Efficient Parallel Processing of Analytical Queries on Linked Data

Stefan Hagedorn and Kai-Uwe Sattler

Ilmenau University of Technology, Ilmenau, Germany
first.last@tu-ilmenau.de

Abstract. Linked data has become one of the most successful movements of the Semantic Web community. RDF and SPARQL have been established as de-facto standards for representing and querying linked data and there exists quite a number of RDF stores and SPARQL engines that can be used to work with the data. However, for many types of queries on linked data these stores are not the best choice regarding query execution times. For example, users are interested in analytical tasks such as profiling or finding correlated entities in their datasets.

In this paper we argue that currently available RDF stores are not optimal for such scan-intensive tasks. In order to address this issue, we discuss query evaluation techniques for linked data exploiting the features of modern hardware architectures such as big memory and multi-core processors. Particularly, we describe parallelization techniques as part of our CameLOD system. Furthermore, we compare our system with the well-known linked data stores Virtuoso and RDF-3X by running different analytical queries on the DBpedia dataset and show that we can outperform these systems significantly.

Keywords: linked data, parallel query processing, micro benchmark.

1 Introduction

The concept of linked data refers to a set of best practices for publishing and connecting structured data on the Web. In practical use RDF and SPARQL have become de-facto standards for this based on a set of rules outlined by Berners-Lee [23]. With a fast growing popularity and a collection of almost 300 datasets containing more than 31 billion triples [36], the linked data movement can be seen as one of the most successful trends in the Semantic Web community.

Using SPARQL as the de-facto language for querying and processing linked data makes triple store-based engines or federated SPARQL engines a viable or even obvious choice. However, in this paper we argue that there are many tasks and use cases of linked data processing where plain SPARQL processors are not an ideal platform. Examples of such tasks are data profiling to determine (statistical) characteristics of a dataset or to detect links and correlations between triples or graphs. Such operations typically need to scan large portions of the datasets and, therefore benefit only marginally from traditional indexing

R. Meersman et al. (Eds.): OTM 2013, LNCS 8185, pp. 452–469, 2013.

and even prohibit query processing on distributed sources. An alternative way of handling such tasks on large datasets would be to leverage the MapReduce paradigm. However, this requires higher effort for programming compared to flexible ad-hoc query facilities of RDF databases – at least as long no automatic translation of declarative languages like Pig or Jaql is used. Furthermore, the nature of triple data requires to compute many joins which is a tedious task to implement in MapReduce.

In summary, there is a need for explicitly supporting analytical tasks on linked data in larger scale while providing the flexibility and expressiveness of a SPARQL engine. For an efficient support, query processing should exploit the features of modern hardware architectures such is big memory and multi-core processors.

In this paper, we try to address these challenges by discussing and evaluating techniques for efficient processing of analytical queries on linked data. We present these techniques as part of our CameLOD[1] system but focus on receipts and experiences derived from experimental evaluations on comparison with other approaches.

The remainder of this paper is structured as follows. In Section 2 we describe what use cases we think are important for analytical queries and what their requirements to a RDF query engine are. After a brief discussion of currently known RDF stores and SPARQL engines in Section 3, we describe how we store RDF data and process queries in CameLOD in Section 4. In Section 5 we show the results of experiments that we ran to compare CameLOD to other systems.

2 Use Cases

There are various use cases for analytical queries on linked data. In this section we introduce two of them and show on what operations their efficiency depends.

2.1 Data Profiling

A common analytical task on any dataset is to compute aggregations. Typical tasks include to compute the average value, the minimum or maximum value, or just to count the frequencies of certain values. For relational data this is done for certain columns. The traditional analytical processes require known semantics of the columns and also well integrated data. [6]

However, for linked data these preconditions are not given. The various sources of the datasets lead to different ways of how data and relations are expressed. Hence, analytical tasks on linked data focus on profiling the data in order to get a better understanding and overview of the contained information. This can be achieved in different ways. For example, the predicates (also referred to as *properties*) can help to determine the semantics of the dataset, but also outgoing links and data types are important.

[1] We pronounce it like the mystical British castle Camelot.

One example is to find statements with the same subject and object, but with different predicates. This can happen because of the many vocabularies that are available for the same topic. This makes it hard to formulate an appropriate query that does not miss any statement in its result set. A second example is to simply count the occurrences of the URIs in datasets to get an overview of the used vocabularies and the main content. In order to execute such profiling tasks, the query processing engine has to be very fast and efficient for scan and aggregation operations.

2.2 Correlations

By analyzing linked datasets, one might be interested in correlations within the data. Such correlations can be found in different aspects. First, entities might be correlated. This means, the dataset contains different entities that describe the same "thing". This might happen if different ontologies were used to create the dataset, e.g., when datasets from various sources are combined. Finding such entities is an active field of research of ontology matching and entity resolution [34] .

Additionally, correlations can be found within the data itself. This means, if the dataset contains information about events, finding correlations means to find events that are similar in any form. For example, two events can correlate spatially, by taking place in the same area, or they can correlate temporally, if they take place at the same time. In order to find such correlated evens efficiently, the data store has to provide dedicated indexes for spatial as well as for temporal data. For spatial data, a R-tree [18] has proven to be very efficient. For temporal data one could use e.g., an interval tree.

2.3 Requirements

By analyzing these use cases we can identify several requirements for efficient query processing techniques:

Highly Efficient Scans: Because scans on large portions of triple data are the core operations in the above scenarios, they have to be highly optimized. This includes to avoid expensive IO operations by exploiting in-memory processing, dictionary encoding of string values as well as further compression to reduce memory consumption and increase CPU cache utilization.

