso, 4store and Bigdata were chosen for the evaluation based on the fact that they were identified as capable of handling up to 10 billion triples knowledge bases. The study includes a feature run-through for each of the possible stores with some commentary on the suitability of each store as a part of the smart space environment. At the time of the evaluation, none of the RDF stores provided full SPARQL 1.1 coverage as it was still in draft status. In the evaluation, no conclusive recommendation of what RDF store would fit the Smart-M3 best was made. The major observations

made in this direction were that OWLIM-SE was identified as having favourable usability aspects and Virtuoso was noted to show a good query performance. What can be considered an omission in the DIEM study is that the hardware constraints of typical Smart-M3 platforms was not properly taken into account. Even though it is often favourable to run the SIB using low-power commodity hardware that are running all the time, the study only considered RDF store benchmark experiments run on server grade hardware. Furthermore, a great deal of progress has happened in the field of RDF stores since the study was made, warranting a new evaluation of the possible RDF stores.

3.4 Defining Requirements in Smart Spaces

The major factor that has affected the outlining of the requirement definition for the storage solution for the Smart-M3 platform is how well it fits into the vision of a scalable in-house smart space system. The envisioned system implies that the RDF store used in RedSIB should be able to store a large amount of RDF triples, while at the same time it should continue to serve the information sharing needs of the smart space environment. With these assumptions in mind, a conscious decision was made to aid the decision process of choosing an RDF store in a cluster structure that consists of several low-energy hardware nodes. It was therefore concluded that a preference should be made for selecting RDF stores for which the triples in the RDF store could be distributed between the low-energy nodes.

For the evaluation, the feasibility of a centralized clustered in-home SIB box, a hardware prototype (hereafter referenced as prototype) that consists of two ODROID U2 [5] development boards. The boards are based on the Samsung Exynos 5 32-bit ARM architecture chipsets with each board equipped with 2 GB of DDR2 SDRAM and an 8 GB SDCARD memory modules. To be considered in the evaluation, the proposed RDF stores should be able to run on the prototype hardware.

Defining requirements for the RDF stores for the Smart-M3 in this evaluation relies on rough estimates, as no Smart-M3 environment has been created that could give an accurate representation of the storage needs of a large scale smart space environment. As discussed in Sect. 2.3, the properties of the RDF dataset that are used and the type of queries that are normally performed in a smart space setting will affect the performance of the RDF-store. Additionally, the size of the datasets that the underlying RDF store can handle will limit the scalability of the system. To know how the above mentioned factors affect the functionality of the Smart-M3 system, it would be preferable to know in advance what kind of data is to be used in the Smart-M3[5].

The defining factor of many of the use case scenarios is that the system will need to handle frequent small inserts and deletions of triples. For updates of

[5] Unfortunately, for the evaluation in this chapter, no figures for either the performance or the scalability of the future needs of the envisioned large scale smart space environment were available at the outline and the only option was to use estimates.

the knowledge base, this works well in RDF stores that do not need to perform expensive index updates every time the knowledge base has been updated. The frequent updates were identified as a possible concern for the index-based RDF solution for which batched triple inserts and removes are preferred.

For evaluation purposes, it was decided that the RDF stores should at a bare minimum be able to handle an arbitrarily chosen number of ten millions of triples, based on what can be considered to be a reasonable number of triples that a smart space should be able to serve. The criteria for the RDF stores in this evaluation are that they should be able to keep loading this number of triples, while at the same time they should be able to perform simple queries within the millisecond range.

At the outlook the evaluation, there were no hard criteria on how fast queries should be handled in a smart space. As is observable from RDF store benchmarks, it is not unreasonable to expect a modern RDF store to be able to execute simple SPARQL queries in milliseconds. This order of magnitude of query execution times should reasonably be assumed not to incur noticeable delays in the RedSIB software. More complex queries require more execution time and this can potentially slow down the Smart-M3 system due to only one query being processed at a time, effectively leaving the whole system waiting for the query to finish before a new query is performed.

RDF store transaction support was not considered to be an obligatory feature as the Smart-M3 software does not, at the time of writing, have support for transactions. Nevertheless, it is not unreasonable to expect that transactions will become part of future releases of the RedSIB software in order to support transactions, as it is a proven method for handling the reliability and security aspects of sensitive information. For this reason, transaction database operation support was considered as a desirable feature for future use. However, the introduction of transactions in the RedSIB software is out of the scope for the work performed in this chapter.

Even though data persistence and data integrity might not be a hard requirement in all smart space environments, there are certain use cases, such as those involving medical data, for which the integrity of the data in the smart space is of high importance. When considering main memory RDF stores, the recoverability of data in case of machine failure or sudden power loss is an issue that cannot be ignored. However, since the RedSIB software does not currently support reversible transactions, transactional data recoverability was considered a preferable feature of the RDF stores, but not a strict requirement. A reasonable system for making regular back-ups of the data would suffice for the RDF stores.

