Semantic Web 1 (2016) 1–5
IOS Press

Warehousing Linked Open Data with Today's Storage Choices

Editor(s): Name Surname, University, Country
Solicited review(s): Name Surname, University, Country
Open review(s): Name Surname, University, Country

Timm Heuss [a]
[a] *Fraunhofer Institute for Communication, Information Processing and Ergonomics FKIE*
E-mail: Timm.Heuss@fkie.fraunhofer.de

Abstract: This paper compares the performance of current storage technologies when warehousing Linked Open Data. This involves common CRUD operations on relational databases (PostgreSQL, SQLite-Xerial and SQlite4java), NoSQL databases (MongoDB and ArangoDB) and triple stores (Virtuoso and Fuseki). Results indicate that relational approaches perform well or best in most disciplines and provide the most stable operation. Other approaches show individual strengths in rather specific scenarios, that might or might not justify their deployment in practice.

Keywords: Database, Benchmark, Linked Open Data, Warehouse, PostgreSQL, SQLite-Xerial, SQlite4java, MongoDB, ArangoDB, Virtuoso, Fuseki

1. Introduction

Warehousing integrated data [11] is a well-tried, battle-tested architectural pattern for applications, providing a number of benefits: It allows developers to pre-select the data, according to exact information needs of the given application. Data can specifically be enriched, modified and streamlined to typical use cases. The application's quality, reliability and performance does not depend on third-party services.

Nowadays, there are many storage technologies and frameworks available to implement such a warehouse. The approaches range from schema-based relational Database Management System (DBMS), over triple stores that store atomic information following the Linked Data paradigms (such as the LOD2 Stack [3] or Marmotta / Linked Media Framework [13]), to Not Only SQL (NoSQL) systems [7] with more flexibility regarding their data schema, such as LibreCat / Catmandu [14] or the MEAN stack [15].

This multiplicity of storage technologies is however not reflected in existing comparisons of the Web Science community, such as the Berlin SPARQL Query Language (SPARQL) Benchmark (BSBM) [4] or the SPARQL Performance Benchmark (SP^{2}Bench) [19]. So currently, to the knowledge of the author, there is no evaluation that combines all the following properties:

- tests fundamental data operations that are of interest for a warehouse scenario
- is designed in a technology-agnostic way
- evaluates the whole spectrum of storage choices available today, including relational DBMS, NoSQL, and triple store approaches
- operates on real-world Linked and Open Data

This lack leaves uncertainties for Linked and Open Data developers and architects when making an essential architectural decision. Selecting the right storage technology is a crucial for the application performance and reliability, and needs to be backed up by realistic comparisons. Given the fact that earlier versions of the BSBM showed significant performance gaps between different data storage approaches, such a comparison can considered to be justified and highly demanded.

2. Methodology

According to Gray, domain-specific benchmarks should meet the following criteria [10, p. 3]:

Relevant - measures must be settled within the specific domain of interest.
Portable - implementation should be possible on different systems and architectures.
Scaleable - scaling the benchmark should be possible.
Simple - the benchmark should be easily understandable.

In regard of these properties, a benchmark endeavour is be defined in the following.

All scripts, queries and program source code is published in the GitHub repository loddwhbench[1].

2.1. Queried Data

The exact choice for a certain dataset is essential, as it focuses the application scenario, and queries must be designed to work with it. So in order to make the benchmark relevant for a specific scenario [10, p. 3f], the selected dataset must be application specific.

In this comparison, the bibliographic catalogue of the Hessisches Bibliotheks- und Informationssystem *(Library and Information System of Hesse)* (HeBIS) will be used as the foundation of the dataset and query scenarios. This exact dataset is suitable for multiple reasons: firstly, it is relevant, as it is true, real-world Linked and Open Data, available at the data portal datahub.io[2]. Secondly, it comprises about 14 GB of data - about 11 million books - so the benchmark can be scaled at arbitrary sizes from small query scenarios up to large ones. Thirdly, it is provided in native Resource Description Framework (RDF), allowing to compare the efforts when RDF can be loaded natively (in triple stores), whereas, in contrast, RDF must first be converted in another structure (e.g. in relational approaches). Table 1 shows the fields of the HeBIS catalogue in detail, including a sample entry.

Different sizes of the HeBIS catalogue are compared in different *test series*, as required by the scalability criteria [10, p. 3f]:

TINY A random selection of 1,000 records.

Property / Column	Samples from the dataset
DCTERMS_IDENTIFIER	334023971
BIBO_OCLCNUM	3252176
RDF_ABOUT	http://lod.hebis.de/resource/33402397
DCTERMS_TITLE	Where the gods are mountains
DCTERMS_PUBLISHER	Weidenfeld and Nicolson
ISBD_P1017	Weidenfeld and Nicolson
ISBD_P1016	London
ISBD_P1008	1. publ. in Great Britain
ISBD_P1006	three years among the people of the Himalayas
ISBD_P1004	Where the gods are mountains
ISBD_P1018	1956
DCTERMS_ISSUED	1956
OWL_SAMEAS	http://d-nb.info/1034554514
RDF_TYPE	{dcterms#BibliographicResource, frbr#Manifestation, bibo#Book}
DCTERMS_MEDIUM	paper
DCTERMS_FORMAT	print
BIBO_EDITION	1. publ. in Great Britain
WDRS_DESCRIBEDBY	http://lod.hebis.de/catalog/html/33402397
DCTERMS_SUBJECT	http://d-nb.info/gnd/4024923-2

Table 1

Per-entry properties of the HeBIS catalogue, including a sample entry (with prefix.cc-based prefixes), following the DINI-AG KIM recommendation [5]. Usually, not all properties are set for every record. Also some fields may contain multiple values, e.g. RDF_TYPE.

SMALL A random selection of 100,000 records.
MEDIUM About 2,2 million records[3], loaded from `hebis_10147116_13050073_rdf_gz`.
LARGE About 5,8 million records[3], loaded from `hebis_10147116_13050073_rdf_gz` and `hebis_29873806_36057474_rdf_gz`.

2.2. Database Operations

All basic data manipulation operations - *Create, Read, Update, Delete (CRUD)* - as well as fundamental schema changes - are likely and relevant:

Schema Change describes operations that change the structure of stored entities, including introduction or removal of attributes. In a data warehouse, this happens when existing data is migrated. In triples stores, this operation is conceptually impossible because there is no predefined entity structure.

[1] https:/ / github.com/ heussd/ loddwhbench, last access on 2016-02-15.

[2] http://datahub.io/dataset/hebis-bibliographic-resources, last access on 2015-09-14.

[3] The HeBIS catalogue is delivered in seven GZIP-compressed RDF/XML parts. As each part can be loaded individually, the sizes of the MEDIUM and LARGE test series represent exactly one (MEDIUM) or two (LARGE) specific original parts of the catalog.

Create includes operations that create or load entities initially. Where the DBMS is schema-aware, this usually involves steps for the creation of structures (e.g. tables in relational DBMS) and entity validation. In a warehouse scenario, this happens when data is imported, e.g. within an Extraction, Transform, Load (ETL) step.