Ordered Data: Organizing data in an ordered fashion helps to increase the compression rate of data. Furthermore, using sort-based operators for aggregation/grouping and join computation avoids memory-intensive hashing strategies, allows to leverage caching effects, and to produce early results.

Special-Purpose Indexing: Efficient handling of spatial and/or temporal data requires dedicated index structures such as R-trees for spatial data or interval trees for temporal data. If a generic triple structure is used as the underlying storage scheme, these index structures should capture all relevant data, e.g., based on the predicates of the triples. Furthermore, scan and

lookup operations have to leverage these index structures for appropriate query predicates.

Parallel Processing: When using the general data organization scheme of triple stores, partitioning of data and parallelization of querying are fundamental techniques to utilize the processing power of today's hardware architectures and guarantee short response times even for big datasets. The partitioning should be flexible and allow a fine-grained work distribution among the CPU cores.

In the following sections we first discuss to which extent these techniques are supported by existing approaches and then describe how we use them in CameLOD for processing analytical tasks efficiently.

3 Related Work

Since RDF and SPARQL have become widely used to represent and query linked data, more and more storage systems and query engines have been published. In the following section we briefly describe some of them and also introduce some benchmarks created to compare them.

RDF stores and SPARQL engines. While RDF expresses information in triples, the need for a storage system that is optimized for the characteristics of such data arose. This lead to different implementations of triple stores that try to allow fast query execution on the often very large datasets.

One often discussed triple store is RDF-3X by Neumann and Weikum [29]. It stores all data in a centralized table and uses a dictionary to compress long literals to IDs. It further uses B^+-trees as its index structure. RDF-3X creates an index for every possible subject – predicate – object combination. Additionally, aggregate indexes are created in order to support partial queries. Furthermore, to speed up query execution it uses two kinds of statistics. The first one are histograms which are used for selectivity estimations. But as it assumes the predicates to be independent to each other, it cannot always be used. Therefore, the second statistics measure computes frequent join paths. These information are used during query optimization to find the optimal execution plan. In [30] Neumann and Weikum present enhancements to RDF-3X that allow fast updating, versioning, and transaction support.

Another triple store is Hexastore introduced by Weiss et al. in [38]. Like RDF-3X, Hexastore creates an index for each possible combination of S, P, and O and also the partial indexes. These indexes share the same payload, i.e., for a triple consisting of *(s, p, o)*, the PO and OP index both point the the same *s* for a given pair *(p,o)* and *(o,p)*, respectively. To compress data Hexastore also applies dictionary encoding.

In [19] Harris and Gibbins present 3store. 3store maintains a central triple table that stores the hashes of the S, P, and O component. A symbol table allows reverse lookups of the hash values. 3store supports SQL and also SPARQL.

To process SPARQL queries, the table is joined with itself for each triple in a graph pattern [32]. 4store [20] is a storage system and query engine for RDF and is designed to run on a cluster of nodes in a shared nothing architecture.

In [39] Wylog et al. propose dipLODocus. It uses a hybrid storage model, that stores RDF data as a graph but also as lists of literal values. Although this hybrid model causes single inserts and simple lookup queries to be less efficient than in other systems, the authors argue that it speeds-up complex analytical queries.

Virtuoso [11] is a hybrid database server supporting in memory processing and row- and column-wise storage. To support RDF, Virtuoso imports the triples into one big Quad-Table, that stores subject, predicate, and object and a graph identifier. The commercial version also supports spatial indexing. Virtuoso also provides a SPARQL endpoint, which is used very often in production[2]. Other well-known relational database systems were extended to support RDF, too. For example, DB2 [7] and Oracle [31] both provide object-relational mapping features to support RDF. The D2R Server from the D2RQ project [4] can be used to provide access to relational data via SPARQL.

Apache Jena[3] is a Java framework to build semantic web applications by providing an API to process RDF. It can be combined with various third party projects that actually store the data. The Jena project already includes two stores. On the one hand there is the SDB project which can be used to store RDF data in relational SQL-based databases. On the other hand, the TDB project is a high performance storage system, that creates indexes for any combination of subject, predicate, and object. The Fuseki[4] server provides a SPARQL endpoint for Jena. To process incoming queries Jena uses ARQ, which supports SPARQL 1.1.

With RDF HDT [13] Fernández et al. present a binary representation model for RDF data that achieves very high compression rates. However, HDT was not designed as a query engine and does not support query operations but only a single graph pattern at a time.

Some systems also support spatial operations. The OGC has published the GeoSPARQL[5] standard for operations on geographic information. The Parliament triple store [2] already has a implementation of GeoSPARQL. In [22] the authors present Strabon, a RDF store that can handle GeoSPARQL and their own development, stSPARQL.

Parallel Query Execution. Parallelization of (relational) database queries dates back to the eighties [9]. Whereas earlier works focused on multiprocessor systems by proposing techniques for intra-operator parallelism, e.g., particularly for expensive join and sort operators [10,24,15], Graefe proposed in [14] the exchange operator as a more general model for parallelizing other operators. This operator model allows both intra-operator parallelism on partitioned datasets as well

[2] `http://www.w3.org/wiki/SparqlEndpoints`
[3] `http://jena.apache.org`
[4] `http://jena.apache.org/documentation/serving_data/`
[5] `http://www.opengeospatial.org/standards/geosparql`

as horizontal and vertical inter-operator parallelism. Our parallel_do operator described in Section 4.2 is inspired by this idea. More recent approaches aim at exploiting features of modern multicore systems. Examples are dedicated join algorithms [21,1]. The works described in [26,27] provide a detailed analysis of in-memory joins on modern hardware architectures.

