Querying Semantic Knowledge Bases with SQL-on-Hadoop

Martin Przyjaciel-Zablocki
University of Freiburg
zablocki@informatik.uni-freiburg.de

Alexander Schätzle
University of Freiburg
schaetzle@informatik.uni-freiburg.de

Georg Lausen
University of Freiburg
lausen@informatik.uni-freiburg.de

ABSTRACT

The constant growth of semantically-annotated data and an increasing interest in cross-domain knowledge bases raises the need for expressive query languages for RDF and novel approaches that enable their evaluation for web-scale data sizes. However, SPARQL, the W3C standard query language for RDF, suffers from a rather limited capability to express navigational queries. More expressive languages have been theoretically studied, however not implemented. In this paper, we continue our work on TriAL-QL, an expressive (SQL-like) RDF query language based on the *Triple Algebra with Recursion* [31]. We present a new version of our TriAL-QL processor, which takes advantage of the current momentum in *in-memory* SQL-on-Hadoop solutions and is built on top of Impala and SPARK while using one unified data storage. We use our system to study the application of multiple evaluation algorithms, storage strategies and optimizations on Impala and SPARK while highlighting their properties. Comprehensive experiments examine the performance of our system in comparison to other competitive RDF management systems. The obtained results demonstrate its suitability for querying semantic knowledge bases by providing interactive query response times for selective queries on datasets with more than one billion triple. More data-intensive use-cases that produce, e.g. over 25 billion results finished in the order of minutes.

1. INTRODUCTION

In past decade, we have witnessed the evolution from a *"Web of Documents"* to a highly-interlinked *"Web of Data"*, in which so-far human-consumable information is given a well defined machine-processable meaning. Driven by the wide adoption of Semantic Web technologies and in particular the so-called RDF data model [32], data from various domains and in multiple languages is becoming more and more interconnected. This facilitates the emergence of *semantic knowledge bases* such as DBpedia [10], YAGO [26], Microsoft's Satori, and Google's Knowledge Vault [18]. However, querying such semantic knowledge bases, which have often a high degree of diversity in the structure and vocabulary, poses new challenges for query languages and their respective implementations [16]. Despite considerable work that has been done in this area, recent work [31, 8, 40] has proven that important properties in RDF data exist which cannot be captured by current RDF query languages including SPARQL 1.1 and its extensions. In order to exploit the real potential of such data, RDF query languages need (1) to capture *all invariants* inherent to the triple-based model of RDF and (2) to allow one to query RDF data along with its ontology and schema and (3) to enable querying path-based connections between arbitrary resources. Given the *graph-like* structure of highly-interconnected knowledge bases, we found expressive *navigational* queries to be well-suited to capture the aforementioned requirements, since they provide valuable information about the interlinking between arbitrary things. Thus, we proposed in [38, 39] TriAL-QL, an expressive (SQL-like) RDF query language based on the *Triple Algebra with Recursion* (TriAL*) [31]. In contrast to many other approaches TriAL* is a compositional algebra, where the output is again RDF data.

While the constant growth of semantically-annotated data and an increasing interest in cross-domain knowledge bases, justifies such expressive, navigational query languages, it raises also the need for novel approaches that enable their evaluation for large data sizes. Therefore, we started to investigate in [38, 39] the application of Hadoop, the de-facto standard platform for *Big Data*, to distribute the workload associated with the evaluation of our language on a cluster of machines. Apart from the widely deployed infrastructures, we see the main advantages of the Hadoop ecosystem in its continuous development which is reflected by novel frameworks and layers that are added continuously. Furthermore, we benefit, as we will also demonstrate in our work, from the concept of a common data storage by means of HDFS (also called *data lake*) that can be accessed by all applications built on top of Hadoop.

In previous work [39], we presented the first prototype of our TriAL-QL processor called TriAL-QL Engine which was based on Impala, a massive parallel SQL query engine. We described the process of compiling TriAL-QL queries to the SQL dialect of Impala and presented an evaluation algorithm for recursive expressions. In continuation of this work, we study in the current paper our new TriAL-QL Engine, which is implemented on top of *Impala* and *Spark* [51], a fast general-execution framework for large-scale data processing while sharing *one* unified data store in HDFS. We

BeyondMR'17 May 19, 2017, Chicago, IL, USA

ISBN 978-1-4503-5019-8/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3070607.3070610

investigate additional *evaluation algorithms* for the most important query patterns, discuss *data storage strategies* and study their application on Spark and Impala.

We can summarize the contributions of this paper as follows: We present the architecture of TriAL-QL Engine describe how it is built on top of Spark and Impala while sharing one unified data storage. The source code is published on GitHub[1] (cf. Section 4). We give a detailed description of our evaluation algorithms, in particular we study patterns going beyond the expressivity of SPARQL 1.1. We further investigate the application of our proposed algorithms and strategies on both, Impala and Spark while discussing their strengths and weaknesses. A comprehensive evaluation examines the performance and scaling properties of our system with respect to different execution strategies and in comparison to other competitive RDF management systems.

The structure of the paper is as follows: Section 2 discusses related work. Section 3 introduces the problem description in more detail and gives the necessary background of TriAL*. Section 4 describes the general architecture of our TriAL-QL Engine, followed by data storage strategies in Section 4.2 and evaluation strategies in Section 4.3 and 4.4. Finally, Section 5 presents a comprehensive evaluation in which our engine is compared with other competitive RDF management systems.

2. RELATED WORK

We can distinguish between three groups of RDF management systems: (1) centralized systems, (2) specialized distributed systems, and (3) distributed systems built on top of Big-Data frameworks.

Centralized systems operate on top of a single powerful machine, often equipped with specialized hardware components to enhance the performance. Common examples for such systems are *Sesame* [13], Virtuoso [19], *Jena* [15], *RDF-3X* [34] and *3store* [23]. In case of an increased amount of RDF data, resources such as processing power, main memory or hard disks can be improved. This strategy is known as *scale-up*. Although very powerful, such systems are limited in their scalability and are known to not be very cost efficient at larger scales.

In case of *specialized distributed systems*, the workload parallelization is implemented as part of the RDF management system rather than relying on an underlying distributed framework such as Hadoop [50]. Interesting approaches are *Virtuoso Cluster* [11], *TriAD* [21], *Dream* [22], *4store* [24], *YARS2* [25] and *Clustered TDB* [35]. Some of these are extensions of centralized systems, e.g. RDF-3X [27] or Jena [35], which have to be independently installed on each machine in the cluster. Scaling in such systems is achieved by adding further machines, which is denoted as *scale-out* strategy. The main drawbacks of these systems are (1) the need for a dedicated infrastructure that has to be maintained solely for the purpose of querying RDF, and (2) the fact that the initial graph partitioning used for spreading data across machines is often done in a centralized way, being the bottleneck for large-scale RDF data [30].

Furthermore, it is worth noting that, since TriAL* can be translated into *Datalog*, parallel Datalog engines [47, 14, 2, 45] are also interesting candidates for distributing the workload on a cluster of machines. A more extensive comparison and discussion with such engines is part of future work.