To ensure that the ethos of openness as pertained for the Smart-M3 project, it was considered mandatory that the RDF store should be provided under both an open-source license, and preferably a free-to-use license. This requirement limits the number of possible RDF stores as many of the more mature RDF stores discussed in Sect. 2.3 that support distributed storage are released under a commercial license.

A non-functional requirement worth mentioning is that due to the limited time frame available for the integration of a new RDF store into the Smart-M3 in combination with the extensive use of the Redland storage library in RedSIB, it is mandatory that the chosen RDF store should be interfacable with the RedSIB software through the Redland storage library. A summary of the requirements and desired features for RDF stores are listed in Table 6 below.

Table 6. Features identified as pertinent when considering an RDF store for smart spaces

Criteria	Requirement
Ease of implementation	Should be implementable in 2 months as part of the RedSIB platform
Hardware criterion	Should run on the prototype hardware
Query language	Should support at least the most essential parts of the SPARQL 1.1 standard
Scalability	Should scale to at least ten million triples. Scale-out feature is preferable
Security	Transaction support is preferable for future needs
Data provenance	Named graphs like feature should be supported for future needs
Data persistence	The storage should as bare minimum offer backups

3.5 RDF Stores Short-List

As was presented in the previous section, there are numerous RDF stores available with both free and commercial licensing options. Considering the requirements listed in Table 6, the list of suitable RDF stores for smart space becomes significantly shorter. Based on these criteria, short-listed promising RDF stores were identified: 4store, Virtuoso, OWLIM, Bigdata and RDF-3X. A discussion on the identified alternatives will follow.

3.6 4store in Smart-M3

4store is one of the few distributed RDF stores that is released under both an open-source and a free-to-use license. 4store has performed favourably in the BSBM version 3.1 experiment, performing on par with Virtuoso 6 for a large number of evaluated queries. An advantage to 4store is that the set up process for 4store back ends is not complicated compared to other RDF stores. Additionally, 4store uses a triple representation that is very close to that used in the Redland libraries, and it uses the Raptor and Rasqal Redland libraries, meaning that the integration of 4store into the Redland storage library can reasonably be assumed to be performed within the allotted time frame.

A concern that was raised for the 4store systems was how well the prototype hardware would handle the heavy use of UMAC 64-bit hashing functions in

4store. The developers of 4store offer no guarantees for the performance and stability of 4store on 32-bit hardware, as 4store has only been tested using 64-bit based systems. Additional concerns were that the 4store software in its current form has some other lack in transaction support and that a large part of the SPARQL 1.1 language is yet to be implemented in 4store.

3.7 Virtuoso in Smart-M3

Virtuoso has shown some of the best query performances in RDF benchmarks of all complete RDF stores, especially version 7 of the software. Virtuoso has also been shown to be able to scale up to datasets over trillions of triples [14]. The compliance with the latest version of SPARQL is also good in Virtuoso, and it can be considered a well-documented system with a sizeable number of active developers working on improving the system. The fact that a storage module for Virtuoso 6 has already been created for the Redland storage library means that adding a Virtuoso option to the RedSIB software is a trivial task. The biggest downside for Virtuoso is, that compared to 4store and Bigdata, that the open source version of the software does not support federation.

3.8 Bigdata in Smart-M3

A third alternative considered was the Bigdata software, which is provided under both open-source and free-to-use license. The software is well-documented, and it showed a comparable performance with both 4store and Virtuoso 6 in the BSBM 3.1 experiment.

A major unfavourable factor when considering Bigdata as RDF store in Smart-M3 is that there is a large difference in the data structures used by Bigdata and the triple representation used in the Redland storage library. Due to the limited time to complete the project, creating the necessary interface between Bigdata and *librdf* was concluded infeasible, and therefore Bigdata had to be discarded as a possible candidate for the project.

3.9 RDF-3X in Smart-M3

The forth storage solution considered was to use a state-of-the-art RDF store in Smart-M3. The most promising alternative was identified as RDF-3X, with the motivation being that it performed well in independent benchmarks and that it had a simple interface. Additionally, it was written in C and it uses a simple triple structure, which would make the implementation of the interface to the Redland libraries considerably easier. The official release of the RDF-3X does not support distributed storage out of the box, but it has been shown that RDF-3X can be used in a cluster setting if it is motivated.

3.10 Choice of RDF Store

The requirements discussed in Sect. 3.4 severely limited the possible alternatives for the selection of an RDF store. Most of the distributed RDF stores that were mature and had good scalability were only available under commercial licenses. The only open-sourced distributed mature RDF stores that are released under a free-to-use license are 4store and Bigdata, which show comparable performance results in internal BSBM tests run on the prototype hardware. Based on the observations presented in this section, a choice was made to integrate 4store as a storage option in the RedSIB software.