In [12] the authors build a SPARQL compiler and optimizer tailored for their Bobox framework. In [16] the authors develop parallel SPARQL engine focusing on join algorithms. The LHD project [37] is a distributed SPARQL engine built for a parallel infrastructure. All these works show that parallel execution can speed up query execution significantly.

SPARQL Benchmarks. There is a number of benchmarks for SPARQL engines and linked data stores. One widely known and accepted benchmark is the Berlin SPARQL Benchmark [3] that was introduced by Bizer and Schultz. The topic of the benchmark is e-commerce where products are offered and purchased and reviewed. The authors run their benchmark among others on D2R server, sesame, and Virtuoso. Their results showed, that none of the tested systems is superior in all queries.

Another benchmark that uses real world data is DBpedia SPARQL benchmark introduced by Morsey et al. in [28]. The queries were selected from the log of queries posed to the official DBpedia SPARQL endpoint.

A third often used benchmark for linked data was proposed by Guo et al. in [17]. The Lehigh University Benchmark (LUBM) generates data in university domain and provides fourteen queries on the data. The goal of the benchmark is to compare knowledge base systems for OWL. This benchmark was extended in [25] to enhance inference and scalability testing.

In the database community the TPC (Transaction Processing Performance Council) benchmarks have become popular. They provide several benchmarks that aim to evaluate different aspects of database systems. For example, the TPC-H benchmark provides ad-hoc queries and concurrent data modifications that simulate real life usage of a database [35]. The TPC benchmarks make heavy use of aggregates and grouping. Then, there is also SP^2B [33] which is settled in the DBLP scenario.

To the best of our knowledge, currently there is only one benchmark that concentrates on analytical query processing on linked data. In [8] Demartini et al. present BowlognaBench. The benchmark provides queries that analyze the content of the generated dataset, e.g., to find a student with the highest grades. However, we think that an analytical benchmark should include queries that generate more workload on the tested stores than the currently provided queries do.

4 Query Processing Techniques in CameLOD

In this section, we present several techniques for query processing to address the requirements described above. We start by giving an overview on the architecture of our CameLOD system followed by a detailed discussion of the query processing techniques.

4.1 System Architecture

The idea of our CameLOD system is to build a membase-like system for LOD which fetches data from sources, materializes them in an in-memory database and evaluates queries (including analytical tasks) on this database only. Sources are explicitly registered by users – CameLOD acts only as a caching systems where primary data sources are checked regularly or on-demand for updated data sets which then are downloaded and (re-)inserted.

In CameLOD triples are stored in chunks which are memory-resident blocks in a linked list. Using a linked list of chunks instead of a single big chunk very similar to index sequential file organization simplifies the inserting of triples but represents a trade-off with scan performance. Each chunk itself is organized into columns where a column is a simple array of integer codes matching the L2 cache size. We employ a dictionary encoding scheme for mapping all string values to integer codes; numeric values are encoded as integers, too. The dictionary encoding is order-preserving – thus, we store triples sorted on a given index column and the list of chunks represents a sorted list of all triples. Furthermore, each chunk stores also the minimum and maximum value of the index column in this chunk. Finally, for each column we maintain a pointer to the dictionary used for encoding the values. This allows to have separate dictionaries for the different triple components as well as for different graphs.

CameLOD follows the index-everything approach by using indexes on the subject, predicate, and object components of the triples resulting in materializing each triple three times in different sort order. These indexes are hash-based in-memory indexes mapping dictionary codes to chunks. Note, that a chunk may contain triples for multiple index entries. In addition, spatial and temporal indexes are built for triples with spatial or temporal values. However, in these indexes only the corresponding triples are stored.

In addition to these exact indexes (i.e., each triple value can be found in the index) a further min-value index is maintained. This index is an additional list of chunks where for each triple chunk an entry is stored which consists of a pair of minimal code value of the indexed column and the chunk address. This index is used to support partitioned scans as described below. For this purpose, each triple chunk maintains also a pointer to its min-value chunk. An overview of the storage organization in CameLOD is given in Figure 1.

The query engine is a Basic Graph Pattern (BGP) evaluator consisting of a set of physical algebra operators such as scan, filter, projection, sort, aggregation, several join implementations as well as special-purpose scans such as spatial and temporal index scans. Although there exists a SPARQL frontend based on the ARQ parser, in this paper we focus on the algebra level provided by a simple algebra language. We use the following notation where $\$i$, $\$k$ denote intermediate results in the form of a tuple stream which is produced by query operators and is also used as input for other query operators:

$$\$i := operator(\$k, params)$$

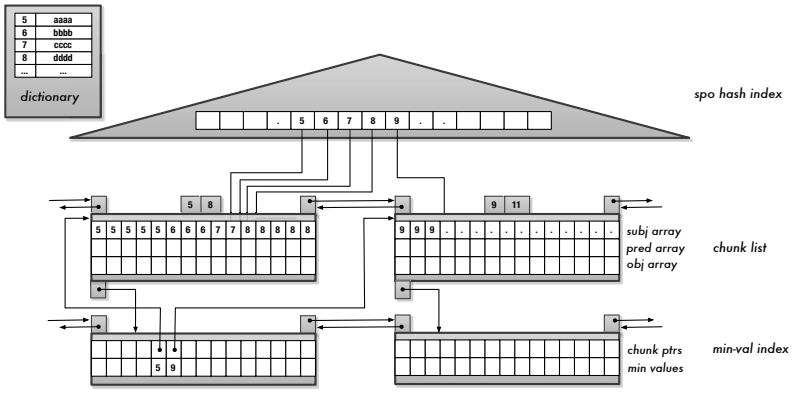


Fig. 1. Storage organization in CameLOD

Furthermore, parameters like `_0`, `_1`, ... denote the first, second, etc. component of a tuple, i.e., for a RDF triple they refer to the subject, predicate, and object component.