The last group of RDF management systems is the one built on top of existing *Big Data frameworks*, such as MapReduce [17] or SQL-on-Hadoop [48, 9] solutions. In recent years, the Hadoop ecosystem has become the de-facto standard for processing Big Data. Large infrastructures are deployed in research or industry and supported by major Cloud providers such as *Amazon Elastic Compute Cloud (EC2)*. Due to its robustness, reliability and scalability while being able to run on heterogeneous commodity hardware, Hadoop gained lot of attention in manifold application fields. Consequently, there have also been many different Semantic Web tasks implemented on top of Hadoop ranging from the evaluation of SPARQL queries [27, 28, 43, 44] to large-scale OWL reasoning [49].

However, to the best of our knowledge, not much work has been done on using Hadoop for the evaluation of expressive, navigational RDF query languages as TriAL-QL. At most, SPARQL 1.1 Property Paths are supported via *Virtuoso Cluster* as being the only available *distributed* RDF management system that runs on a cluster of machines [11]. Implementations of RDF query languages having a higher expressiveness than Property Paths can only be found in centralized systems, such as in extensions of Sesame and Jena, which lack the support for querying web-scale RDF data.

3. CHALLENGES OF RDF QUERYING: THE TRIAL APPROACH

Most navigational RDF query languages are derivatives from standard graph query languages like *nested regular expressions* (NRE) [37]. However, in contrast to the standard graph model, an edge label in RDF (*predicate*) does not come from a predefined alphabet and may also appear as a source or destination (*subject* and *object*, respectively) of another edge (triple). Consequently, they are not capable of certain constructs and lose important features, e.g. reasoning over predicates within a query [6]. To the best of our knowledge, there are only two RDF query languages that enable expressive navigational capabilities with reasoning and can be evaluated in combined (low-degree) polynomial time, namely *Triple Query Language Lite* (TriQ-Lite) [8] and *Triple Algebra with Recursion* (TriAL*) [31]. TriQ-Lite is defined as a general Datalog extension that captures SPARQL queries enriched with the OWL 2 QL profile, whereas TriAL* is a closed language, i.e. the output is a set of triples rather than mappings or bindings. Its core idea is to work directly with triples rather than transforming the RDF model into a graph-based model where, e.g., $\{(s,p,o),(p,s,o')\}$ would not be a valid graph [31]. These features, together with the descent of TriAL* from relational algebra, predestinate it as a basis for more expressive RDF querying on SQL-on-Hadoop solutions.

TriAL* takes the relational algebra as its basis with some restrictions to guarantee closure. The most important operator is a *triple join* between two ternary relations E_1 and E_2 representing sets of triples, defined as: $E_1 \bowtie^{i,j,k}_{\theta,\eta} E_2$, where $i,j,k \in \{s_1,p_1,o_1,s_2,p_2,o_2\}$ indicate the implicit projection on three fields to keep the operation closed with s_1 referring to the subject of E_1, etc. θ represents the join conditions whereas η is a set of conditions between objects and

[1]The TriAL-QL Engine was further developed in the course of a master thesis [12].

data values. To express paths of arbitrary length, recursion is added with the *right* $(e \bowtie^{i,j,k}_{\theta,\eta})^*$ and *left* $(\bowtie^{i,j,k}_{\theta,\eta} e)^*$ *Kleene closure*, where e is a TriAL* expression. We refer to [31] for a more detailed description of TriAL*.

Although TriAL* is a neat approach for querying RDF, its algebraic notation is not easy to write. Thus, we have introduced the TriAL* Query Language (TriAL-QL) in [39] and defined its mapping to the algebra of TriAL*. The basic idea behind TriAL-QL is to flatten the algebra expressions of TriAL* to a sequence of SQL-like statements, while preserving its expressiveness. We refer to [39], for a more detailed introduction of TriAL-QL. For the sake of brevity, we will follow the algebraic notation of TriAL* for the remainder of the paper.

4. TRIAL WITH SQL-ON-HADOOP

The TriAL-QL Engine is a distributed processor of our TriAL-QL query language. It is built on top of Hadoop, where we make use of the current momentum on scalable SQL-on-Hadoop frameworks. We published the code on GitHub[2]. An essential advantage of such frameworks is the possibility to use of SQL as an adequate intermediate layer. This is particularly beneficial for processing query languages such as TriAL-QL which are based on relational algebra, due to an intuitive mapping between both. Furthermore, we can (1) benefit from an inter-compatibility between different SQL-on-Hadoop solutions, (2) be independent from future Hadoop changes, and (3) take advantage of the continuously optimized Hadoop stack. However, besides the differences in syntax and expressiveness of the supported SQL dialect, there are crucial differences in how the respective SQL-on-Hadoop solutions process a given SQL query, revealing different characteristics for various query types. In line with this, we examine the applicability of a scalable processor for TriAL-QL on top of Hadoop using *Impala*, *Spark* and *Hive* and compare its properties. Please note that, although not included in the latest TriAL-QL Engine, we also consider Hive in the followed discussion and experiments[3]. The architecture of our implementation can be conceptually structured into three main components:

- **RDF Store:** The basis for distributed querying is an efficient yet unified data pool which is supported by all three of our systems: Spark, Hive and Impala. It maintains input RDF data and in the case of a disk-based execution also intermediate results and the final results of queries, which can be (if specified within a query) reused as input in a later query.

- **Query Compiler:** A Query Compiler composes components required to translate a given TriAL-QL query into the respective SQL dialect of the desired SQL-on-Hadoop system. Furthermore, it is also the place where certain query optimizations are applied. Most notable are two interesting query patterns, for which we describe optimized evaluation strategies, namely (1) the recursions captured by the left and right Kleene Closure of TriAL* and (2) the connectivity between two resources.

[2]http://github.com/martinpz

[3]The support for Hive was dropped. However, we could execute the compiled SQL queries on Hive which formed the basis for a few experiments shown later.

- **Query Processor:** The execution of SQL queries is done by the Query Processor, where multiple execution strategies are suggested which differ, for instance, in how queries are composed and when intermediate results become materialized to disk. Another important task that this component is taking care of are termination conditions. Based on which algorithms are chosen by the Query Compiler, different termination constraints need to be checked. Further, due to the differences in the support of recursions in the respective SQL-on-Hadoop systems, an individual query processor exists for each supported system.

The architecture of our TriAL-QL Engine, illustrated in Figure 1, reveals a much more granular view on all three components, including the technical integration with Impala, Spark, and HDFS.

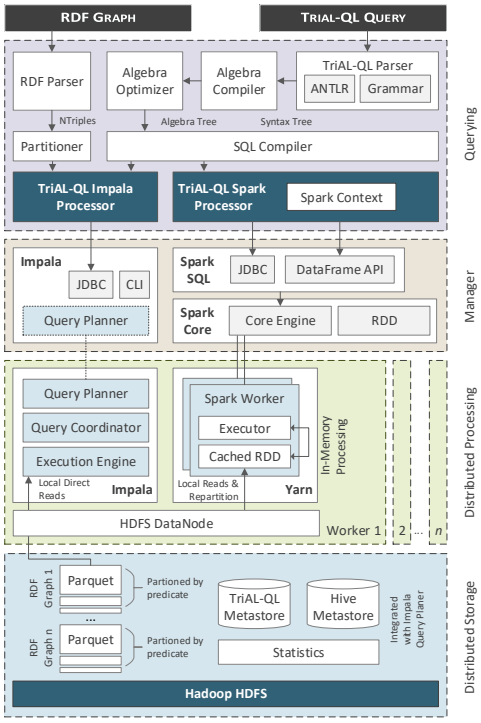

Figure 1: Architecture of TriAL-QL Engine

4.1 Relational Mappings for RDF

Typically, RDF triplestores with DBMS back-ends represent an RDF dataset in a so-called *triples table* with three columns, containing one row for each RDF statement, i.e. $triples(sub, pred, obj)$. Query evaluation then essentially boils down to a series of self-joins on this table. Therefore, it is often accompanied by several indexes over some or all (six) triple permutations, e.g. based on B^+-trees, for query speedup. However, this kind of indexes are not well-suited and hard to maintain in a distributed computing environment like Hadoop. In [1] the authors propose a *vertical partitioned* schema having a two-column table for every RDF predicate, e.g. $knows(sub, obj)$, enabling more efficient pruning strategies. Otherwise, having a separate table for every predicate is not a well-suited schema for joins on predicates. This may not be common in SPARQL but it is a natural

join pattern in TriAL*, e.g. for on the fly reasoning over predicates (cf. Use Case 2 in Sec. 5).