Read describes operations to access stored information. This usually includes more advanced queries, containing features such as aggregations, groups or joins. In a data warehouse, reading is the fundamental usage scenario to answer several kind of questions.

Update specifies operations to change stored information. Again, in the case the DBMS is schema-aware, this usually involves validation of inserted information. In a data warehouse, this also happens as part of an ETL-process.

Delete includes the deletion of stored information. In a warehouse scenario, deletion happens for example when outdated data needs to be removed.

Naturally, the performance of many operations is dependent on their *selectivity* - the amount of information that is affected on the exact operation. In a warehouse scenario, this selectivity might occur in two principle variants:

High Selectivity The pre-selection of data is specific, little information is affected by the testified operation.

Low Selectivity The pre-selection of data is unspecific, the operation affects large parts of stored information.

2.3. Measured Metrics

Read-only operations are measured three times, and all timings are logged individually. This way, the impact of caching mechanisms can be observed. Write-based operations, including initial dataset loading times, are executed and measured once. Measuring plain query performance, however, does not represent real application scenarios. In real scenarios, the performance of intermediate layers such as Java Database Connectivity (JDBC) is just as important, as these layers make the content actually actionable for the application's program logic. Time measures in this benchmark thus not only contain the query times a DBMS takes, but the time required to make the results of these queries available in a dedicated software object, that is equally expected to be delivered from all databases.

2.4. Query Scenarios

The *query scenarios* cover specific schema change, read, update and delete operations, making use of the nature of the HeBIS catalogue. The table does not contain create operations, as they are covered by the initial load of the datasets.

2.5. Execution Phases

Putting it together, for a given database implementation, the benchmark is executed in the following phases:

Set Up Initialises the environment of the given database, makes sure it is available, prepares and clears internal structures. It is only executed once for each database.

Load Imports a given test series (see page 2), consisting of one or more files. The required time is measured. Errors in this step invalidate all follow-up timings.

Prepare Instructs the database to prepare for a specific query scenario (see page 4). This preparation might include creation of additional data structures, such as index views, etcetera. Timings are measured for this phase. It is executed once for each query scenario.

Query Executes one or more queries, required to fulfil a certain query scenario (see page 4). If the query scenario only contains read operations, this phase is executed three times to observe caching behaviour. All timings are recorded individually.

2.6. Storage Choices

The choice for the considered databases in the following is driven by engineering: which candidate would be used in practice? Besides this technical consideration, licensing costs play a role, too. As this work seeks to help a wide range of developers to benefit from Linked and Open Data, the required infrastructure should be open and available cross-platform. This leaves the ultimate freedom of choice to the developer. It should also foster experiments out of curiosity and help to reduce costly upfront investment in the technology.

Existing Web Science benchmarks are usually based on triple stores. Conceptually, a fundamental advantage of triple stores is the fact that loading existing Linked Data is easy and effortless, as the database can

	Query scenario	Description
Schema Change	SCHEMA_CHANGE_MIGRATE_RDF_TYPE	Create a boolean field for every single RDF_TYPE, each indicating if a record is of this type. After that, remove RDF_TYPE from every record.
	SCHEMA_CHANGE_REMOVE_RDF_TYPE	Remove RDF_TYPE from every record.
	SCHEMA_CHANGE_INTRODUCE_NEW_PROPERTY	Create a new property for every record, defaulting to "cheesecake".
	SCHEMA_CHANGE_INTRODUCE_STRING_OP	Take substring of ID (RDF_ABOUT) and store it in a new property.
Read	ENTITY_RETRIEVAL_BY_ID_ONE_ENTITY ENTITY_RETRIEVAL_BY_ID_TEN_ENTITIES ENTITY_RETRIEVAL_BY_ID_100_ENTITIES	Retrieve the first, ten or 100 entities of a list of all complete entities, ordered by DCTERMS_MEDIUM, ISBD_P1008, DCTERMS_CONTRIBUTOR, and DCTERMS_SUBJECT.
	AGGREGATE_PUBLICATIONS_PER_PUBLISHER_TOP10 AGGREGATE_PUBLICATIONS_PER_PUBLISHER_TOP100 AGGREGATE_PUBLICATIONS_PER_PUBLISHER_ALL	Count the publications per publisher and order by this count descending. Return two columns (publisher + the respective count). Show 10, 100 highest or all highest counts.
	AGGREGATE_ISSUES_PER_DECADE_TOP10 AGGREGATE_ISSUES_PER_DECADE_TOP100 AGGREGATE_ISSUES_PER_DECADE_ALL	Count the publications per issued century and order by this count descending. Return two columns (century + the respective count). Show the 10, 100 highest or all counts.
	CONDITIONAL_TABLE_SCAN_ALL_STUDIES CONDITIONAL_TABLE_SCAN_ALL_BIBLIOGRAPHIC_RESOURCES CONDITIONAL_TABLE_SCAN_ALL_BIBLIOGRAPHIC_RESOURCES_AND_STUDIES	Returns all complete entities matching: 1) "Studie" or "Study" (case insensitive) in their title (about 2% of the records). 2) "Bibliographic Resources"-type (about 92% of all entities). 3) both 1) and-chained with 2)
	GRAPH_LIKE_RELATED_BY_DCTERMS_SUBJECTS_1HOP	For each record, find records that share a DCTERMS_SUBJECT and return record identifiers and shared subjects.
Update	UPDATE_LOW_SELECTIVITY_PAPER_MEDIUM	Set DCTERMS_MEDIUM to "recycled trees" on records having DCTERMS_MEDIUM == "paper", affects about 90% of the records.
	UPDATE_HIGH_SELECTIVITY_NON_ISSUED	Set DCTERMS_ISSUED to 0 on records that have no value for this property, affects about 2% of the records.
Delete	DELETE_LOW_SELECTIVITY_PAPER_MEDIUM	Remove records having DCTERMS_MEDIUM == "paper" (about 90% of the records).
	DELETE_HIGH_SELECTIVIY_NON_ISSUED	Remove records that have no value for DCTERMS_ISSUED (about 2% of the records).

Table 2

Query scenarios of this evaluation. They cover schema change, read, update and delete operations. Create operations are covered in the load of datasets.

parse RDF natively. Thus, in this evaluation, two triple stores are considered: Virtuoso, usually one of the best performers in existing benchmarks, as well as Apache Fuseki.

A traditional way to store data in applications are relational DBMS. Regarding the fact that earlier works of the BDSM pointed out significant differences between triple stores and relation databases, it is interesting to include PostgreSQL and the SQLite-based sqlite4java and SQLite-Xerial as well. In contrast to the latter one, sqlite4java has no JDBC abstraction layer, and thus claims to be faster [2].

In the context of NoSQL, alternative storage forms became popular, battle-tested and developer-friendly as well. Thereby, conceptually, a promising class to handle (large) collections of homogeneous data is a document store. In this evaluation, a popular implementation, MongoDB, is included in the comparison. It is used in a number of recently advocated architectures, such as the MEAN stack [15].