Query operators work always on chunks by resembling the vectorized processing model [40], i.e., instead of a one-tuple-at-a-time strategy in the open-next-close cycle, an operator consumes a complete chunk and produces also a chunk as the result of the `next` call. In order to avoid unnecessary memory allocation and copying, operators which do not create new data but restrict existing data (e.g., scan, filter) do not allocate new chunks. This is achieved by using a bitstring per result chunk indicating by the corresponding bit position whether the triple is still in the result or not. Operators such as join, sort, and aggregation create new chunks with the appropriated number of columns. However, theses chunks are only temporary and are not indexed. Note, that operators do not decode the triple values: predicates etc. are translated by the query parser into predicates using integer codes.

4.2 Partition-Aware Scans

Exploiting the capabilities of modern multi-core architectures requires parallel processing of queries. Apart from increasing the throughput of a system by using inter-query parallelism, intra-query parallelization helps to reduce response times of single queries. However, modern architectures draw several challenges to query parallelization in order to overcome limiting factors of scalability:

- low overhead for synchronization (e.g., by using lock-free data structures) and thread creation,
- dealing with load imbalance in partitioning the work caused by data skew,
- keeping the cores busy for better CPU utilization.

For intra-query parallelization, CameLOD provides a special physical operator parallel_do(*qtree, n*) which distributes the work of the query tree *qtree* across *n* tasks. Here, the leaf node of a query tree is always an index scan operator. An scan operator accepts the index, an optional range specification, and an optional predicate as arguments. For example

```
$1 := scan(s-index, "http://example.org/Alice", "http://example.org/Paul",
      _1 == "http://xmlns.com/foaf/0.1/mbox");
```

uses the subject index to find triples with a subject in the range from Alice to Paul and a predicate with the value foaf:mbox. This operator returns all chunks containing such triples in increasing order of subjects and initializes the corresponding bitstring accordingly.

To partition the work, i.e., the scan region, with low synchronization overhead and to be able to adjust the query execution to the number of available cores, the parallel_do operator is processed in three steps:

1. The query tree qtree is cloned n−1 times such that each query tree instance can be assigned to its own CPU thread.
2. In the preprocessing phase the min-val index is consulted to assign each query tree instance its own partition in terms of a range of consecutive chunks. This is done by looking up the chunk for the lower range value and getting the corresponding min-val index chunk. By scanning the index chunks, all chunks up to the end of range are collected in a list which is then split by the number n of tasks.
3. Finally, the query is executed by running each query tree instance in a thread. In our implementation we use the Intel TBB library which provides a task abstraction over CPU threads.

The parallel_do operator is also responsible for collecting and merging the individual results from the different query tree instances. This is achieved by maintaining an internal queue of result chunks which are forwarded to the parent operator.

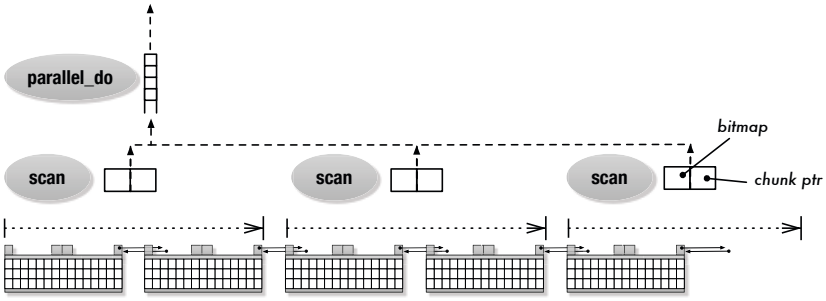


Fig. 2. Parallel scans

Figure 2 illustrates this for the following example query plan where the subject index is again scanned in the range from Alice to Paul in parallel by ten scan instances which also evaluate the given predicate:

```
$1 := scan(s-index, "http://example.org/Alice", "http://example.org/Paul",
        _1 == "http://xmlns.com/foaf/0.1/mbox");
$2 := parallel_do($1, 10);
printer($2);
```

Note, that this does not require any coordination or synchronization between the multiple query tree instances.

4.3 Query Operators

Beside index scans several standard query operators for evaluating SPARQL queries are implemented in CameLOD. This includes physical operators for the SPARQL algebra such as filter, project, sort (order by), distinct, union as well as several join implementations. CameLOD provides different implementations of these operators, e.g., there exist variants of sort, distinct, aggregation, and merge join which exploit the ordering of tuple stream stemming from the chunk list of a given index. In this way, we can avoid both building hash tables and sorting tuples.

In addition, the merge join uses further information from the chunks: because ordered chunks maintain the minimum and maximum values of the sorted column, we can quickly check whether two chunks overlap in their join columns and, therefore, we should join their tuples or simply skip one of the chunks.