Another typical approach is the use of so-called *property tables* where all predicates (or properties) that tend to be used in combination are stored in one table, e.g. all predicates used to describe a person. This reduces the number of subject-subject self-joins for *star-shaped* query patterns. Although such patterns can also occur in TriAL*, it is not the dominant pattern in navigational queries. Furthermore, TriAL* is a closed language that allows to derive new triples to be added to the triplestore which would result in update operations on one or more property tables, an operation that is currently not supported by any of the investigated systems (Impala, Spark, Hive) due to the fact that the underlying distributed filesystem (HDFS) is append-only.

In our use case, we basically need to consider two aspects in our table layout: (1) the fact that the data is distributed on a cluster of machines hampering the use of indexes, and (2) the flexibility imposed by TriAL* to add new triples and perform joins on all possible pattern combinations (i.e. also predicate-predicate joins). This in mind, we use a triples table internally partitioned by predicates, $triples_{pred}(sub, obj)$, to represent an RDF dataset and use the name of the dataset as the name of the triplestore (table). Tables are stored using *Parquet* [7] , an efficient columnar storage format for Hadoop with built-in support for compression (snappy), run-length and dictionary encoding. This table layout is supported by all three systems, hence we can use the very same table for all of them, demonstrating the benefit of a unified data storage. Partition pruning is applied transparently whenever possible, i.e. unnecessary partitions are filtered out automatically, and table statistics (e.g. partition sizes) are used to optimize the query plans, e.g. improving join order. Thus, we combine the full flexibility of a triples table with the efficiency enhancement of vertical partitioning while having just a single unified data storage without the need for expensive data exchange or conversion from one system to another.

4.2 Evaluation of Kleene Closure

One of the most challenging expressions of TriAL* is the *right* $(e \bowtie^{i,j,k}_{\theta,\eta})^*$ and $left$ $(\bowtie^{i,j,k}_{\theta,\eta} e)^*$ *Kleene closure*, which allows to express paths of arbitrary length. Its computation requires a continuous sequence of iterations until *all* reachable instances are retrieved. Since neither Impala nor Hive or Spark support any kind of recursion, such an expression cannot be translated into a single SQL statement but need to be broken down into several smaller queries that are executed subsequently. Therefore, an additional mechanisms is needed that (1) initiates each of these iterations and (2) determines the progress and decides whether it terminates. For Impala and Hive, the intermediate result of each iteration has to be materialized on disk and used as input for the next iteration as there is no support to preserve those intermediate tables in memory across multiple SQL queries. However, Spark supports to cache tables in memory, such that a subsequent iteration does not need to read from disk.

For the actual processing, we need to define an efficient algorithm which evaluate the recursive expressions in TriAL*. Indeed, we can reduce such an expression to the problem of calculating the transitive closure (TC), which is a well-studied research field [14, 20, 29]. There is an ongoing debate whether the so-called *semi-naive* or *smart* TC algorithm is superior in distributed environments like MapReduce [3, 47]. However, there has not been much work yet on investigating the trade-offs using novel SQL-on-Hadoop solutions. We believe that, for our scenario, a *semi-naive* evaluation [14] is the better choice as it distributes the workload over more rounds and produce less derivations on graphs with cycles [4]. In contrast, a *smart* TC algorithm based on a *nonlinear* (*recursive-doubling*) execution [29] uses a logarithmic, rather than linear, number of rounds but with much higher costs (with regard to the data volume) per round [4]. Thus, using the *semi-naive* evaluation leads to more but less expensive joins, which in turn increases the chances that the actual join processing can be done in memory without spilling to disk.

Our algorithm for computing the *right Kleene closure* $(E \bowtie^{i,j,k}_{\theta,\eta})^*$ based on *semi-naive* evaluation is depicted in Algorithm 1. E denotes the input triplestore (table) and ΔP^n contains those triples which were newly derived in the n-th iteration. The final result, P, is the union of all previous iterations.

Algorithm 1: Semi-naive eval. of *right Kleene closure*

```
   input: triplestore E
1  n ← 0, ΔP^0 ← E
2  while ΔP^n ≠ ∅ do
3  |  n ← n + 1
4  |  tmp = ΔP^{n-1} ⋈^{i,j,k}_{θ,η} E
5  |  ΔP^n = tmp − (ΔP^0 ∪ ... ∪ ΔP^{n-1})
6  end
7  return P = ΔP^0 ∪ ... ∪ ΔP^n
```

In addition, this approach can be further improved by exploiting some properties of partitioned tables in Impala and Hive. Instead of creating a new table for each ΔP^n, it is far more effective to use a single table $P_{iter}(sub, pred, obj)$ partitioned by iteration number *iter* and only add a new partition to that table. This way, the amount of required operations can be reduced since the results of all *union* operations used in line 5 and line 7 can be retrieved by partition pruning without any computational effort. At the time when we performed our experiments, the support of Spark for partitions was in a rather early stage, not allowing us to adapt this strategy as efficiently as for Impala and Hive. Thus for Spark we need to introduce two tables: one for newly derived triples (ΔP^n), and a second one that keeps the result of $(\Delta P^0 \cup ... \cup \Delta P^{n-1})$. The algorithm terminates, if round i does not derive any new triples, i.e. all the derived triples are already contained in $(\Delta P^0 \cup ... \cup \Delta P^{n-1})$ or $(\Delta P^{n-1} \bowtie E)$ is empty. This step is realized by an additional SQL query that counts the number of triples in ΔP^n.

4.3 Evaluation of Connectivity Patterns

A further challenging query type that we identified to be relevant for many application fields is the connectivity between two given resources, thus the existential question whether there exists a path that connects both:

$$\sigma_{s=<\text{startNode}>,\ o=<\text{endNode}>} \left((E \bowtie^{i,j,k}_{\theta,\eta})^* \right)$$

Like the aforementioned *Kleene closure* expression, its computation involves a continuous sequence of iterations, thus it cannot be translated into a single SQL statement. A

naive compositional evaluation with the previous algorithm would derive a huge amount of redundant triples that need to be discarded afterwards. However, instead of retrieving all connections between both resources, we solely need to check for the existence of at least *one* path connecting them. The corresponding algorithms is depicted in Algorithm 2.