Recently, databases started combining multiple NoSQL approaches [6]. One sample for such a database is ArangoDB, which is a document store, graph store as well as a key/value store. To see how this database

	Open Virtuoso	PostgreSQL	sqlite4java	SQLite-Xerial	MongoDB	ArangoDB	Fuseki
Version	07.20.3214	9.4.4.1	392 (3.8.7)	3.7.2	3.0.6	2.6.9	2.3.0
Consistency	Full	Full	Full	Full	Limited	Limited	Full
Query Language	SPARQL	SQL	SQL	SQL	MongoDB Query Language	ArangoDB Query Language	SPARQL

Table 3
Characteristics of Evaluated DBMS

performs compared to others, ArangoDB is part of the evaluation as well.

Table 3 provides an overview of characteristic properties these databases.

2.7. Installation

The evaluation is executed on an Apple MacBook Pro Retina, 15-inch, Late 2013, 2.6 GHz Intel Core i7 with 16 GB memory and 1 TB flash storage.

For the target platform macOS, most DBMS could be installed using the package manager Homebrew. This included PostgreSQL, MongoDB, and ArangoDB.

There is also a current version of Virtuoso available via Homebrew, however, after installation, it cannot be launched due to segmentation faults. This could only be solved by manually downloading and compiling a previous version.

Both SQLite-approaches are retrieved and managed by Maven. Fuseki was manually downloaded[4] and extracted.

2.8. Setup

All databases are configured for an environment with 16 GB memory. Java-based databases (SQLite-Xerial, sqlite4java, and Fuseki) are given 16 GB heap space[5].

Virtuoso requires the configuration of each individual data import folder using the `DirsAllowed` setting in the main configuration file Virtuoso.ini. This file was also used to configure a 16 GB memory environment[6]. It is important that the current working directory is `/usr/local/virtuoso-opensource/var/lib/virtuoso/db` when launching Virtuoso. Otherwise, the database does not start, and prints no error message, even though its start-command is available system-wide. Parameters must be passed using `+`, e.g. `virtuoso-t +foreground`, and not using the standard Unix convention of minus (`-`)[12, p. 13].

MongoDB requires the one-time, manual creation of the folder `/data/db`. To automate the database creation on the fly, Fuseki is started with parameters to automatically create a plain database. Analogously, ArangoDB is instructed to create an empty database after launch. PostgreSQL is configured using the tool pgtune to specify 16 GB system memory.

The detailed startup and initialisation scripts can be found online[7]. Besides heap space and a database file name, neither SQLite-based approaches are configured at all.

2.9. Persistence of certain Fields

Some fields, such as `RDF_TYPE` or `DCTERM_SUBJECTS`, may contain multiple values for certain entities. In relational DBMS approaches, multiple values are embedded within the tables, using the individual capabilities that the database offers.

In PostgreSQL, multiple values are stored as an array type [17]. To implement the `GRAPH_LIKE`-query scenarios, there are a number of options. Using native ar-

[4] `https://jena.apache.org/download/index.cgi`, last access on 2015-11-22.

[5] Using the usual `-xmx16GB` Java Virtual Machine (JVM) parameter.

[6] Virtuoso.ini already contains configurations for different memory setups, the suggested 16 GB includes `NumberOfBuffers = 1360000` and `MaxDirtyBuffers = 1000000`

[7] `https://github.com/heussd/loddwhbench/tree/master/src/main/resources/shell`, last access on 2017-01-11.

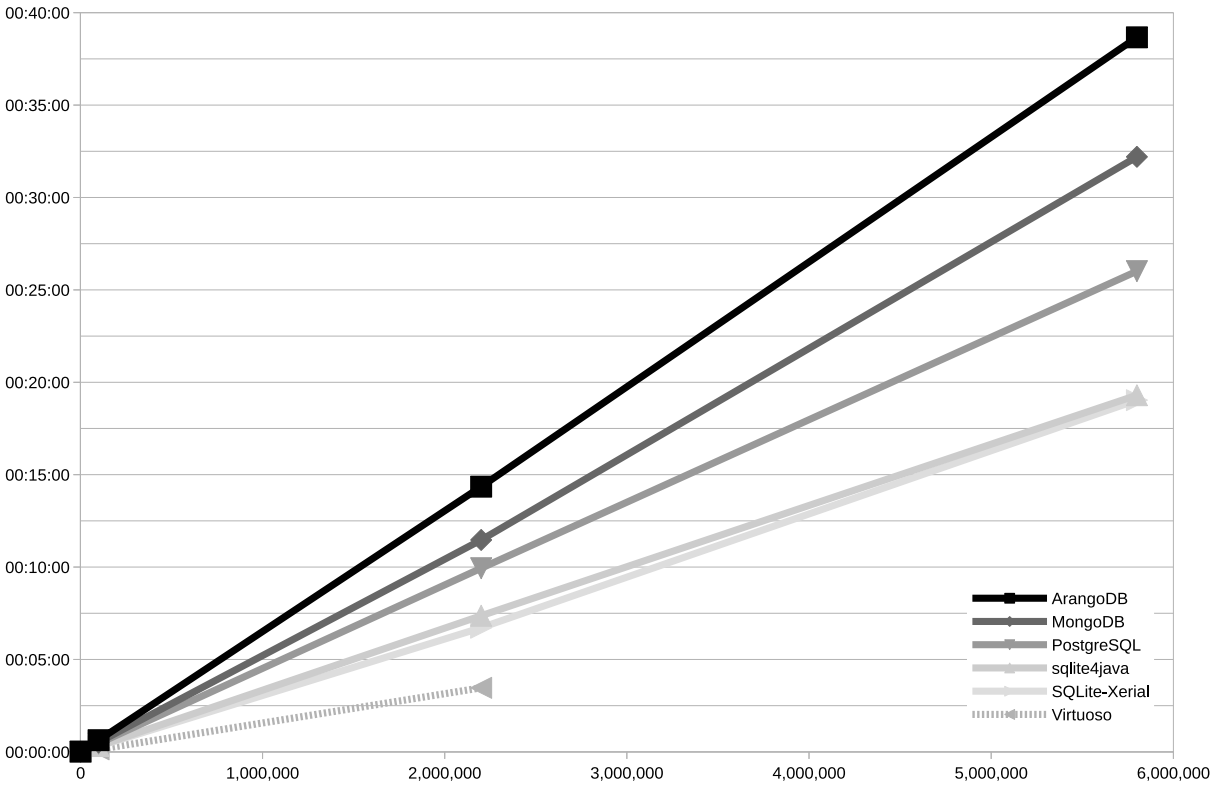


Figure 1. Loading throughput: number of loaded items versus time consumption

ray operators [1, Section 9.17. Array Functions and Operators] such as contains (<>) or `unnest` turned out to be slower than using dedicated materialised views. So, additional views are created exclusively for `GRAPH_LIKE`-query scenarios. The time required to create and index this view is measured in the prepare phase of the first `GRAPH_LIKE`-query scenario.