4 Implementation and Evaluation

As motivated from previous sections, 4store was identified as the only viable addition to the array of RDF storage options for the Smart-M3 platform and the RDF store was subsequently integrated as a storage option in the RedSIB software during a two month time period. This section presents a rough overview of the integration of 4store into the RedSIB software. This presentation is followed by an evaluation of the implementation in comparison to the default RDF storage option in RedSIB.

4store Integration into Smart-M3. As the RedSIB software almost exclusively uses the Redland libraries for handling all its storage needs, it was a natural choice to integrate 4store to the RedSIB software through the Redland storage library. An overview of the logical structure on how the integration of 4store into Smart-M3 was accomplished is outlined in Fig. 3. The additions that were created are: the 4store C front end, which serves as an interface to the functions in the 4store front end; the 4store Model/Storage inside the Redland storage library, which is used to perform database operations on the 4store back end; and the 4store Query module inside the Redland storage library, which is used to evaluate SPARQL queries on the 4store back ends.

4store C Front End. At the start of the implementation, the only interfaces available for accessing the 4store back ends was either through a HTTPS front end [10] or through a command line front end. As these interfaces are not suitable to be used in the Redland storage library, the first point of action was to create a separate 4store front end with C bindings. The aim with the new front end was to support the functionality of both the 4store Model/Storage module and the 4store Query module. A list of the functionality that the 4store C front end should support is presented below:

- creation of connections to 4store back ends
- addition of individual triples
- bulk insertion of triples
- removal of individual triples

– removal of entire named graphs
– evaluation of SPARQL queries

The front end was implemented as a Linux shared library. The shared library was based on, unrelated to the work done towards this chapter, work done by the Perl Community during a Perl Hackathon event in London 2012. The original library had the functionality of supporting basic triple matching operations and was aimed to be an addition to the Trine framework [11]. As the syntax for RDF::Trine is very similar to that of the Redland storage API, most of the work performed in the original library could be used directly with only minor modifications. The additions made especially for the librdf integration consisted of functionality for adding and removing triples and performing SPARQL queries on 4store back ends. Transactional support was not included as it is not yet supported in 4store, but it could be added to the 4store C front end if the support for it was added to the 4store software in the future.

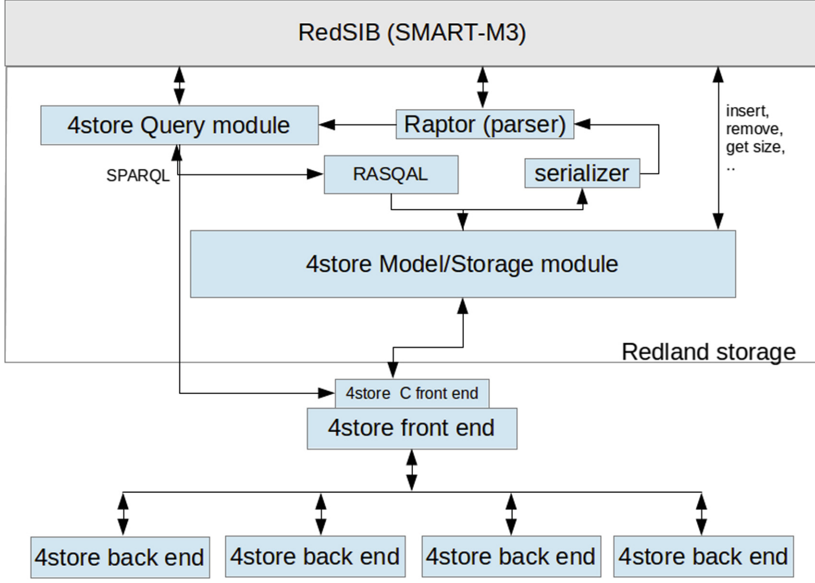

Fig. 3. Logical overview of the integration of 4store into Redland *librdf* and Smart-M3

Integration of 4store into the Redland Libraries. After the appropriate functionality had been added to the 4store C front end, the integration of 4store into the librdf library was started. First the 4store Model/Storage module was integrated and when it was completed, the integration of the 4store Query module followed.

4store uses the Raptor [16] library to parse and serialize RDF data and the RASQAL [17] library to parse SPARQL queries that both are part of the Redland

library collection. Additionally, the triple representation structure in the Redland libraries is similar to the ones used in 4store. The similarities alleviated the implementation of a large part of the functions for the 4store Model/Storage and the Query modules as the need for triple representation transform was minimal.

4store Model/Storage Module. The librdf library makes a distinction between Model and Storage. For the access of the underlying storage solution, the librdf library uses two separate modules: a *model* module that works as an interface that the user can call to access the triples and a *storage* module that the model module can call upon to perform operations in the underlying storage solution. Applications create instances of the storage module that can later be bound to an instance of a model module. A more in-depth description of the modules and their functionality can be found in an article by Dave Beckett released in 2001 [23]. For simplicity purposes, the Model and the Storage modules are treated as a single module in this work.