Again, we use the *semi-naive* evaluation, but this time we start with two initial tables denoted by ΔP_l^0 and ΔP_r^0, which contain only those triples that *start* and *end* with the given resources, respectively. The algorithms then alternately derives new triples for ΔP_l^n and ΔP_r^n. In other words, we perform a *breadth-first* search that starts from both ends. As in the previous case, we can exploit Impalas partitioning strategy by creating two tables, $Pl_{iter}(sub, pred, obj)$ and $Pr_{iter}(sub, pred, obj)$, partitioned by iteration number to reduce the computational effort. Finally, the existence of a path of length i is checked by joining the two tables for ΔP_l^n and ΔP_r^n, where just the number of results is needed (denoted by res). The algorithm terminates, if either a connection is found, thus $res \neq 0$, or both ΔP_l^n and ΔP_r^n are empty, i.e. we computed the *reachability* starting from both ends without finding an intersection.

Algorithm 2: Semi-naive eval. of *connectivity pattern*

input: triplestore E, resource $startNode$, resource $endNode$

$n \leftarrow 0$, $\Delta P_l^0 \leftarrow \sigma_{s\,=\,startNode}(E)$, $\Delta P_r^0 \leftarrow \sigma_{o\,=\,endNode}(E)$
$res = count\left(\Delta P_l^{\lceil n/2 \rceil} \bowtie_{\theta,\eta}^{i,j,k} \Delta P_r^{\lceil n/2 \rceil}\right)$
while $\left(\Delta P_l^{\lceil n/2 \rceil} \neq \emptyset \;\;\&\;\; \Delta P_r^{\lceil n/2 \rceil} \neq \emptyset\right) \;\&\; (res = 0)$ **do**
  $n \leftarrow n + 1$
  if $(n \; mod \; 2) = 1$ **then**
    $tmp_l = \Delta P_l^{\lceil (n-1)/2 \rceil} \bowtie_{\theta,\eta}^{i,j,k} E$
    $\Delta P_l^{\lceil n/2 \rceil} = tmp_l - \left(\Delta P_l^0 \cup ... \cup \Delta P_l^{\lceil (n-1)/2 \rceil}\right)$
  else
    $tmp_r = E \bowtie_{\theta,\eta}^{i,j,k} \Delta P_r^{\lceil (n-1)/2 \rceil}$
    $\Delta P_r^{\lceil n/2 \rceil} = tmp_r - \left(\Delta P_r^0 \cup ... \cup \Delta P_r^{\lceil (n-1)/2 \rceil}\right)$
  end
  $res = count\left(\Delta P_l^{\lceil n/2 \rceil} \bowtie_{\theta,\eta}^{i,j,k} \Delta P_r^{\lceil n/2 \rceil}\right)$
end
return res

4.4 Query Composition

As described in the previous sections, each TriAL expression is mapped to one or more corresponding SQL queries. In a sequence of queries, $q_0 \ldots q_i$, where q_i uses the output of q_{i-1} as input, we can exploit the fact that SQL itself is also a closed and compositional language. Instead of executing i isolated queries sequentially (referred to as ***materialized*** execution), we can combine them into a single composite query (referred to as ***composite*** execution). This is especially interesting for Impala to avoid materializing the output of $q_0 \ldots q_{i-1}$ to HDFS to serve as input for the next query. Unfortunately, we cannot use this strategy for recursive TriAL expressions as Impala does currently not support recursion. In Spark, we have the choice whether to materialize the result of a query in HDFS or not as Spark supports to cache tables in memory. Moreover, we can even use a composite execution for recursive TriAL expressions by utilizing the fact that in Spark a user can embed SQL queries in a general Scala or Java program used as a driver. For Hive (on MapReduce), it does not make a difference between both strategies as intermediate results are anyway materialized in HDFS.

However, composition is not always superior to a sequential execution mainly for two reasons: (1) Composition complicates the query plan and may lead to incorrect cardinality estimations in a sequence of joins and thus suboptimal execution plans. (2) We observed that a sequence of small queries reduces the overall memory usage compared to a single pudgy query and thus increases the probability that the system does not spill to disk which slows down join processing substantially. Experiments in Sec. 5 demonstrate that none of both strategies is clearly superior to the other.

5. EXPERIMENTS

The experiments were performed on a cluster with ten machines, each equipped with a six core Xeon E5-2420 CPU, 2×2 TB disks and 32 GB RAM. We used the Hadoop distribution of Cloudera CDH 5.7.0 with Impala 2.5.0, Spark 1.6.2 and Hive 0.13.1. The machines were connected via Gigabit network. Both the Impala daemon and Spark executor were using 24 GB of main memory, broadcast joins were disabled, and the Parquet filter push-down optimization were enabled. For competitors executed on a single machine we used a workstation equipped with a Intel Xeon E5-2640 CPU with six cores, twelve threads, 192 GB main memory and 2×1 TB disks.

5.1 Experimental Use Cases

The first part of our experiments is based on a set of representative use cases with different characteristics that enable us to demonstrate the expressiveness of TriAL-QL along with its costs regarding scalability on large RDF graphs. The actual queries are inspired by typical graph analytical questions and recommendations on social networks. We use the state of the art *Social Network Benchmark* (SNB) data generator[4] to generate synthetic social networks of up to 1.8 billion triples (edges) with power-law structure which also has been used in the SIGMOD 2014 programming contest. The load times and store sizes are listed in Table 1. We examine both execution strategies introduced in Sect. 4.4 (*materialized* and *composite*) for Impala and Spark and compare them with an execution on Hive using MapReduce. All results are listed in Table 2. Each query was executed five times, the corresponding coefficients of variation c_v (ratio between standard deviation and mean) are also given.

Table 1: Load times and store sizes

Scaling Factor	1	3	10	30
Triples (in M)	59.5	176.4	594.7	1,799
RDF size	4.1 GB	12.3 GB	41.4 GB	126 GB
Triples Table	0.6 GB	1.7 GB	6.2 GB	19 GB
Loading	32.4 s	90.4 s	299.3 s	893.1 s
Parquet file size	8 MB	8 MB	16 MB	32 MB

Use Case 1: Socialized Recommendations. In the first experiment we ask for the posts of a users friends that he has not liked yet, computed for all users in parallel. For the sake of brevity, we omit the TriAL* expression for *knows*,

[4]http://ldbcouncil.org/developer/snb

posts and *likes*, where each is retrieved by an additional join on the input table *snb*. However, they are included in the execution times. The corresponding TriAL* expressions are defined as follows:

$$friendsPosts = knows \bowtie^{s_1,\ \text{fposts},\ o_2}_{o_1=s_2} posts$$
$$likedFriendsPosts = friendsPosts \bowtie^{s_1,\ \text{fposts},\ o_2}_{s_1=s_2,\ o_1=o_2} likes$$
$$postSuggestions = friendsPosts - likedFriendsPosts$$

The query contains no recursion and could also be expressed with SPARQL, but it illustrates the strength of composition where an expression processes the results of previous ones. In total it consists of five joins and a set operation that process large portions of the underlying RDF graph. We can see that the *composite* execution with Impala performs best (2x faster than *materialized* on average) while scaling almost linear with increasing data size. Spark is competitive for smaller data sizes but gets significantly slower for larger ones. Hive is an order of magnitude slower than Impala but still scales out smoothly with larger data sizes. Summarized, for rather data-intensive but not too complex queries as used for this use case, Impala performs best while Hive exhibits slightly better scaling properties.