Neither of the two SQLite implementations provide support for non-trivial, embedded data types. For both, the Java tooling creates JavaScript Object Notation (JSON) structures to store multiple field values. For the `GRAPH_LIKE`-query scenarios, dedicated tables is are produced during the prepare phase.

3. Results

The measurement results of all database implementations are depicted in the following, starting with Figure 1 on this page. Full results[8], all scripts, queries and program source code is published in the GitHub repository loddwhbench[9].

In some cases, errors occurred during load or query execution. These cases are represented by missing values. Full error logs are published online with the results[8].

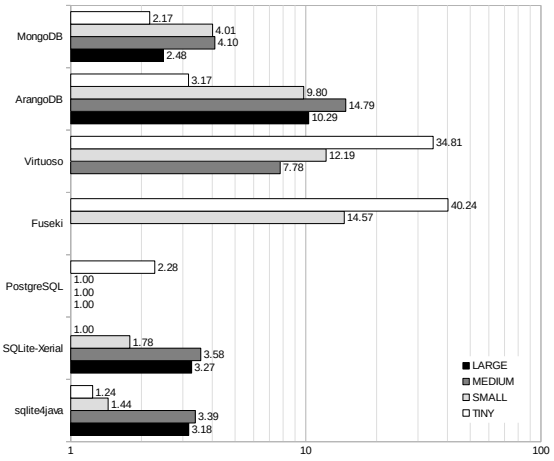
The performance score compared in the Figures 2, 3, and 4 on the following pages is calculated as the follows:

$$performance(database, queryscenario, testseries) = (prepare + execution1[+execution2 + execution3])/3$$

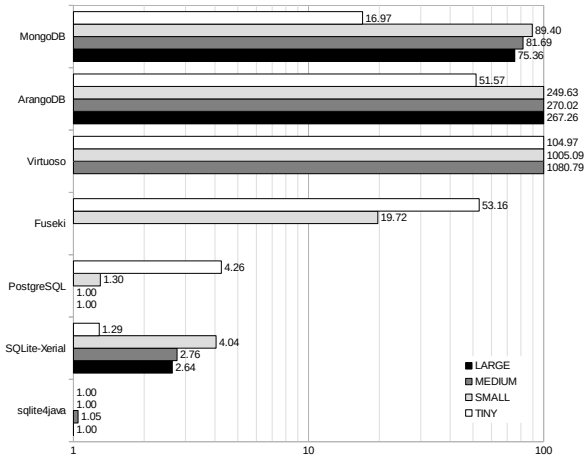
Moreover, each performance measurement is normed by the best performer in the given $(database, query\ scenario, testseries)$ scenario. This way, scores are independent from the exact environment and can be compared more easily (a score of 1.0 is always the best performer). The online results[8] contain the raw performance measures.

[8] `https://github.com/heussd/loddwhbench/tree/master/results`, last access on 2016-02-15.

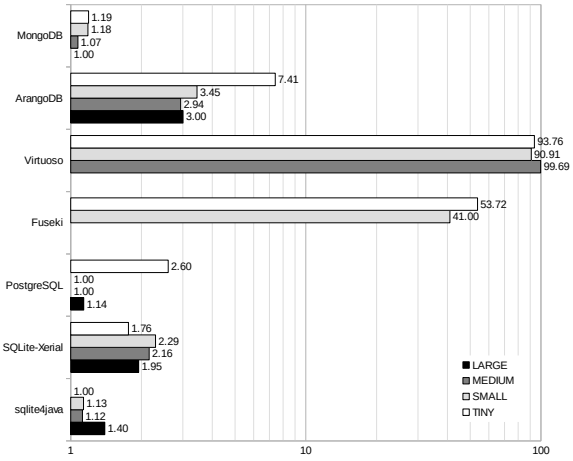
[9] `https://github.com/heussd/loddwhbench`, last access on 2016-02-15.