A further special join implementation is the star join. This join operator is used to evaluate basic graph patterns like

$$\{\ \ ?\mathsf{s}\ \mathsf{pred}_1\ ?\mathsf{o}_1\ .\ ?\mathsf{s}\ \mathsf{pred}_2\ ?\mathsf{o}_2\ .\ \ldots\ .\ ?\mathsf{s}\ \mathsf{pred}_n\ ?\mathsf{o}_n\ \}$$

The star join performs a variant of an index join: for each subject value of the first binding the subject index is probed and the corresponding chunk is retrieved. Next, this chunk is scanned for all predicates $\mathsf{pred}_1 \ldots \mathsf{pred}_n$ of the given subject value which may be stored at the same chunk or one of the few following chunks. This is further supported by pushing down filter predicates to the chunk scans. The following query plan implements the query shown above:

$$\$1 := \mathsf{scan}(\text{s-index});$$
$$\$2 := \mathsf{star\text{-}join}(\$1, \text{s-index}, \_0, \_1, [\mathsf{pred}_1, \mathsf{pred}_2, \ldots, \mathsf{pred}_n]);$$

The parameters of the star join operator denote which index is used (subject index), which column of the input stream provides the lookup values (column 0, i.e., the subject), and which column of the right hand-side input is used for checking the predicate literals pred_i (in this case, the predicate column 1).

The star join operator provides performance benefits by improving cache utilization – not only while joining the various components of a given subject but also while scanning triples in subject order. Furthermore, the star join implementation is available in different forms. Beside the full join a left join variant is provided, too.

4.4 Work Stealing vs. Work Sharing

The parallel_do operator as the core building block for intra-query parallelization in CameLOD is not restricted to partitioned scans. In fact, any partitionable query tree can be parallelized using parallel_do – even operators such as sorting and aggregation at least partially.

However, among the above mentioned limiting factors to scalability load imbalance could become a serious problem. Load imbalance occurs if some query tree instances filter out a significant portion of data earlier and as a result the threads executing these query instances are left idle and have to wait for the busy threads to finish.

Work stealing is a popular scheduling strategy for MIMD computation. According to [5] the base idea dates back to the eighties. In the work stealing model, underutilized processors take the initiative if they need computational work: they try to steal threads from other processors. This results in less frequent thread migration, simply because when all processors are busy, no threads have to be migrated by the scheduler. This model is implemented for example in Intel's Threading Building Blocks (TBB) library[6]. The TBB library abstracts the physical threading model by allowing to treat units of work as *tasks* which are mapped to the underlying threads. In addition, TBB provides several parallel control structures, e.g., parallel loops where the iteration space is divided into multiple sub-spaces which are assigned to tasks.

We rely on this model by assigning query tree instances to tasks instead of threads and let the TBB scheduler do the work. This leads to the following possible strategies for query execution:

Data Flow Graphs: Each operator of a the query plan is implemented by its own task running independently from other operators. Though supported by dedicated TBB control structures (flow graphs), it requires communication between tasks and, therefore, does not seem to be a good idea for query execution in our context.

Work Stealing: Creating enough tasks in parallel_do which can be assigned to physical threads by the scheduler. Each task evaluates its own query tree instance without sharing data with other tasks.

Work Sharing: Similar to work stealing, but as soon as one operator in a query tree instance has finished its work, it tries to get work in terms of chunks from its corresponding neighbor operator in other query tree instances.

In CameLOD, work stealing is used for parallelizing scans as described in Section 4.2, because equal-sized partitions can be easily determined using the min-val indexes. However, if a query tree contains filter and join operators the individual partitions (i.e., range of chunks) may be reduced in their size in different ways resulting in load imbalance. Although it would be possible to redistribute the partitions again, this would require a synchronization among the tasks between two operators which we want to avoid.

[6] `http://threadingbuildingblocks.org`

To address this issue, the work sharing approach can be used particularly for more expensive operators such as joins. For this purpose, we have developed a cooperative variant of the index join which maintains two additional state variables: a role and a sibling list. The role describes whether the particular join instance acts as a giver (if not finished with its own partition) or an asker (otherwise), the sibling list contains the sibling operators, i.e., the corresponding join instances of the other tasks. As soon as a join instance changes its role to asker, it contacts one of its siblings from its sibling list and asks for taking over some work. The giver passes one or more chunks from the end of its partition which are not yet processed. The process stops when all join instances changed their state to asker.

In this approach, the following factors have a strong impact on the efficiency:

- the granularity of work shared between tasks,
- the overhead for registering givers and requesting work from them.

Obviously, there exist a tradeoff between the granularity of work and the overhead: finer granularities (e.g., triples) allow a better load balancing but lead to higher overhead. Because we use chunks as unit of processing in query operators, chunks are the natural granularity level for work sharing.

Askers and givers interact in the following way as part of the `next` function of an operator: each join task maintains a list of its siblings for checking their state as well as registration vector for exchanging work. As soon as a join task has finished the processing of its partition it asks one of the non-finished siblings from the vector by registering in its registration vector. If all tasks have already finished their work the processing stops. Furthermore, after processing a chunk each join tasks checks its registration vector for request to share work. In this case, a yet non-processed chunk is passed to the asker.

4.5 Spatial Indexing and Joins

Usally, B-trees or its variants are used to index column values. However, if the column does not just contain plain values, but rather values that describe an object in a higher domain, a more dedicated index can be used to make use of characteristics of these objects and to perform operations on them. Linked datasets often contain information about the location of events or the description of the shape of an entity. In order to efficiently use these information in query processing, a dedicated index structure is needed.