Use Case 2: Reachability. This experiment defines a reachability query that cannot be answered by just traversing the input graph, but rather needs reasoning capabilities. We consider two arbitrary persons to be reachable, if we can derive a path between them where (1) each intermediate pair of persons work together in the same company, and (2) all persons along the path share the same spoken language. The corresponding TriAL* expressions are as follows:

$$worksAt = \text{snb} \bowtie^{s_1,\ \text{worksAt},\ o_2}_{o_1=s_2,\ p_1=sn:workAt,\ p_2=sn:hasOrga} \text{snb}$$
$$sameWork = worksAt \bowtie^{s_1,\ \text{worksWith},\ s_2}_{o_1=o_2,\ s_1!=s_2} worksAt$$
$$sameLang = \text{snb} \bowtie^{s_1,\ o_1,\ s_2}_{o_1=o_2,\ s_1!=s_2,\ p_1=p_2,\ p_1=sn:speaks} \text{snb}$$
$$colleagues = sameLang \bowtie^{s_1,\ p_1,\ o_1}_{s_1=s_2,\ o_1=o_2} sameWork$$
$$reach = \left(colleagues \bowtie^{s_1,\ p_1,\ o_2}_{p_1=p_2,\ o_1=s_2}\right)^*$$

We split the analysis of the execution times in Table 2 into two parts: In ① we summarize all non-recursive expressions including *colleagues* and in ② we investigate the computation of the transitive closure expressed by *reach*.

For ① we can see that the Impala runtimes again scale almost linear with increasing data sizes. But this time, *materialized* execution is superior to *composite* for larger datasets. We explain this behavior with a better execution plan of Impala due to statistics computed for intermediate tables. The costs for computing statistics are included in Table 2 (cf. column *statistics*). For ② again Impala outperforms Spark, although it was heavily spilling to disk during query execution (starting from SF 10). Moreover, Spark runs out of memory on larger datasets (indicated by `MEM`). This might result from the two table approach required for Spark (c.f. Sect. 4.2) compared to the partitioned table approach used for Impala. As a consequence, Spark needs more memory and requires additional computational effort. Again, Hive was significantly slower than Impala and Spark but exhibits a better scaling for larger datasets. More carefully, we can observe that the performance benefit of Impala is mainly attributed to the more efficient determination of newly derived triples (ΔP^i) rather than join computation.

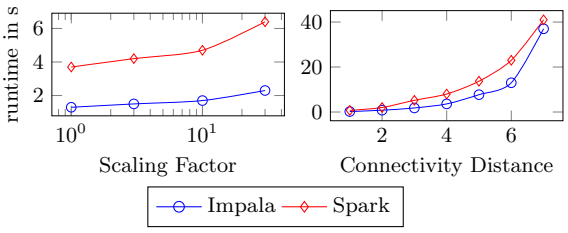

Figure 2: Mean runtimes (in s) for Use Case 3: (left) total mean, (right) by path distance for SF 30

Use Case 3: Connectivity. The last experiment asks for the existence of a path between two given persons by following the friendship relationship. We choose 50 persons randomly and compute their connectivity based on the following TriAL* expressions:

$$knows = snb \bowtie^{s_1,\ \text{knows},\ s_2}_{o_1=s_2,\ p_1=sn:knows,\ p_2=sn:hasPers} snb \quad ③$$
$$path = \sigma_{s=\text{person\_1},\ o=\text{person\_2}}((knows \bowtie^{s_1,\ p_1,\ o_2}_{o_1=s_2})^*) \quad ④$$

Note that ③ *knows* is only computed once and stored as a new relation (table) in the triplestore and used to compute ④ *path* over and over again, bridging the gap between ETL-like workloads and explorative ad-hoc style queries. Hive was not considered in this experiment since MapReduce start-up costs are already higher than desired runtimes for this query type. However, it will be interesting to investigate novel derivations of Hive (Hive on Tez, Hive on Spark) that aim to replace MapReduce with a more interactive execution framework for future work.

The mean runtime for computing the connectivity between two randomly chosen persons with Impala was only 2.3 seconds on a dataset with 1.8 billion triples while scaling smoothly with the data size (cf. Figure 2). For Spark it is 6.4 seconds which is still competitive. The average distance between two persons was 2.8 for the smallest dataset and increased up to 3.2 for the largest one. Table 2 also lists the runtimes by distance (denoted by ⑤). Considering distances of at most 2, we get runtimes of < 1 second and a maximum runtime of 37 seconds for a distance of 7. This is also illustrated in the right plot of Figure 2 where both Impala and Spark exhibit same exponential scaling behavior.

Comparison of Impala and Spark

In summary, the overall performance our TriAL-QL Engine using Impala was continuously better than for Spark. However, a more granular view on the actual costs for the respective operations reveals some further interesting properties which highlight some strengths and weaknesses of both. Figure 3 illustrates the percentage of the total runtime of the respective operations for Use Case 2 (see Algorithm 1 in Sect. 4.2). Here we can see that computing ΔP^i, i.e. determine the newly derived triples in iteration i, dominates the total execution time of Spark whereas this task is very efficient in Impala due to the partitioned table approach which is not applicable in Spark 1.6.2 (cf. Sect 4.2). However, if we consider the actual join processing costs ($tmp = \Delta P^{i-1} \bowtie^{i,j,k}_{\theta,\eta} E$), the performance of Spark is even faster than Impala. Thus, a better support for partitioned tables in future Spark versions might reveal other performance characteristics.

Table 2: Mean runtimes (in s) comparing *composite* and *materialized* execution on Impala, Spark- and Hive, c_v = coefficient of variation, ΔP^i and *tmp* match with Algorithm 1.

Use Case 1

SF	Impala (comp.) total (c_v)	Impala (comp.) write	Impala (mat.) total (c_v)	Impala (mat.) stats	Impala (mat.) write	Spark (comp.) total (c_v)	Spark (comp.) write	Hive total (c_v)	Hive results
1	15.7 (2.5%)	3.6	42.3 (4.8%)	6.9	5.3	24.1 (2.0%)	2.3	283.4 (1.3%)	1,161,253
3	29.0 (1.8%)	5.0	57.9 (2.9%)	9.4	5.0	57.8 (2.4%)	4.1	444.1 (0.8%)	3,569,709
10	67.5 (1.1%)	4.9	120.7 (1.0%)	20.1	4.7	188.1 (1.0%)	5.9	979.2 (0.5%)	12,635,382
30	188.8 (1.4%)	7.1	323.7 (1.4%)	51.6	5.4	831.5 (3.8%)	5.9	2553.5 (0.8%)	39,442,329

Use Case 2 ① *colleagues*

SF	Impala (comp.) total (c_v)	Impala (comp.) write	Impala (mat.) total (c_v)	Impala (mat.) stats	Impala (mat.) write	Spark (comp.) total (c_v)	Spark (mat.) total (c_v)	Hive total (c_v)
1	16.9 (2.2%)	3.9	29.3 (2.7%)	5.5	3.6	44.5 (1.3%)	50.0 (1.9%)	250.7 (1.0%)
3	38.3 (1.5%)	4.0	43.2 (4.9%)	6.4	4.2	83.8 (2.5%)	90.6 (1.1%)	294.4 (0.6%)
10	159.0 (0.7%)	5.1	121.1 (3.9%)	11.6	6.2	321.6 (1.3%)	333.2 (1.7%)	605.3 (0.8%)
30	866.6 (0.4%)	9.7	631.9 (6.2%)	42.4	9.1	2048.0 (5.2%)	2039.9 (2.7%)	2501.2 (0.3%)