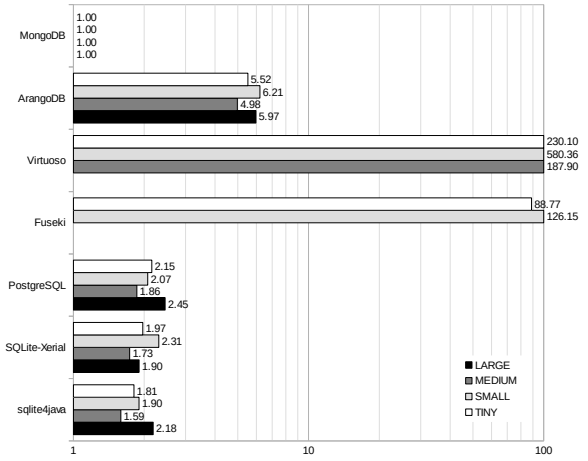
(a) ENTITY_RETRIEVAL_BY_ID_ONE_ENTITY



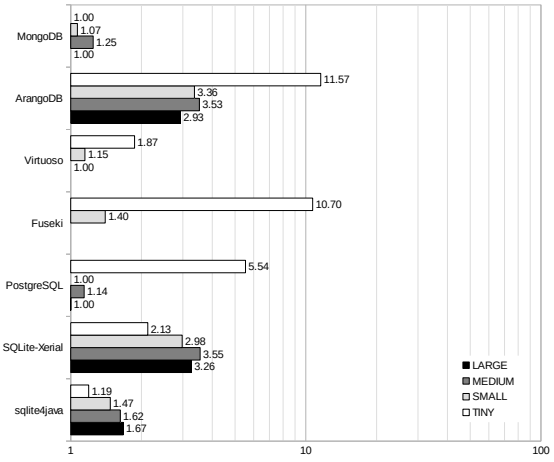
(b) ENTITY_RETRIEVAL_BY_ID_100_ENTITIES



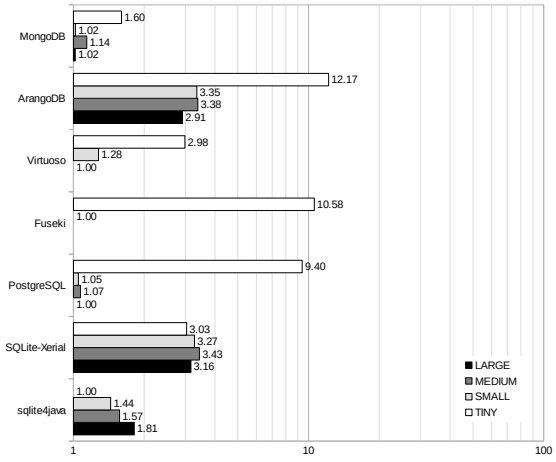
(c) CONDITIONAL_TABLE_SCAN_ALL_STUDIES



(d) CONDITIONAL_TABLE_SCAN_ALL_BIBLIOGRAPHIC_RESOURCES

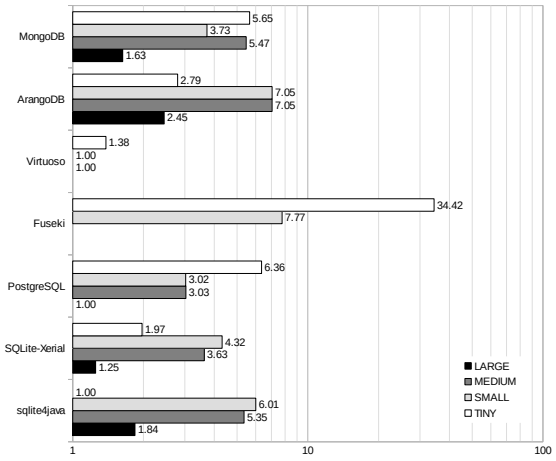


(e) AGGREGATE_ISSUES_PER_DECADE_TOP100

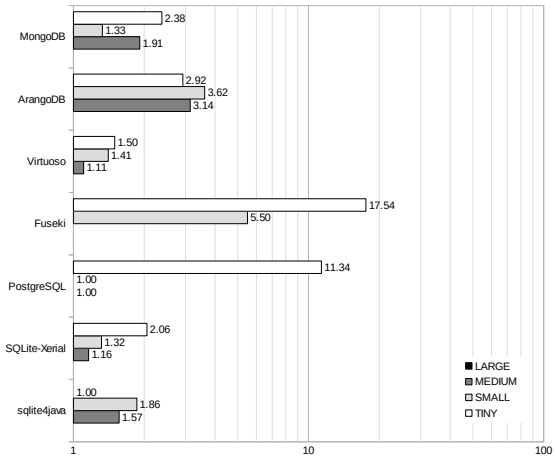


(f) AGGREGATE_ISSUES_PER_DECADE_ALL

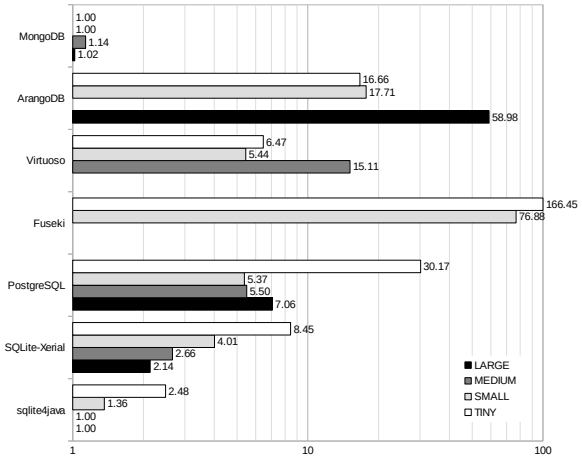
Figure 2. Query performances by scenario, database and test series, each normed by the best performer (1.0 is the best score archived).



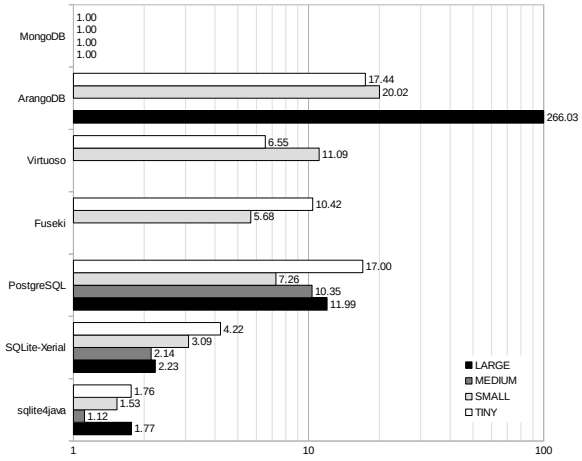
(a) AGGREGATE_PUBLICATIONS_PER_PUBLISHER_TOP10



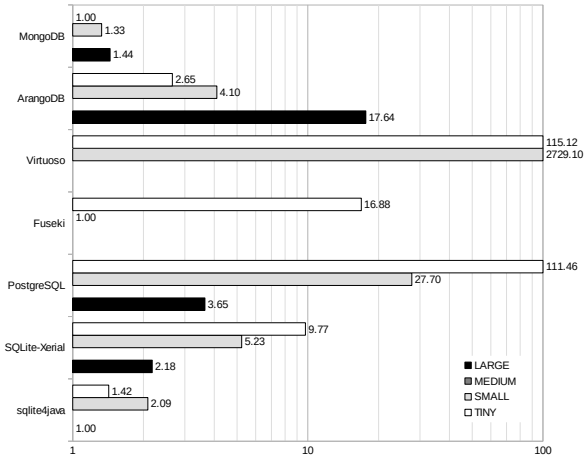
(b) AGGREGATE_PUBLICATIONS_PER_PUBLISHER_ALL



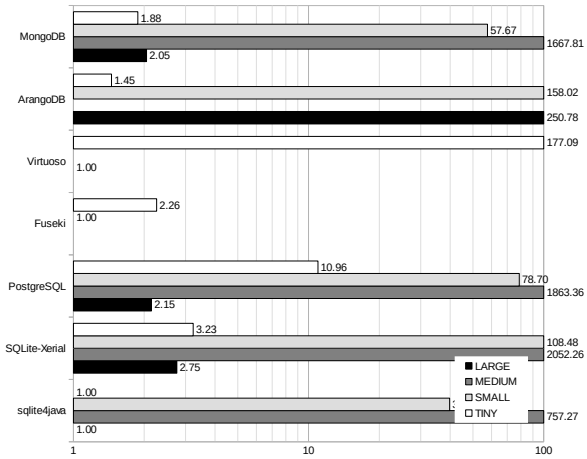
(c) UPDATE_HIGH_SELECTIVITY_NON_ISSUED



(d) UPDATE_LOW_SELECTIVITY_PAPER_MEDIUM

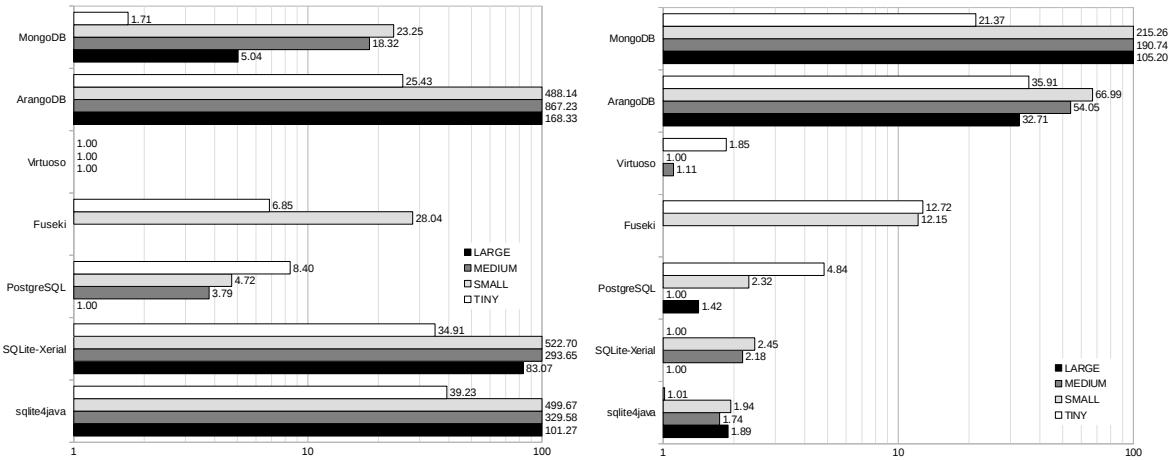


(e) DELETE_HIGH_SELECTIVIY_NON_ISSUED



(f) DELETE_LOW_SELECTIVITY_PAPER_MEDIUM

Figure 3. Query performances by scenario, database and test series, each normed by the best performer (1.0 is the best score archived).