Statements that describe such spatial features of entities can be identified by the corresponding predicate. There exist various vocabularies that can be used to express such spatial relations. Currently CameLOD can identify spatial predicates from the GeoRSS[7] and W3C[8] ontology. When such predicates are recognized during data import, they will be used together with the corresponding object value to create a shape object. This shape will be indexed in a R-tree

[7] `http://www.georss.org/georss/`
[8] `http://www.w3.org/2003/01/geo/`

along with a triple identifier. The triple identifier points to the chunk on which the triple is stored and also contains the row offset within this chunk. Using this R-tree we are able to efficiently process queries that rely on some spatial relationship between objects. This allows to scan for all entities that for instance intersect with a given query region or belong to the k nearest neighbors of a given point. An example query, that scans for all entities that intersect with a given region is shown in the following:

```
$1 := scan(geo-index, intersects[region 46.999 15.355 47.109 15.481]);
printer($1);
```

To process this query CameLOD simply uses the given region to query its spatial index. CameLOD also implements a spatial join operator, which allows to join entities in the dataset regarding a spatial predicate, i.e., some spatial relation. We currently support four spatial join predicates, where a shape r is added to the join result of shape s, if

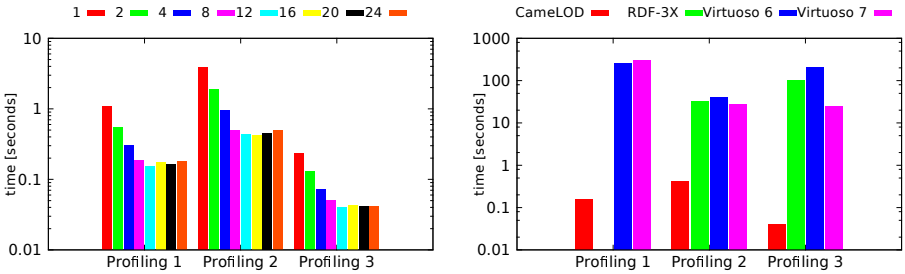
contains: s completely contains r,
location: r completely contains s,
intersects: r intersects with s, and
kNN: r belongs to the k nearest neighbors of s.

This spatial join operator is implemented as an index join. To compute the join result the operator gets the result of a previous operator (e.g., a scan) as input. For each triple in the input a lookup in the spatial index is performed. The type of this R-tree lookup depends on the join predicate. Because a R-tree only stores the minimum bounding rectangle (MBR) of shapes, the result of this index lookup may contain false positives. For example, the MBR of a circle intersects with another shape, but the circle itself does not. In order to prune these false positives, each shape of the index lookup has to be checked explicitly if it matches the join predicate. The pruned lookup result contains all join partners of the current triple which are now passed to the next operator in the plan and the next input triple can be processed.

5 Comparison and Evaluation

As we showed in Section 3, currently there is only the BowlognaBench benchmark that focuses on analytical queries. This benchmark contains queries that e.g., aim to find the students with the best grades or the professor supervising the most master theses. These queries are a good starting point for evaluating and comparing RDF stores. However, in our opinion to really show the strengths and weak points of such systems, queries that produce more workload are needed.

In Section 2 we showed some use cases that we believe are typical analytical tasks on linked data. In the following, we are going to show that the none of the existing systems provides adequate support for such tasks and we will further show how the query execution time can be reduced by utilizing parallel execution and special purpose indexes.



(a) Run times for different numbers of tasks (b) Comparison of different systems

Fig. 3. Run times for profiling queries

5.1 Setup

The tests were run on a server machine with 4 Intel Xeon E5-2630 CPUs with 6 cores each running at 2.30 GHz and 128GB DDR3 RAM. The machine has two 2TB S-ATA disks running in an RAID unit. As operating system we use Ubuntu 12.04 LTS with Kernel 3.2.0-40, 64 bit. We decided to work with the latest DBpedia 3.8[9] dataset. From this dataset we imported 215.000.000 triples into each store and executed different types of analytical queries, that are described in the following. The size of the raw dataset is 30 GB, while CameLOD only needs about 18 GB to store it.

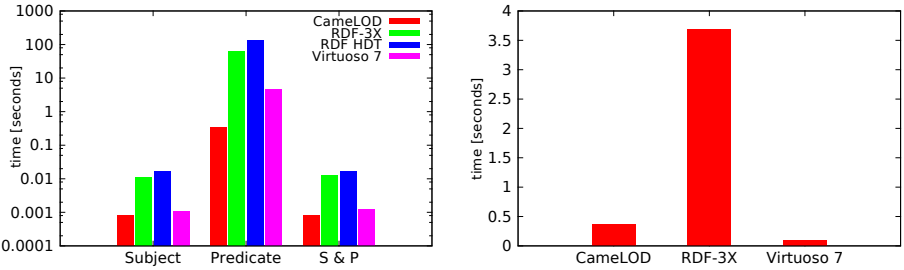
5.2 Experiments

First, we tested how parallelization of query plans impacts execution times. Since CameLOD is the only one of the considered systems that is able to execute the same query in parallel, we can only show the results for different numbers of tasks for CameLOD. We created the following three profiling queries:

Profiling 1. Count the number of subjects, that have more that 100 predicates
Profiling 2. Count the number of subjects, that are of at least two types[10]
Profiling 3. Get the 50 most often used predicates and their counts for subjects that start with a given prefix

As Figure 3a shows, the execution time decreases significantly with the increase of the number of tasks, but only until a certain point. If the number of tasks gets too high, no more benefit can be achieved and the execution times start to increase. This is due to the increasing overhead of data partitioning and for work sharing. The results of this experiment suggest that the optimal number of tasks for a parallel query is circa half the number of available cores. Further experiments will have to evaluate this issue.