Model/Storage modules inside librdf give a limited number of functions that can be called from applications to perform actions on the underlying storage. The functions can be grouped based on their functionality into actions that are related to the creation, initialization and closing of connections to the underlying storage module, the addition and removal of statements to and from the storage module, fetching of triples from the storage using triple matching patterns and the optional transactional and statement context-related functionality.

The 4store Model/Storage module was implemented in a similar fashion to the respective Virtuoso Model/Storage module. The functions provided in the Model/Storage module are transformed into the appropriate 4store C front end operations that ensure that the relevant actions are performed in the 4store knowledge base.

4store Query Module. The librdf has an internal query handler module in librdf for query processing and an external query module for using the RDF stores own query engine. The librdf internal query processing in librdf is performed by translating the SPARQL query into corresponding RASQAL statements that can be evaluated on the storage module. The functionality of the internal query processing only encompassed a limited set of the SPARQL language with most of the SPARQL 1.1 features yet to be implemented at the moment.

For native RDF stores that implement their own query engine, the internal query processing in librdf can be considerer to be rather inefficient as a result of fact that the query optimization of the query engine in librdf is done based on the in-data structures of the embedded storage modules. In addition to that, the query capabilities of the librdf are limited in functionality compared to query engines in native RDF stores. It was therefore a conscious decision to let the query engine process all the queries for 4store.

When using the 4stores own query engine, the queries are passed directly onto the 4store C front end, in which the 4stores' own query engine evaluates

the query. The query bindings that are produced in the 4store query engine are passed back to the Redland query interface where they are processed and serialized. Queries can also be performed through the librdf's internal query engine, but without any guarantees of accuracy and performance of receiving the correct results.

Changes in RedSIB. No major structural changes had to be made inside the RedSIB software. An option to use the 4store module was included in the same fashion as the other storage option, with the exception that the 4store storage instance was set to use the 4stores' own query engine for evaluating SPARQL queries and importing multiple triples to 4store is performed as an bulk operation.

4.1 Experiments

Almost all the functionality that was set out in the planning phase of the work was completed during the allotted time. All the functionality for the 4store storage module in librdf thought of in the planning phase was also indeed implemented. The bulk insert was implemented using the same procedure as in 4store itself. The similarities on how triples handles in both 4store and librdf made the work easier. The major difference was that 4store stores triples as quads, while librdf stores triples as triple statements extended with the context field. This difference was resolved by setting the context resource to the name of the Smart-M3 smart space instance name. In RedSIB, this implies that all triples will be inserted in 4stores with the smart space instance name as the model for all triples.

The evaluation of the implementation non-functional aspects is more difficult, mostly due to the vagueness of the criteria set out in Sect. 3.4. An attempt to measure the query performance and scalability was nonetheless performed using the LUBM data generator [37] and the provided test queries.

LUBM Experiment Setup. The test queries ware performed on the Smart-M3 system using data generated with the LUBM data generator using options for 1 and 10 universities. The LUBM dataset size for one university option consists of approximately 100K triples and the dataset for the ten universities option consists of approximately 1,2 million triples. The 14 text queries from the LUMB test suite was then performed on the Smart-M3 system with the BDB and 4store storage options through the Smart-M3 Python KPI interface.

LUBM Experiment Results. The results from the benchmark are displayed in Figs. 4 and 5 below. Missing from the figure are the results from the BDB option as they could not be produced with that storage option. Even with the smaller dataset, none of the queries could successfully be performed when the BDB storage was used.

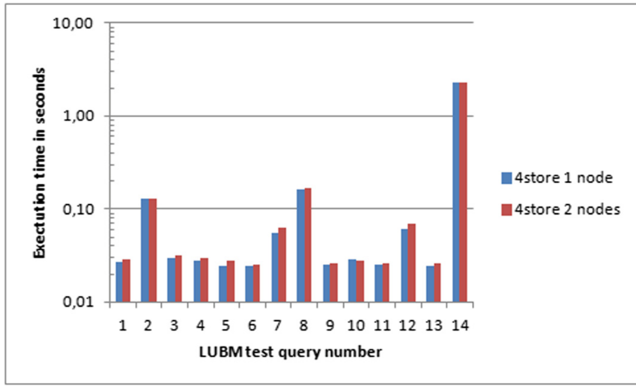

Fig. 4. Experimental run of LUBM test queries for the dataset with one university