(a) `GRAPH_LIKE_RELATED_BY_DCTERMS_SUBJECTS_1HOP_ONE_ENTITY` (b) `GRAPH_LIKE_RELATED_BY_DCTERMS_SUBJECTS_1HOP_10_ENTITIES`

Figure 4. Query performances by scenario, database and test series, each normed by the best performer (1.0 is the best score archived).

4. Observations

The following observations are based on the conducted query performance comparison of sqlite4java, SQLite-Xerial, PostgreSQL, Apache Jena Fuseki, Virtuoso, MongoDB, and ArangoDB.

4.1. Stability of Operation

A first observation is the fact that errors occurred on all DBMSs except the relational approaches, reproducible for all kinds of dataset sizes.

At very low data sizes such as 100,000 entries, Virtuoso, ran into an error during query execution[10]. With growing data sizes, the number of execution problems increases as well. With 1,7 million records, Virtuoso[11] and ArangoDB[12] show occasional but repeatable query execution issues.

Despite the issue in MEDIUM, ArangoDB loads the largest test series LARGE, consisting of 5.7 million records, flawlessly. A single issue can be observed for MongoDB[13]. Fuseki times out during load[14]. Syntax errors in the LARGE data source prevent Virtuoso from loading it entirely[15].

4.2. Loading Times

With all approaches, loading of 1,000 (TINY) or 100,000 entries (SMALL) only requires a few milliseconds or seconds. With larger dataset sizes, Virtuoso in particular can make use of its inherent advantage of loading data dumps directly: it is twice as fast as the fastest non-native RDF store. However, there is a critical weakness of this loading method: even small syntax errors can invalidate the entire loading process. This happens in case of the LARGE test series: One of the two datasets involved has a syntax error, which causes Virtuoso to fail when loading it[16]. The other triple store, Fuseki, starts failing with MEDIUM data sizes because of timeouts[14]. Even though very large timeout thresholds are configured, they don't seem to be sufficient.

[10] `https://github.com/heussd/loddwhbench/blob/master/results/Errors_Virtuoso_SMALL.txt`, last access on 2017-01-18.

[11] `https://github.com/heussd/loddwhbench/blob/master/results/Errors_Virtuoso_MEDIUM.txt`, last access on 2017-01-18.

[12] `https://github.com/heussd/loddwhbench/blob/master/results/Errors_Arango_MEDIUM_Since_SCHEMA_CHANGE_MIGRATE_RDF_TYPE.txt`, last access on 2017-01-18.

[13] `https://github.com/heussd/loddwhbench/blob/master/results/Errors_MongoDB_LARGE_Aggregation.txt`, last access on 2017-01-18.

[14] `https://github.com/heussd/loddwhbench/blob/master/results/Errors_Fuseki_Load_of_LARGE_and_MEDIUM.txt`, last access on 2017-01-18.

[15] `https://github.com/heussd/loddwhbench/blob/master/results/Errors_Virtuoso_LARGE.txt`, last access on 2017-01-18.

[16] `https://github.com/heussd/loddwhbench/blob/master/results/Errors_Virtuoso_LARGE.txt`, last access on 2017-01-18.

Given the fact that MongoDB does not provide transactional security, it seems surprising that it is not one of the fastest data loaders. ArangoDB is the slowest loader, requiring about 19% more time in loading the LARGE test series than the second slowest loader MongoDB.

The relational approaches, with SQLite implementations leading the way, seem to provide the best ratio between load throughput and load stability.

4.3. Entity Retrieval

Entity retrieval scenarios evaluate how fast a storage system can retrieve complete logical entities that are relevant to the given specific application scenario. Consequently, for many application specific scenarios, entity retrieval might be one of the most common query patterns. In case of the HeBIS catalogue, such an entity retrieval involves all data associated with a given bibliographic resource (as shown in Table 1).

In this evaluation, the query scenarios prefixed with ENTITY_RETRIEVAL and CONDITIONAL_TABLE_SCAN require the creation of complete, logical entities. Different storage approaches provide different means for entity retrieval, depending on their storage philosophy.

Relational approaches already store requested entities. The construction of entities is part of the integration of data. Triple stores, in contrast, have to constitute entities on the fly, as they just store atomic triples. Thereby, usually, the SPARQL command DESCRIBE can be used to retrieve all triples associated to a given set of RDF subjects (which then constitute entities) [18].

This works flawlessly with Fuseki, allowing the Java parser to sequentially read and create entities. With Virtuoso, however, DESCRIBE-queries returned all triples of all involved subjects in an arbitrary order. This might be due to known a bug in Virtuoso [9]. Due to this, it is impossible to read records sequentially, and thus efficiently from a seemingly-random ordered set of triples of multiple records.

To overcome this issue, a SELECT statement with a predefined column format was used, shown in Listing 1. In cases where the fields might have multiple values, a comma-separated embedded serialisation format is produced, using `group_concat`. This serialisation format then needs to be merged with the Java logic.

In the comparison it can be observed that relational approaches outperform all other approaches by at least a factor of two.

Apparently, both the on-the-fly-constitution of entities (required to be done by triple stores) as well as the DESCRIBE-query-workaround (specific to Virtuoso) seems to impact entity retrieval efforts significantly: compared to the relational results, the query times differ by one order of magnitude for Fuseki, and Virtuoso even doubles that. With growing dataset sizes, this effect seems to be amplified on all non-relational storage technologies.

```
select
 ?s
 (group_concat(distinct ?edition ; separator = ",")
       as ?edition)
 ?oclcum
 ?format
 ?identifier
 ?issued
 ?medium
 ?publisher
 (group_concat(distinct ?subject ; separator = ",")
       as ?subject)
 (group_concat(distinct ?title ; separator = ",")
      as ?title)
 (group_concat(distinct ?contributor ; separator =
      ",") as ?contributor)
 (group_concat(distinct ?P1004 ; separator = ",")
      as ?P1004)
 ?P1006
 (group_concat(distinct ?P1008 ; separator = ",")
      as ?P1008)
 (group_concat(distinct ?P1016 ; separator = ",")
      as ?P1016)
 ?P1017
 ?P1018
 ?sameAs
 (group_concat(distinct ?type ; separator = ",") as
       ?type)
 ?describedby
```

Listing 1 Select clause of a SPARQL statement to workaround a possible DESCRIBE bug in Virtuoso with a SELECT statement.