[9] `http://wiki.dbpedia.org/Downloads38`
[10] where type means `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`



(a) Simple scans (b) Spatial scans

Fig. 4. Query execution times for scans

Next, we compare how different systems can handle data profiling queries. We will compare our CameLOD with RDF-3X 0.3.7, OpenLink Virtuoso 6.1.4 Open Source Edition, and Virtuoso 7 Commercial Edition for glibc 2.12. Although RDF-3X and Virtuoso 6 are disk-based, we chose them because Virtuoso is often used as a SPARQL endpoint for large datasets and RDF-3X is famous for its very fast query execution times and known to be one of the fastest engines currently available. We took the three profiling queries from the previous experiment and adapted them to each system. It showed, that RDF-3X is not capable of GROUP BY operations and implemented COUNT differently from the common sense. Thus, we can only show results for two of our three example queries for RDF-3X. The results of the query executions are shown in Figure 3b.

One can see that CameLOD is significantly faster than the other stores. These results can be achieved because CameLOD is an in-memory system and focuses on scan efficiency. RDF-3X is able to quickly process queries that rely on index lookups, but if the query needs to scan large parts of the dataset, the execution time increases. Virtuoso 6 is much slower than the other systems. For this reason we decided to exclude Virtuoso 6 from the rest of our experiments. Although Virtuoso 7 is an in-memory system, it still is not as fast as CameLOD in this scenario.

To compare the scan efficiency, we conducted a micro benchmark with our CameLOD, RDF-3X, RDF HDT, and Virtuoso 7. The queries we posed for this experiment are simple scans. The scan finds all entities with a given subject, the second one finds all entities with a given predicate, and the last query is a combination of these both queries, i.e., it contains a scan on the subject and another scan on the predicate. We chose the subject[11] and the predicate[12] that occur most frequently in the dataset so that the system has to scan large parts of the dataset. The result of this micro benchmark is shown in Figure 4a. HDT needs the longest time to complete the queries.

[11] `http://dbpedia.org/resource/Alphabetical_list_of_comuni_of_Italy`, 8498 times

[12] `http://dbpedia.org/ontology/wikiPageWikiLink`, $\sim$ 5 Million times

Their very high compression ratio makes it difficult to execute such scan operations as fast as the other systems. RDF-3X can use its B^+-tree index to easily find the needed entries. However, CameLOD is even faster than RDF-3X, because it can also use its ordered indexes to find the first chunk containing the needed entries, but does not need to load the values from disk. Virtuoso 7 is almost as fast as CameLOD in the cases where the subject is scanned. This may be because the optimizer reduces the intermediate result size using the subject and thus speeds up the join and retrieval of the result elements.

Analytical queries also have to filter for entities within ranges of complex types like location or time. To test the support for this kind of queries, a benchmark must provide appropriate queries. BowlognaBench contains queries that test the support for temporal operations, but it misses queries that operate on spatial data. Our spatial query scans for all entities that are located within a given region. This query region is given as a rectangle with the latitude/longitude coordinates of the lower left (29, -127) and upper right (49, -68) corner, which covers the continental area of the USA. The dataset contains ca. 1.3 million points. We executed the query 100 times each on a randomly chosen subset of these points. The results of the spatial queries are shown in Figure 4b. Using a R-tree as spatial index, CameLOD can quickly decide which sub-tree might contain objects that may belong to the query result. In contrast, RDF-3X does not have built-in support for spatial data. Hence, the query had to be expressed as shown in listing 1.1

```
SELECT ?s WHERE {
 ?s <http://w3.org/2003/01/geo/wgs84_pos#lat> ?lat .
 ?s <http://w3.org/2003/01/geo/wgs84_pos#long> ?lon .
 FILTER(?lat > 29 && ?lat < 49 && ?lon > -127 && ?lon < -68) }
```

Listing 1.1. Spatial Query rewritten for RDF-3X

Because RDF-3X cannot use a spatial index for this range query, it has to check every entity that has the latitude and longitude property if their values fall within the given query region. Fetching the entries from disk and the large amount of comparisons takes much longer than the index lookup that CameLOD has to execute. On the other hand, Virtuoso 7[13] is faster than CameLOD in this particular case. The query is not representative as real spatial analytical queries would use distance functions or polygons, which cannot be easily expressed in SPARQL.

6 Conclusion

In this paper we showed that existing RDF systems are not optimal for scan-intensive tasks or for tasks on complex types like spatial or temporal data. Especially analytical queries that require to scan large portions of a dataset can benefit from a scan-optimized and cache-aware engine, because often the dataset

[13] At the time of writing Virtuoso 7 was just released and therefore, we were not yet able to run this experiment with their spatial processing enabled. This will be part of our future work.

can be partitioned easily and then the multi-core architectures of modern hardware can be utilized efficiently.

We introduced our RDF storage engine CameLOD and compared it to the well known systems RDF-3X and Virtuoso. Thanks to our work stealing and work sharing approach, our scan-efficient implementation of the required indexes as well as our special purpose indexes for spatial and temporal data, we are able to execute scan-intensive queries much faster than the other systems. However, because there is no complete benchmark for analytical queries on linked data, we could only run queries that we think are representative for analytical use cases and we leave it to the linked data community to create a benchmark that covers all important features of efficient processing of analytical queries on linked data.