Use Case 2 ② *reach*

SF	Impala (mat.) total (c_v)	Impala (mat.) ΔP^i	Impala (mat.) *tmp*	Spark (mat.) total (c_v)	Spark (mat.) ΔP^i	Spark (mat.) *tmp*	Hive total (c_v)	Hive ΔP^i	Hive *tmp*	iterations	results
1	201 (1.4%)	32	122	483.7 (2.2%)	396.1	87.6	2727 (0.3%)	1046	1101	17	2.6 M
3	431 (2.6%)	65	316	847.1 (0.7%)	539.6	307.5	4749 (1.5%)	1737	2424	17	19.3 M
10	5184 (1.0%)	362	4760	MEM	MEM	MEM	33048 (0.5%)	2768	29732	15	153 M
30	22933 (4.7%)	725	22148	MEM	MEM	MEM	60178 (1.4%)	4611	54929	14	591 M

Use Case 3

	SF	③ *knows* total (c_v)	④ *connect.* (*mean*) total	④ *connect.* (*mean*) ∅*distance*	⑤ *connectivity by distance*, total (c_v) 1	2	3	4	5	6	7
Impala	**1**	4.4 (4%)	1.3	(∅ 2.8)	0.1 (0%)	0.5 (0%)	1.3 (3%)	2.7 (2%)	4.8 (3%)	7.9 (2%)	13 (2%)
	3	8.1 (4%)	1.5	(∅ 3.0)	0.1 (0%)	0.5 (3%)	1.4 (3%)	2.9 (2%)	5.3 (2%)	8.9 (2%)	14 (1%)
	10	13.0 (6%)	1.7	(∅ 3.0)	0.1 (24%)	0.6 (7%)	1.6 (5%)	3.2 (4%)	5.9 (3%)	9.9 (2%)	20 (1%)
	30	25.1 (2%)	2.3	(∅ 3.2)	0.2 (0%)	0.8 (2%)	1.8 (3%)	3.6 (3%)	7.7 (2%)	13 (2%)	37 (1%)
Spark	**1**	3.3 (0%)	3.7	(∅ 2.8)	0.4 (2%)	1.5 (6%)	4.1 (4%)	6.8 (5%)	12.1 (5%)	18 (5%)	33 (3%)
	3	7.7 (0%)	4.2	(∅ 3.0)	0.5 (1%)	1.6 (3%)	4.2 (1%)	6.8 (0%)	12.3 (3%)	19 (2%)	34 (1%)
	10	18.8 (3%)	4.7	(∅ 3.0)	0.5 (2%)	1.6 (1%)	4.2 (1%)	7.1 (7%)	12.6 (7%)	20 (4%)	35 (3%)
	30	56.6 (1%)	6.4	(∅ 3.2)	0.7 (30%)	2.0 (28%)	5.2 (15%)	8.0 (10%)	13.7 (5%)	23 (10%)	41 (9%)

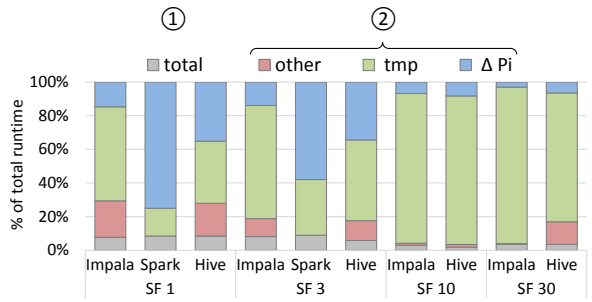

Figure 3: Runtimes by task (in %) for Use Case 2

Another aspect is the superlinear increase in time required to compute ② *reach* starting from SF 10 which is mainly attributed to the actual join processing (*tmp*). Spark runs out of memory and fails while Impala starts to spill heavily to disk but is able to complete the job. This demonstrates that Impala is currently more robust when it comes to memory bottlenecks and thus the need for on-disk joins.

5.2 Waterloo SPARQL Diversity Test Suite

We based the second part of our experiments on the *Waterloo SPARQL Diversity Test Suite* [5] (WatDiv), which provides a test environment for RDF data management systems with more diverse workloads than other benchmarks. We generated datasets from ten million to a billion RDF triples using the WatDiv data generator with scaling factors 100, 1000, and 10000. Since we focus in this dissertation on path-based queries, we used the *Incremental Linear Testing* use case which focus on path-shaped patterns. It consists of three query types (IL-1, IL-2, IL-3) which are bound by user, retailer or unbounded, respectively. Each query starts with a length of 5 (IL-1-5) and becomes incrementally increased up to 8 (IL-1-8).

We compare our TriAL-QL Engine with six competitive RDF Management systems and one graph database[5]. Since the original WatDiv queries are written in SPARQL, we translated them in the languages supported by the respective system. As two representative SPARQL-query-processors that use MapReduce we chose Shard [41] and PigSPARQL [42]. Shard is written directly in MapReduce, where each triple pattern is mapped to exactly one reduce-side-side-join. Data is stored in HDFS, where triples are grouped by their subjects and put together in one line. PigSPARQL is a SPARQL query processor which, instead of a direct mapping, translates into Pig Latin as an intermediate layer between MapReduce. Data is stored vertically-partitioned in HDFS. With H2RDF+ [36], we have one representative SPARQL engine built on top of a NoSQL store. H2RDF+ uses HBase to store triples sorted by row keys in six different triple permutations. Based on the selectivity of a triple pattern, queries are either executed on a single node or distributed using MapReduce. Two representative SPARQL

[5]Results for TriAL-QL, S2RDF and Neo4j are based on two master thesis [12, 46]

engines that utilize in-memory processing frameworks are Sempala [43] and S2RDF [44]. Sempala is built on top of Impala. Its RDF data layout is highly optimized for star-shaped queries and enables interactive querying times on large RDF graphs. S2RDF is a fast SPARQL-on-Hadoop engine built on top of Spark SQL. It stores its data using Extended Vertical Partitioning (ExtVP), which efficiently minimizes the query input size regardless of the query pattern shape and diameter. Virtuoso Open Source Edition v7.1.1 [19] represents a state-of-the-art centralized RDF data management system using a relational database to store data. It supports SPARQL 1.1, OWL reasoning, and benefits from indexes and a two-level compression strategy optimized for RDF. As a last competitor we chose the graph database system Neo4j [33]. Neo4j, of which we used version 3.0.6, is one of the most prominent native graph-databases running on a single-machine. Data representation is based on the *Labeled Property Graph* model, which consists of entities (nodes) and relationships (labeled edges). A connection between two entities is then represented by a directed and named relationship. While loading the WatDiv data into Neo4j, we modeled an RDF triple (s, p, o) of entities s and o connected by a relationship p. Here it was crucial to ensure that two identical IRIs (only subjects and objects) refer to the same node in the property graph. We skipped literals, since they are not required for the *Incremental Linear Testing* use case, although they could have been easily represented by so-called attributes. Further, we translated the WatDiV queries into Cypher, Neo4j's graph query language, which is a declarative, pattern-matching language comparable to regular path queries.