Both NoSQL approaches perform well on a small result sets, but perform significantly worse on larger ones.

4.4. Conditional Table Scan

Just like the entity retrieval scenarios, conditional table scan scenarios retrieve complete entities relevant to the given application context. In addition, instead of the very specific selection criteria (such as an entity ID), the tested scenarios match for about 2% in CONDITIONAL_TABLE_SCAN_ALL_STUDIES and about 92% CONDITIONAL_TABLE_SCAN_ALL_BIBLIOGRAPHIC_RESOURCES of the entities.

All relational approaches perform similarly, differing only by a few milliseconds. Like before, they display stable operation at good performances.

MongoDB performs close to relational approaches, but better. It can clearly outperform the best relational result in case of CONDITIONAL_TABLE_SCAN_ALL_BIBLIOGRAPHIC_RESOURCES. It seems that the more entities are involved, the relatively faster MongoDB can retrieve

them, though this observation could not be made in the entity retrieval scenarios earlier.

Regarding CONDITIONAL_TABLE_SCAN_ALL_STUDIES, ArangoDB improved also, compared to its previous entity retrieval performance.

Triple store approaches show bad performances, just as they do in entity retrieval scenarios. With Fuseki, the query time increases ten-fold compared to other approaches. Virtuoso completes all requested queries, though with worse performances. The performance differences with respect to the best competitor vary from two to three orders of magnitudes. Again, this might be due to the query workaround.

Roughly said, previous findings regarding entity retrieval performances can be confirmed for conditional table scan scenarios: Relational approaches perform just as well, NoSQL approaches perform better, triple stores worse.

4.5. Aggregation

In contrast to the complete record generation required in the entity retrieval scenarios, aggregation represents yet another class of scenarios where only certain fields of the stored data are relevant and processed using a group or sum operation.

This comparison contains scenarios: AGGREGATE_ISSUES_PER_DECADE aggregates values that must be produced with a substring operation on the fly, while AGGREGATE_ISSUES_PER_PUBLISHER

Apparently, the fact that Virtuoso already stores atomic field information instead of complete records has benefits in both aggregation scenarios: the triple store is usually the best aggregation performer.

Moreover, PostgreSQL displays a solid aggregation performance, independent from the fundamental test series and scenario. In contrast, query times in SQLite-based approaches vary: aggregation seems to be a principal weak point especially in sqlite4java, as well as for SQLite-Xerial when talking about the aggregation of on-the-fly-calculated values. When aggregating already stored field values, SQLite-Xerial performs nearly as well as PostgreSQL.

MongoDB can compete in the aggregation of calculated values, but shows poor performance in the aggregation of existing field values. This might be explained by a more sophisticated field indexing, causing MongoDB to be more efficient on string operations.

ArangoDB performs worst in all aggregation scenarios, taking about 2 to 5 times longer compared to the best performers.

4.6. Graph Scenarios

Graph scenarios represent a further class of queries which also does not require formation of complete records. Instead, they operate on an n:m relation between DCTERMS_IDENTIFIER and DCTERM_SUBJECT (see Table 1).

The simple graph scenario tested in this evaluation aim at relations between entities: How long does it take to retrieve the ids of related items of n base items, sharing field values?

A first observation is the fact that especially Virtuoso perform naturally well. In the tested graph scenarios, the triple store is multiple factors up to orders of magnitude faster than its competitors. Fuseki, in contrast, is surprisingly slow.

From all relational approaches, especially PostgreSQL is notable as it can compete with Fuseki. Moreover, all three relational approaches can compete if more base items are considered. All mentioned databases process the graph scenarios within milliseconds, almost independent from the size of the base dataset and the number of requested graph relations. The bad scores in Figure 4 can be explained with an expensive preparation phase.

MongoDB is not a natural graph store, and the fact that its a document store actually seems to hinder the structured, relation-based querying of data significantly. It can be observed that absolute query times grow almost linearly, based on the number of base test series.

ArangoDB, in contrast, is a graph store. The preparation phase explicitly structures the data in ArangoDB so that graph query processing is possible. Therefore, the displayed performance is surprisingly bad, compared to the good performances of Virtuoso and Virtuoso on the one hand and the non-performance of MongoDB on the other.

4.7. Change Operations

Besides read-only operations, certain write patterns are relevant for this evaluation. In the following, delete and update operations are compared. These operations can be distinguished depending on their selectivity: Highly selective operations provide a very specific preselection of affected data, so very few records are affected by the operation. Low selective operations accordingly provide a very unspecific data selection criteria and the operation affects large parts of stored information.

In this evaluation, based on the HeBIS catalogue, high selectivity operations affect about 2% of the records, low selectivity operations about 90%.

At first glance on the absolute numbers, it is noticeable that update operations are always more expensive than delete operations, and that low selectivity is always more expensive than high selectivity. These two rules of thumb can be observed across all different database approaches.

Even though most databases perform quite differently, MongoDB is one of the best in all these operations. This might be due to the fact that MongoDB is the only database in this evaluation that does not provide full support of atomicity, consistency, isolation, durability (ACID), but rather eventual consistency. This might also explain ArangoDBs good performance in the DELETE scenarios.

Relational approaches (PostgreSQL, sqlite4java, and SQLite-Xerial) form a mid-range of data manipulation performances, though there are noticeable differences when comparing sqlite4java with PostgreSQL, for example.

The performances of remaining approaches strongly vary, e.g. in the case of Fuseki, from the by-far-best performance in low selectivity delete operations, to errors in low selectivity update operations.

5. Conclusions

A comparison of query scenarios is executed on a number of current storage choices, involving relational DBMS, triple stores and two NoSQL approaches. Results are critical information for the design and the technological choice of a data warehouse.

Generally speaking, relational DBMS (PostgreSQL, sqlite4java and SQLite-Xerial) perform usually well in almost all scenarios, provide the most stable operation, and can load data of all compared sizes with adequate speeds and offer the best ratio of throughput and load-stability. They are the first choice whenever application-specific entities are involved in the queries, yet they can compete in scenarios where this is not the case - even in simple graph scenarios. They show best caching behaviours and display suitable performances for data manipulation. In detail, PostgreSQL, sqlite4java and SQLite-Xerial roughly perform analogously in TINY, SMALL and MEDIUM sizes, though there are significant differences in certain scenarios such as manipulation and aggregation.

For non-relational approaches, the performance strongly depends on details of the query scenario in specific technological architectures.

Triple stores can load native RDF most rapidly, as long as the data is 100% free of syntax errors. This assumption does not hold true for real world datasets: LODstats [8] for example shows that currently, about two thirds the datasets include errors[17].

Once loaded, especially Virtuoso performs well on aggregation scenarios, in graph scenarios, and in the certain deletion scenarios, though the performance is roughly comparable to the performance of relational DBMS in some cases. In scenarios with application-specific entities, triple stores are outperformed by three up to four orders of magnitudes, making them basically unsuitable for these scenarios.