References

1. Albutiu, M.-C., Kemper, A., Neumann, T.: Massively parallel sort-merge joins in main memory multi-core database systems. PVLDB 5(10), 1064–1075 (2012)
2. Battle, R., Kolas, D.: Enabling the geospatial Semantic Web with Parliament and GeoSPARQL. Semantic Web 3(4), 355–370 (2012)
3. Bizer, C., Schultz, A.: Benchmarking the performance of storage systems that expose SPARQL endpoints. In: Proceedings of the ISWC (2008)
4. Bizer, C., Seaborne, A.: D2RQ - treating non-RDF databases as virtual RDF graphs. In: Proceedings of the ISWC (2004)
5. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM 46(5), 720–748 (1999)
6. Böhm, C., Naumann, F., Abedjan, Z., Fenz, D., Grütze, T., Hefenbrock, D., Pohl, M., Sonnabend, D.: Profiling linked open data with ProLOD. In: ICDEW, pp. 175–178. IEEE (2010)
7. Briggs, M., Sahoo, P.R., Raghavendra, G., Appavu, R., Anwar, F.: Resource description framework application development in DB2 10. Technical report (2013)
8. Demartini, G., Enchev, I., Wylot, M., Gapany, J., Cudré-Mauroux, P.: BowlognaBench - Benchmarking RDF Analytics. In: Aberer, K., Damiani, E., Dillon, T. (eds.) SIMPDA 2011. LNBIP, vol. 116, pp. 82–102. Springer, Heidelberg (2012)
9. DeWitt, D., Gray, J.: Parallel Database Systems: The Future of High Performance Database Systems. Communications of the ACM (1992)
10. DeWitt, D.J., Gerber, R.H.: Multiprocessor Hash-Based Join Algorithms. In: VLDB, pp. 151–164 (1985)
11. Erling, O., Mikhailov, I.: Rdf support in the virtuoso dbms. CSSW 221 (2007)
12. Falt, Z., Čermák, M., Dokulil, J., Zavoral, F.: Parallel SPARQL Query Processing Using Bobox. International Journal on Advances in Intelligent Systems 5, 1–13 (2012)
13. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF Representation for Publication and Exchange (HDT). Journal of Web Semantics (2013)
14. Graefe, G.: Encapsulation of Parallelism in the Volcano Query Processing System. In: SIGMOD Conference 1990, pp. 102–111 (1990)
15. Graefe, G.: Parallel Query Execution Algorithms. In: Encyclopedia of Database Systems, pp. 2030–2035 (2009)
16. Groppe, J., Groppe, S.: Parallelizing join computations of SPARQL queries for large semantic web databases. In: SAC, New York (2011)

17. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Semantic Web Journal 3(2-3), 158–182 (2005)
18. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: SIGMOD Conference, pp. 47–57 (1984)
19. Harris, S., Gibbins, N.: 3store: Efficient Bulk RDF Storage. In: PSSS (2003)
20. Harris, S., Lamb, N., Shadbolt, N.: 4store: The design and implementation of a clustered rdf store. In: SSWS (2009)
21. Kim, C., Sedlar, E., Chhugani, J., Kaldewey, T., Nguyen, A.D., Blas, A.D., Lee, V.W., Satish, N., Dubey, P.: Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. PVLDB 2(2), 1378–1389 (2009)
22. Kyzirakos, K., Karpathiotakis, M., Koubarakis, M.: Strabon: A Semantic Geospatial DBMS. In: Cudré-Mauroux, P., et al. (eds.) ISWC 2012, Part I. LNCS, vol. 7649, pp. 295–311. Springer, Heidelberg (2012)
23. Lee, T.B.: Linked Data - Design Issues (2009), `http://www.w3.org/DesignIssues/LinkedData.html` (last accessed: May 24, 2013)
24. Lu, H., Tan, K.-L., Shan, M.-C.: Hash-Based Join Algorithms for Multiprocessor Computers. In: VLDB, pp. 198–209 (1990)
25. Ma, L., Yang, Y., Qiu, Z., Xie, G.T., Pan, Y., Liu, S.: Towards a complete OWL ontology benchmark. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 125–139. Springer, Heidelberg (2006)
26. Manegold, S., Boncz, P.A., Kersten, M.L.: What happens during a join? dissecting cpu and memory optimization effects. In: VLDB, pp. 339–350 (2000)
27. Manegold, S., Boncz, P.A., Kersten, M.L.: Optimizing main-memory join on modern hardware. IEEE Trans. Knowl. Data Eng. 14(4), 709–730 (2002)
28. Morsey, M., Lehmann, J., Auer, S., Ngonga Ngomo, A.-C.: DBpedia SPARQL benchmark – performance assessment with real queries on real data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 454–469. Springer, Heidelberg (2011)
29. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. Proceedings of the VLDB Endowment 1(1), 647–659 (2008)
30. Neumann, T., Weikum, G.: x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases 3(1), 256–263 (2010)
31. Oracle. Semantic Technologies Developer's Guide. Technical report (2012)
32. Sakr, S., Al-Naymat, G.: Relational processing of RDF queries. ACM SIGMOD Record 38(4), 23 (2010)
33. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP2Bench: A SPARQL Performance Benchmark. CoRR (2008)
34. Shvaiko, P., Euzenat, J.: Ontology Matching: State of the Art and Future Challenges. TKDE 25(1), 158–176 (2013)
35. Transcation Processing Performance Council. TPC Benchmark H - Decision Support. Technical report (2013)
36. W3C. Linking Open Data (2013), `http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData` (last accessed: May 24, 2013)
37. Wang, X., Tiropanis, T., Davis, H.C.: LHD: optimising linked data query processing using parallelisation. In: Linked Data on the Web, LDOW 2013 (2013)
38. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. PVLDB 1(1), 1008–1019 (2008)
39. Wylot, M., Pont, J., Wisniewski, M., Cudré-Mauroux, P.: dipLODocus$_{[RDF]}$—Short and Long-Tail RDF Analytics for Massive Webs of Data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 778–793. Springer, Heidelberg (2011)
40. Zukowski, M., Boncz, P.A., Nes, N., Héman, S.: Monetdb/x100 - a dbms in the cpu cache. IEEE Data Eng. Bull. 28(2), 17–22 (2005)