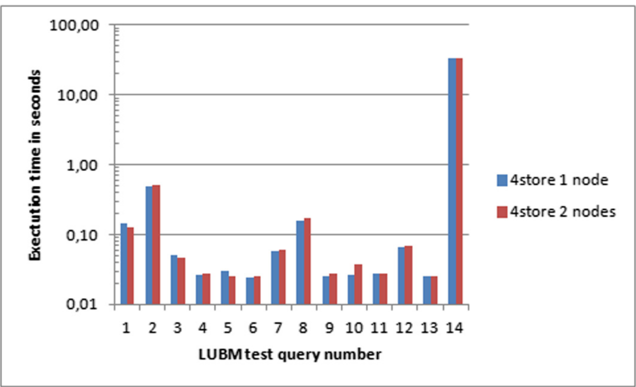

Fig. 5. Experimental run of LUBM test queries for the dataset with ten universities

As can be noted in Figs. 4 and 5 above, the query execution times are not that affected by the number of 4store back end nodes that are used in small to moderately sized datasets. The main advantage of the distributed 4store knowledge base is that it makes the system as a whole capable of storing larger datasets. The inability of the Smart-M3 system with the BDB storage option to produce results even with the smallest dataset clearly points out the limitations of the query capabilities for the default storage option in the Smart-M3.

The implementation of the Smart-M3 query modules can take SPARQL queries and execute them without any major overhead compared to running the queries directly through the other 4store front end. SPARQL UPDATE operations were not implemented in the 4store query module in librdf as the knowledge base modification operations that are already present in the librdf storage module were considered sufficient to serve the needs of the smart space. As can be noted from the experiment results, the 4store storage option in Smart-

M3 can serve significantly larger dataset sizes than the default storage option of the Berkeley DB. With regards to this, the 4store storage option in Smart-M3 can be considered successful.

An unanticipated flaw detected in 4store during the implementation was that the inserting of individual triples is a highly inefficient process compared to the bulk insertion of triples. A modification of a single triple results in that all the indices in 4store that contain that triple must also be updated. This feature of 4store was not properly taken into account during the planning phase, leading to significantly slower single triple insertion times compared to the BDB based storage module. The long time it takes to update the indices of a large 4store knowledge bases can be considered a very unfavourably feature in smart space use cases for which the majority of operations are related to the addition and removal of individual triples.

5 Discussion and Future Work

In this chapter, the RDF store's landscape was outlined based on publicly available literature in Sect. 2.3 and summarized in Table 7. The Smart-M3 platform was introduced and the problems related to the current RDF store in the RedSIB were identified in Sect. 3. According to the findings, when exploring available RDF stores, the most suitable RDF store for the Smart-M3 project was identified as the 4store. The implementation and results of the integration of the chosen system into a Smart Space environment were presented in Sect. 4.

Even if the integration of 4store on the whole system was successful, the 4store had drawbacks that resulted in a poor performance in Smart Space use cases consisting of single triple addition and removal. Therefore, further scalability storage solutions to run efficiently on low-power devices must be studied.

The Smart-M3 system was not able to scale to the lengths envisioned for an off-line ambient intelligence setting. This inability is due to factors outside the scope of the underlying RDF store. The most obvious fault of Smart-M3 is the incapacity to import knowledge bases larger than 60K triples at a time, the limits of the restricted SSAP protocol and the overall instability of the system.

Furthermore, the best suiting RDF store choice is highly dependent on the intended use case. The type of low-energy profile hardware used in Smart Space environments needs a different type of RDF store than for example large scale Web data mining systems. Finding the right RDF store for the use case and the hardware implies knowledge about the system needs before it is built, a task that can be very challenging to predict. Luckily, the RDF framework is very lenient when it comes to migrating from an RDF store to another. This means that several RDF stores need to be evaluated in order to find the right one by using a minimal amount of effort. Future work should evaluate newer RDF stores such as BitMat, MonetDB, TriAD, AdPart, H2RDF, etc. Other benchmark tests may include WatDiv, Bio2RDF, Yago2. While we focused on reasoning capabilities, other RDF store functionality should be further assessed depending on the use case; for instance, evaluating OWL reasoning, filter capabilities, nested queries,