With its document store approach, MongoDB performs well and partially best for the mass-retrieval of application-specific entities and in some data manipulation and some aggregation scenarios. In other scenarios, MongoDB lacks the flexibility to optimise data storage towards a scenario, which results in comparable bad performances.

ArangoDB's strength could not be discovered in this comparison. Even in scenarios for which the multi-approach database was specifically prepared, results are not as good as they are with competitors. In case of graph scenarios, for example, the performance is worse than one order of magnitude, compared to triple stores and relational DBMSs.

It is shown that various data storage philosophies perform quite different in different scenarios. While it is not suprising that some approaches might be ahead of others in certain disciplines, this work has shown that the gap can in fact be even wider and that even different implementations of the same paradigm showed noticable differences, e.g. in case of SQLiteXerial vs. sqlite4java vs. PostgreSQL or Virtuoso vs. Fuseki. Some approaches are practically unusable for certain scenarios.

This underlines the necessity of polyglot persistence architectures [16], the use of multiple database technologies in applications, based on application-specific use cases. Moreover, using architectures and frameworks built with a single fixed database implementation or persistence paradigm must considered to be questionable for applications with multiple use cases.

[17] http://lodstats.aksw.org/, last access on 2017-02-21.

6. Acknowledgements

This work was originally researched as part of a Ph. D. thesis[18] at the University of Plymouth and the University of Applied Sciences Darmstadt.

Thanks to Marco Hamm for helping in the development of the software in this paper. Thanks to Maria Lusky for reviewing the SPARQL queries.

Thanks to Uta Störl, Ralf Dotzert, Uwe Reh and Thomas Striffler for the valuable exchange.

References

[1] *PostgreSQL 9.1.18 Documentation*. http://www.postgresql.org/docs/9.1. Version: 2015. – Last access 2016-09-12.

[2] ALM WORKS: *almworks / sqlite4java — Bitbucket*. Version: 12 2015. https://bitbucket.org/almworks/sqlite4java, Abruf: 2015-12-07. – Last access 2015-12-07.

[3] In: AUER, Sören ; BÜHMANN, Lorenz ; DIRSCHL, Christian ; ERLING, Orri ; HAUSENBLAS, Michael ; ISELE, Robert ; LEHMANN, Jens ; MARTIN, Michael ; MENDES, Pablo N. ; NUFFELEN, Bert van ; STADLER, Claus ; TRAMP, Sebastian ; WILLIAMS, Hugh: *Managing the Life-Cycle of Linked Data with the LOD2 Stack*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012. – ISBN 978–3–642–35173–0, 1–16

[4] BIZER, Christian ; SCHULTZ, Andreas: *Berlin SPARQL Benchmark (BSBM)*. Version: 1 2015. http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/, Abruf: 2015-01-10. – Last access 2015-01-10.

[5] DINI, AG KIM Gruppe T.: Empfehlungen zur RDF-Repräsentation bibliografischer Daten. Version: 2014. http://edoc.hu-berlin.de/docviews/abstract.php?id=40697. Deutsche Initiative für Netzwerkinformation (DINI), 2014 (DINI-Schriften 14)

[6] EDLICH, Stefan: *NOSQL Databases*. Version: 12 2015. http://nosql-database.org/, Abruf: 2015-12-07. – Last access 2015-12-07.

[7] EDLICH, Stefan ; FRIEDLAND, Achim ; HAMPE, Jens ; BRAUER, Benjamin ; EDLICH, Stefan (Hrsg.): *NoSQL: Einstieg in die Welt nichtrelationaler Web-2.0-Datenbanken*. 2., aktualisierte und erw. Aufl. München : Hanser, 2011. http://dx.doi.org/10.3139/9783446428553. http://dx.doi.org/10.3139/9783446428553. – ISBN 978–3–446–42753–2 ; 3–446–42753–8

[8] ERMILOV, Ivan ; LEHMANN, Jens ; MARTIN, Michael ; AUER, Sören: LODStats: The Data Web Census Dataset. In: *Proceedings of 15th International Semantic Web Conference - Resources Track (ISWC'2016)*, 2016

[9] GITHUB VIRTUOSO ISSUE TRACKER: *SPARQL: ORDER BY not working with DESCRIBE queries · Issue #23 · openlink/virtuoso-opensource*. Version: 12 2015. https://github.com/openlink/virtuoso-opensource/issues/23, Abruf: 2015-12-21. – Last access 2015-12-21.

[10] GRAY, Jim: Database and Transaction Processing Performance Handbook. In: GRAY, Jim (Hrsg.): *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993. – ISBN 1–55860–292–5

[11] GRAY, Paul ; WATSON, Hugh J.: Present and Future Directions in Data Warehousing. In: *SIGMIS Database* 29 (1998), Juni, Nr. 3, 83–90. http://dx.doi.org/10.1145/313310.313345. – DOI 10.1145/313310.313345. – ISSN 0095–0033

[12] KERNIGHAN, Brian W. ; PIKE, Rob: *The UNIX Programming Environment*. Prentice-Hall, 1984. – 1–15 S. http://dx.doi.org/10.1002/spe.4380090102. http://dx.doi.org/10.1002/spe.4380090102. – ISBN 978–0–13–937681–8

[13] KURZ, T. ; SCHAFFERT, S. ; BURGER, T.: LMF: A Framework for Linked Media. In: *2011 Workshop on Multimedia on the Web (MMWeb)*, Institute of Electrical & Electronics Engineers (IEEE), September 2011, 16–20

[14] LIBRECAT: *LibreCat — Open Tools for Library and Research Services*. Version: 9 2015. http://librecat.org/, Abruf: 2015-09-29. – Last access 2015-09-29.

[15] LINNOVATE: *MEAN.io - MongoDB Express Angular Node.js*. Version: 2014. http://mean.io/. – Last access 2017-02-21.

[16] MARTIN FOWLER: *Polyglot Persistence*. https://martinfowler.com/bliki/PolyglotPersistence.html. – Last access 2017-02-21.

[17] POSTGRESQL DOCUMENTATION: *PostgreSQL: Documentation: 9.1: Arrays*. Version: 10 2015. http://www.postgresql.org/docs/9.1/static/arrays.html, Abruf: 2015-10-05. – Last access 2015-10-05.

[18] PRUD'HOMMEAUX, Eric ; SEABORNE, Andy: SPARQL Query Language for RDF / W3C. Version: 01 2008. http://www.w3.org/TR/rdf-sparql-query/, Abruf: 2015-10-02. 2008. – url. – Last access 2015-10-02.

[19] SCHMIDT, M. ; HORNUNG, T. ; LAUSEN, G. ; PINKEL, C.: SP^2Bench: A SPARQL Performance Benchmark. In: *IEEE 25th International Conference on Data Engineering, 2009. ICDE '09*, Institute of Electrical & Electronics Engineers (IEEE), März 2009, 222–233

[18] https://pearl.plymouth.ac.uk/handle/10026.1/8088, last access on 2017-02-21.