Table 7. List of features for a selection of RDF stores

Name, creator	Storage technique	Programming language	State	Licence	Supported query languages	Distributed storage	Semantics support	Documentation quality	Discussed in this chapter
4store, Garlik	Custom DBMS	C	Open-source development	GPL v3	SPARQL	Yes	None/separate RDFS version	Poor	Yes
5store, Garlik	Custom DBMS	C	Active	Commercial	SPARQL	Yes	Unknown	Not available	No
AllegroGraph, Franz Inc	Graph store	Lisp	Active	Closed source free/ commercial	SPARQL	Free/ commercial	Unknown	Good	No
BigData, SYSTAP LLC		Java	Active	GPL2/ commercial	SPARQL	Yes	RDFS	Decent	Yes
OWLIM-SE, Ontotext		Java	Active	Commercial	SPARQL	Yes	RDFS, OLW Horst and Max, OWL2 QL	Good	Yes
Mulgara	Unknown	Java	Active	OSL-3.0 and Apache v2	SPARQL	Yes	Unknown	Decent	No
Ontobroker, Semafora	3d-party	Java	unknown	Commercial	subset of SPARQL	Yes	RDFS OWL, OWL2	Decent	No
Oracle Spatial and Graph 11g/12c	Oracle DB	Java	active	Commercial	SPARQL	Yes	RDFS OWL2 SKOS	Decent	No
OWLIM-Lite, Ontotext		Java	Active	LGPLv2	SPARQL	No	RDFS, OLW Horst and Max, OWL2 QL and RL	Good	Yes
RDF-3X, Planc institute	Vertex partitioning	C	Academic research	Free for non-commercial use	Limited SPARQL 1.1	Yes	None	Poor	Yes
Virtuoso 6/7, Openlink	Relational table/column graph	C/C++	Active	GPL v2 and commercial	SPARQL	Yes	Limited OWL2	Good	Yes
YARS2	Unknown	Java	Discontinued	Find out	SPARQL	I guess not but could be	None	Poor	To some extent
Trinity.RDF, Microsoft research	Graph database	Java	Research	Unknown	SPARQL 1.1	Yes	Find out	Not available	Yes
Jena TDB	Main memory	Java	Active	Apache 2.0	SPARQL 1.1	Find out	Yes	Good	No
Jena SDB	RDBMS	Java	Active	Commercial	SPARQL 1.1	Find out	Yes	Good	No
RYA	Main memory	Same as Accumulo	Java	Academic research	SPARQL	Yes	No	Poor	No

property paths, etc. For less low-power and less domain specific review on RDF storage and solutions we refer the reader to [34,47,54].

Distributing the knowledge base over several RDF store nodes is not a choice that should be taken lightly. Even though, in theory, the distribution seems to provide a good way in order to achieve scalability in a system, in many cases, the distribution often means adding complexity to the system that cannot be motivated by the upsides of the distributed storage. When dealing with Smart Spaces, one can speculate that in most cases it is best to store the knowledge base on one node and to delay the distribution of RDF stores until the system cannot possibly scale vertically any more. Again, the standardized RDF data format makes the migration to a scalable RDF store easy.

A lot of interesting research is being conducted within both RDF stores and energy-efficient devices. Future work within the context of RDF stores in Smart Spaces would be to further explore a wider array of RDF stores and to investigate how well they perform on a range of low-power hardware suitable for Smart Spaces.

Acknowledgments. The authors acknowledge EU COST Action IC1303 Algorithms, Architectures and Platforms for Enhanced Living Environments (AAPELE www.aapele.eu) within the WG4 - Medical Data Acquisition and Algorithms.

References

1. Allegrograph. http://franz.com/agraph/support/documentation/current/agraph-introduction.html
2. Apache Jena website. https://jena.apache.org/
3. Apache Lucene, a high-performance, full-featured text search library. http://lucene.apache.org/
4. Named graph wikipedia page. http://en.wikipedia.org/wiki/Named_graph
5. Odroid XU development board. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G137510300620
6. OpenRDF Semame website. http://www.openrdf.org/
7. Reduced instruction set computing Wikipedia page. http://en.wikipedia.org/wiki/Reduced_instruction_set_computing
8. Architecture for Sofia Interoperability Platform - Deliverable 5.22: Logical Service Architecture. ARTEMIS JU SP3 D5.22-v1.0, ARTEMIS JUs SP3/100017: Smart Objects For Intelligent Applications (SOFIA), March 2009. http://www.sofia-community.org/files/SOFIA_D5-22-LogicalServiceArchitecture-v1-2011-01-02_0.pdf
9. Oracle NoSQL Database. White Paper, September 2011. http://www.oracle.com/technetwork/database/nosqldb/learnmore/nosql-database-498041.pdf
10. 4store SPARQL HTTP server wiki (2012). http://4store.org/trac/wiki/SparqlServer
11. RDF: Trine - An RDF Framework for Perl. Webpage (2012). http://search.cpan.org/gwilliams/RDF-Trine-1.007/lib/RDF/Trine.pm
12. Redland librdf RDF API: library (2012)
13. Database, bigdata: architecture, May 2013

14. BSBM V3.1 Results, April 2013. http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/results/V7/index.html
15. Oracle Spatial and Graph: 12c RDF. Semantic graph (2013)
16. Raptor RDF syntax: library (2013)
17. Rasqal RDF query: library (2013)
18. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.: SW-store: a vertically partitioned DBMS for Semantic Web data management. VLDB J. **18**(2), 385–406 (2009)
19. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Using the Barton libraries dataset as an RDF benchmark. Technical report, MIT-CSAIL-TR-2007-036, MIT (2007)
20. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of RDF data management systems. In: Mika, P., et al. (eds.) ISWC 2014. LNCS, vol. 8796, pp. 197–212. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11964-9_13
21. Atre, M., Chaoji, V., Zaki, M.J., Hendler, J.A.: Matrix bit loaded: a scalable lightweight join query processor for RDF data, pp. 41–50(2010)
22. Beauregard, B.: Oracle database 11g semantic technologies. Technical report (2011)
23. Beckett, D.: The design and implementation of the Redland RDF application framework. In: Proceedings of the 10th International Conference on World Wide Web, WWW 2001, pp. 449–456. ACM, New York (2001)
24. Bishop, B., Kiryakov, A., Ognyanoff, D., Peikov, I., Tashev, Z., Velkov, R.: OWLIM: a family of scalable semantic repositories. Technical report (2010)
25. Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. Int. J. Semant. Web Inf. Syst. **5**(2), 1–24 (2009)
26. Callahan, A., Cruz-Toledo, J., Ansell, P., Dumontier, M.: Bio2RDF release 2: improved coverage, interoperability and provenance of life science linked data. In: Cimiano, P., Corcho, O., Presutti, V., Hollink, L., Rudolph, S. (eds.) ESWC 2013. LNCS, vol. 7882, pp. 200–212. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38288-8_14
27. Chang, F., et al.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. **26**(2), 4:1–4:26 (2008)
28. Díaz-Rodríguez, N., Kankaanpää, P., Saleemi, M.M., Lilius, J., Porres, I.: Programming biomedical smart space applications with bioimagexd and pythonrules, pp. 10–11(2011)
29. Rodríguez, N.D., Lilius, J., Cuéllar, M.P., Calvo-Flores, M.D.: Extending semantic web tools for improving smart spaces interoperability and usability. In: Omatu, S., Neves, J., Rodriguez, J.M.C., Paz Santana, J.F., Gonzalez, S.R. (eds.) Distributed Computing and Artificial Intelligence. AISC, vol. 217, pp. 45–52. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-00551-5_6
30. Duan, S., Kementsietsidis, A., Srinivas, K., Udrea, O.: Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, pp. 145–156. ACM, New York (2011)
31. Erling, O.: Virtuoso, a hybrid RDBMS/graph column store. IEEE Data Eng. Bull. **35**(1), 3–8 (2012)
32. Erling, O., Mikhailov, I.: RDF support in the virtuoso DBMS. In: Pellegrini, T., Auer, S., Tochtermann, S., Schaffert, S. (eds.) Networked Knowledge - Networked Media. SCI, vol. 221, pp. 7–24. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02184-8_2

33. Erling, O., Mikhailov, I.: Virtuoso: RDF support in a native RDBMS. In: de Virgilio, R., Giunchiglia, F., Tanca, L. (eds.) Semantic Web Information Management, pp. 501–519. Springer, Berlin (2009). https://doi.org/10.1007/978-3-642-04329-1_21
34. Faye, D.C., Curé, O., Blin, G.: A survey of RDF storage approaches. ARIMA J. **15**, 11–35 (2012)
35. Filali, I., Bongiovanni, F., Huet, F., Baude, F.: A survey of structured P2P systems for RDF data storage and retrieval. In: Hameurlain, A., Küng, J., Wagner, R. (eds.) Transactions on Large-Scale Data- and Knowledge-Centered Systems III. LNCS, vol. 6790, pp. 20–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23074-5_2
36. Goasdoué, F., Kaoudi, Z., Manolescu, I., Quiané-Ruiz, J., Zampetakis, S.: CliqueSquare: efficient Hadoop-based RDF query processing. In: Journées de Bases de Données Avancées, BDA 2013, Nantes, France, October 2013
37. Guo, Y., Pan, Z., Heflin, J.: LUBM: a benchmark for OWL knowledge base systems. Web Semant. Sci. Serv. Agents World Wide Web **3**(2–3), 158–182 (2005)
38. Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M.: TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing, pp. 289–300 (2014)
39. Haarslev, V., Hidde, K., Möller, R., Wessel, M.: The RacerPro knowledge representation and reasoning system. Semant. Web **3**(3), 267–277 (2012)
40. Harbi, R., Abdelaziz, I., Kalnis, P., Mamoulis, N., Ebrahim, Y., Sahli, M.: Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. VLDB J. Int. J. Very Large Data Bases **25**(3), 355–380 (2016)
41. Harris, S., Gibbins, N.: 3store: efficient bulk RDF storage, June 2003
42. Harris, S., Lamb, N., Shadbolt, N.: 4store: the design and implementation of a clustered RDF store. In: Scalable Semantic Web Knowledge Base Systems, SSWS 2009, pp. 94–109 (2009)
43. Harth, A., Decker, S.: Optimized index structures for querying RDF from the web. In: Proceedings of the Third Latin American Web Congress, LA-WEB 2005, Washington, DC, USA, p. 71. IEEE Computer Society (2005)
44. Haslhofer, B., Roochi, E.M., Schandl, B., Zander, S.: Europeana RDF store report. Technical report. University of Vienna, Vienna, March 2011
45. Hosseinzadeh, S., Virtanen, S., Díaz-Rodríguez, N., Lilius, J.: A semantic security framework and context-aware role-based access control ontology for smart spaces, p. 8 (2016)
46. Mahdisoltani, F., Biega, J., Suchanek, F.M.: YAGO3: a knowledge base from multilingual Wikipedias. In: CIDR (2013)
47. Modoni, G.E., Sacco, M., Terkaj, W.: A survey of RDF store solutions. In: 2014 International Conference on Engineering, Technology and Innovation (ICE), pp. 1–7, June 2014
48. Morandi, F., Roffia, L., D'Elia, A., Vergari, F., Cinotti, S.T.: RedSib: a smart-M3 semantic information broker implementation. In: Balandin, S., Ovchinnikov, A. (eds.) Proceedings of the 12th Conference of Open Innovations Association FRUCT, Oulu, Finland, pp. 86–98. State University of Aerospace Instrumentation (SUAI), November 2012
49. Morsey, M., Lehmann, J., Auer, S., Ngomo, A.-C.N.: DBpedia SPARQL benchmark - performance assessment with real queries on real data. In: ISWC 2011 (2011)
50. Murray, C.: Oracle® spatial resource description framework (RDF). Technical report, July 2005. http://download.oracle.com/otndocs/tech/semantic_web/pdf/rdfrm.pdf

51. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. VLDB J. **19**(1), 91–113 (2010)
52. Neumann, T., Weikum, G.: x-RDF-3X: fast querying, high update rates, and consistency for RDF databases. Proc. VLDB Endow. **3**(1–2), 256–263 (2010)
53. Oraskari, J., Törmä, S.: Smart–M3 storage solutions. Aalto University, Department of Computer Science and Engineering, July 2011
54. Özsu, M.T.: A survey of rdf data management systems. Front. Comput. Sci. **10**(3), 418–432 (2016)
55. Papailiou, N., Konstantinou, I., Tsoumakos, D., Karras, P., Koziris, N.: H_2 RDF+: high-performance distributed joins over large-scale RDF graphs, pp. 255–263. IEEE (2013)
56. Rohloff, K., Schantz, R.E.: High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store. In: PSI EtA, p. 4 (2010)
57. Saleemi, M.M., Díaz-Rodríguez, N., Lilius, J.: Erratum to: exploiting smart spaces for interactive TV applications development. J. Supercomput. **70**(3), 1617 (2014)
58. Mohsin Saleemi, M., Díaz Rodríguez, N., Lilius, J., Porres, I.: A framework for context-aware applications for smart spaces. In: Balandin, S., Koucheryavy, Y., Hu, H. (eds.) NEW2AN/ruSMART -2011. LNCS, vol. 6869, pp. 14–25. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22875-9_2
59. Saleemi, M.M., Suenson, E., Lilius, J., Porres, I.: Ontology driven smart space application development. In: Semantic Interoperability: Issues, Solutions, and Challenges, pp. 101–125 (2012)
60. Salvadores, M., Correndo, G., Harris, S., Gibbins, N., Shadbolt, N.: The design and implementation of minimal RDFS backward reasoning in 4store. In: Antoniou, G., et al. (eds.) ESWC 2011. LNCS, vol. 6644, pp. 139–153. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21064-8_10
61. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP2Bench: a SPARQL performance benchmark. CoRR, abs/0806.4627 (2008)
62. Shao, B., Wang, H., Li, Y.: Trinity: a distributed graph engine on a memory cloud. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, pp. 505–516. ACM, New York (2013)
63. Sidirourgos, L., Goncalves, R., Kersten, M., Nes, N., Manegold, S.: Column-store support for RDF data management: not all swans are white. Proc. VLDB Endow. **1**(2), 1553–1563 (2008)
64. Stegmaier, F., Gröbner, U., Döller, M., Kosch, H., Baese, G.: Evaluation of current RDF database solutions. In: Proceedings of the 10th International Workshop on Semantic Multimedia Database Technologies (SeMuDaTe), 4th International Conference on Semantics and Digital Media Technologies (SAMT) (2009)
65. Thompson, B.: Literature Survey of Graph Databases, January 2013. http://www.systap.com/pubs/graph_databases.pdf
66. Wilkinson, K.: Jena property table implementation. In: SSWS, Athens, Georgia, USA, pp. 35–46 (2006)
67. Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z.: A distributed graph engine for web scale RDF data. Proc. VLDB Endow. **6**(4), 265–276 (2013)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.