Table 3: WatDiv load times and HDFS sizes

		SF100	SF1000	SF10000
tuples	**original**	10.91 M	109.2 M	1091.5 M
	TriAL-QL VP	10.91 M	109.2 M	1091.5 M
	S2RDF VP	10.91 M	109.2 M	1091.5 M
	S2RDF ExtVP	119.94 M	1197.9 M	11967 M
	Neo4j Graph	9.58 M	95.99 M	959.4 M
HDFS size	**original**	507 MB	5.3 GB	54.9 GB
	TriAL-QL VP	103 MB	1.2 GB	13.2 GB
	S2RDF VP	82 MB	0.6 GB	6.6 GB
	S2RDF ExtVP	914 MB	6.2 GB	63.7 GB
	H2RDF+	517 MB	5.2 GB	57.0 GB
	Sempala	249 MB	3.5 GB	40.4 GB
	PigSPARQL	871 MB	8.9 GB	92.5 GB
	SHARD	981 MB	9.9 GB	100 GB
	Neo4j Graph	4425 MB	50.9 GB	536.7 GB

Discussion on Store Sizes. The store sizes for all three generated datasets are listed in Table 3. We can see that the store sizes of our engine are significantly smaller than the sizes of the original RDF graph. This is achieved by Parquets built-in support for run-length and dictionary encoding in combination with snappy compression that perform great for storing RDF in a column-oriented format. The largest store sizes are created by Neo4j's *Labeled Property Graph* model, which was also reflected in its loading times. Loading the smallest dataset took 12 hours and the largest one ten days. The main workload was therefore the insertion of nodes and the updating of their indexes.

Discussion on Performance. Figure 4 illustrates the runtime differences between all benchmarked systems in a log-scaled bar chart. The first result is that our Trial-QL Engine executed on Impala outperforms most competitors. Regarding individual runtimes, we want to emphasize that for IL-3 on SF 10000, which produces more than 25 billion results, TriAL-QL required less than 24 minutes to finish. In comparison, the slowest execution, which was PigSPARQL on MapReduce, lasts for more than 11 hours and the best competitor, S2RDF, required about 34 minutes to compute its results. Both single-machine engines, Neo4j and Virtuoso, were not able evaluate this query on the largest dataset using a time constraint of 24 hours. The other range of execution times is also worth noting. For IL-1 and IL-2, the mean runtimes for TriAL-QL are 1.2 seconds on SF 100, which contains 11 million triples. On SF 10000, which is a graph with over one billion triples, TriAL-QL finished its execution in 7.8 seconds for Il-1 and 7.5 seconds for Il-2.

Next, we discuss the performance differences between both Spark and Impala. We can see that our engine perform significantly better using Impala rather than Spark, which is in line with our previous experimental observations in the first part of this section. One reason for the better performance of Impala are our proposed algorithms and storage strategies that are closely related to relational algebra, which in turn is the core component of Impala. Moreover, Impala is a pure SQL-engine, whereas Spark is a general-purpose execution framework with support for many other querying interfaces. Overall we can conclude, that the synergy effects of using Impala together the proposed evaluation and storage strategies are a good fit for the examined queries. Nonetheless, both, Impala and Spark, have their strengths and weaknesses while being continuously improved. Future Spark versions might therefore reveal other performance characteristics.

Next, we have a closer look at both MapReduce engines (Shard and PigSPARQL), which are clearly outperformed by all other systems. This is actually not a surprising result due to the batch-oriented workflow of MapReduce and the usage of disk-based operations. On the positive side we can note that all queries were executable even on the largest datasets, whereas single-machine approaches such es Neo4j and Virtuoso failed on the heavy-load query (IL-3). Due to their disk-based operators, we expect that these systems work also on much larger datasets, whereas our approaches based on in-memory frameworks are expected to fail once they run out of main memory. There is also support for disk-based operations in Impala and Spark, which are significantly slower but at least ensure the execution without running out of memory. However, this has to be specified in advance while compiling a query. During query execution, there is only support to perform so-called spilling to disk, which massively decreases the overall performance and is only applicable in cases where just a relatively small amount of data needs to be moved to disk.

Summary. Overall, the evaluation clearly demonstrates that distributed frameworks such as Impala and Spark, combined with proper evaluation strategies and good data storage, provide an excellent basis for the evaluation of navigational query languages such as TriAL-QL. Our implementation outperformed most evaluated competitors. For selective queries we exhibit runtimes in the order of a few seconds. On data-intensive queries, which produced more than 25 billion results, we obtained runtimes in 25 minutes.

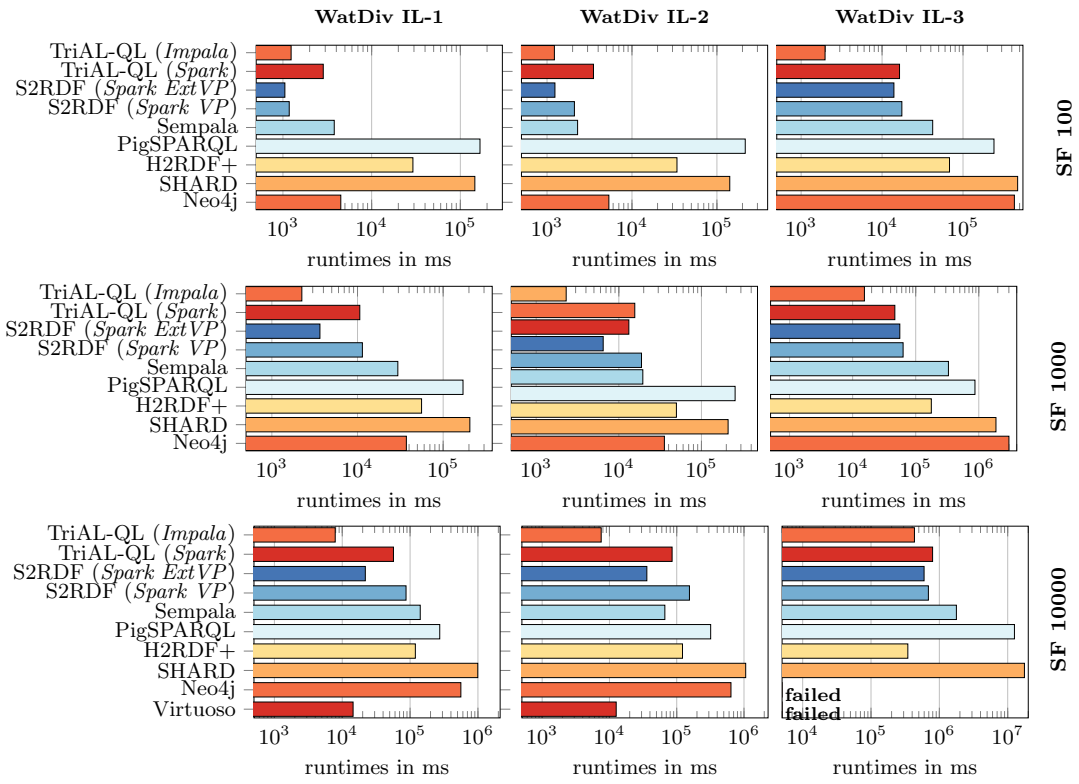

Figure 4: Comparison of mean runtimes on a log scale for WatDiv Incremental Linear Testing 1, 2, and 3.

6. CONCLUSION

Our TriAL-QL Engine demonstrated that the Hadoop ecosystem provides suitable solutions for processing expressive, navigational queries against web-scale knowledge bases. Selective queries exhibit query response times in the order of seconds on datasets with more than one billion triples. More data-intensive use-cases that produces, e.g. over 25 billion results finished in the order of minutes. Yet one of the main benefits we see in the usage of the Hadoop ecosystem is the concept of a common data pool that is shared across various RDF management engines, developed on top of Hadoop. This key concept further constitutes a compatible ecosystem for processing RDF data where we see our engine as complementary tool that can then be embedded in a workflow for, e.g. preprocessing more complex paths. For future work, we are planing to extend TriAL-QL with the concept of provenance, to trace the origin of a triple by means of paths and consider Apache Kudu for storing data.

7. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *VLDB*, pages 411–422, 2007.

[2] S. Abiteboul, M. Bienvenu, A. Galland, and M. Rousset. Distributed datalog revisited. In *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, pages 252–261, 2010.

[3] F. N. Afrati, V. R. Borkar, M. J. Carey, N. Polyzotis, and J. D. Ullman. Map-reduce extensions and recursive queries. In *EDBT 2011, Sweden, March 21-24*, 2011.

[4] F. N. Afrati and J. D. Ullman. Transitive closure and recursive datalog implemented on clusters. In *EDBT'12, Berlin, Germany, March 27-30, 2012*, pages 132–143, 2012.

[5] G. Aluc, O. Hartig, M. Özsu, and K. Daudjee. Diversified Stress Testing of RDF Data Management Systems. In *ISWC*, volume 8796 of *LNCS*, pages 197–212, 2014.

[6] R. Angles. A comparison of current graph database models. In *28th ICDE Workshops, 2012, Arlington, USA*, 2012.

[7] Apache. Apache Parquet. http://parquet.io.

[8] M. Arenas, G. Gottlob, and A. Pieris. Expressive languages for querying the semantic web. In *Proc. of the 33rd ACM Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 14–26, 2014.

[9] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394. ACM, 2015.

[10] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. DBpedia - A crystallization point for the Web of Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):154–165, 2009.

[11] P. A. Boncz, O. Erling, and M.-D. Pham. Experiences with Virtuoso Cluster RDF Column Store. In *Linked Data Manag.*, pages 239–259. Chapman and Hall/CRC, 2014.

[12] V. Boshnjaku. A Scalable Engine for TriAL-QL on SQL-on-Hadoop. M.Sc. Thesis, University Freiburg, 2016.

[13] J. Broekstra, A. Kampman, and F. v. Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *The Semantic Web - ISWC 2002*, number 2342 in Lecture Notes in Computer Science, pages 54–68. Springer Berlin Heidelberg, 2002.

[14] F. Cacace, S. Ceri, and M. A. Houtsma. An overview of parallel strategies for transitive closure on algebraic machines. In *Parallel Database Systems*, pages 44–62. Springer, 1991.

[15] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: Implementing the Semantic Web Recommendations. In *Proc. WWW Alt.*, pages 74–83, 2004.

[16] P. Csermely, T. Korcsmáros, H. J. Kiss, G. London, and R. Nussinov. Structure and dynamics of molecular networks: a novel paradigm of drug discovery: a comprehensive review. *Pharmacology & therapeutics*, 138(3):333–408, 2013.

[17] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation (OSDI)*, pages 137–150, San Francisco, California, USA, 2004. USENIX Association.

[18] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge Vault: A Web-scale Approach to Probabilistic Knowledge Fusion. In *Proc. of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 601–610. ACM, 2014.

[19] O. Erling and I. Mikhailov. Virtuoso: RDF Support in a Native RDBMS. In *Semantic Web Information Manag.*, pages 501–519. Springer Berlin Heidelberg, 2010.

[20] G. H. L. Fletcher, M. Gyssens, D. Leinders, J. V. den Bussche, D. V. Gucht, S. Vansummeren, and Y. Wu. The impact of transitive closure on the expressiveness of navigational query languages on unlabeled graphs. *Ann. Math. Artif. Intell.*, 73(1-2):167–203, 2015.

[21] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing. In *Proc. of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 289–300. ACM, 2014.

[22] M. Hammoud, D. A. Rabbou, R. Nouri, S.-M.-R. Beheshti, and S. Sakr. DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication. *Proc. VLDB Endow.*, 8(6):654–665, 2015.

[23] S. Harris and N. Gibbins. 3store: Efficient Bulk RDF Storage. In *Proceedings of the First International Workshop on Practical and Scalable Semantic Systems*, volume 89 of *CEUR Workshop Proceedings*, 2003.

[24] S. Harris, N. Lamb, and N. Shadbolt. 4store: The Design and Implementation of a Clustered RDF Store. In *Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, volume 517 of *CEUR Workshop Proceedings*, 2009.

[25] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *The Semantic Web*, number 4825 in Lecture Notes in Computer Science, pages 211–224. Springer Berlin Heidelberg, 2007.

[26] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *AI*, 194:28–61, 2013.

[27] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11):1123–1134, 2011.

[28] M. F. Husain, J. P. McGlothlin, M. M. Masud, L. R. Khan, and B. M. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE TKDE*, 23(9), 2011.

[29] Y. E. Ioannidis. On the computation of the transitive closure of relational operators. In *VLDB'86, August 25-28, 1986, Kyoto, Japan, Proceedings.*, pages 403–411, 1986.

[30] K. Lee and L. Liu. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *Proc. VLDB Endow.*, 6(14):1894–1905, 2013.

[31] L. Libkin, J. L. Reutter, and D. Vrgoc. Trial for RDF: adapting graph query languages for RDF data. In *Proceedings of the 32nd ACM PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 201–212, 2013.

[32] F. Manola, E. Miller, and B. McBride. RDF 1.1 Primer. http://www.w3.org/TR/rdf-primer/, 2014.

[33] Neo-Technology. Neo4j. https://neo4j.com/, 2016.

[34] T. Neumann and G. Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *The VLDB Journal*, 19(1):91–113, 2010.

[35] A. Owens. Clustered TDB: A Clustered Triple Store for Jena. In *Proceedings of the 18th international conference on World Wide Web (WWW)*, 2009.

[36] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris. H2rdf+: High-performance distributed joins over large-scale RDF graphs. In *2013 IEEE International Conference on Big Data*, pages 255–263, 2013.

[37] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. *J. Web Sem.*, 8(4):255–270, 2010.

[38] M. Przyjaciel-Zablocki, A. Schätzle, and A. Lange. TriAL-QL: Distributed Processing of Navigational Queries. In *Proc. AMW, Lima, Peru*, volume 1378 of *CEUR Workshop Proceedings*, 2015.

[39] M. Przyjaciel-Zablocki, A. Schätzle, and G. Lausen. TriAL-QL: Distributed Processing of Navigational Queries. In *Proc. of the 18th International Workshop on Web and Databases (WebDB), Melbourne, Australia*, WebDB '15, pages 48–54, 2015.

[40] J. L. Reutter, A. Soto, and D. Vrgoc. Recursion in SPARQL. In *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, pages 19–35, 2015.

[41] K. Rohloff and R. E. Schantz. Clause-iteration with MapReduce to Scalably Query Datagraphs in the SHARD Graph-store. In *Proc. of the 4th International Workshop on Data-intensive Distributed Computing*, DIDC '11, pages 35–44. ACM, 2011.

[42] A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen. PigSPARQL: Mapping SPARQL to Pig Latin. In *Proceedings of the International Workshop on Semantic Web Information Management (SWIM), Athens, Greece*, SWIM'11, pages 4:1–4:8, 2011.

[43] A. Schätzle, M. Przyjaciel-Zablocki, A. Neu, and G. Lausen. Sempala: Interactive SPARQL Query Processing on Hadoop. In *Proceedings of the 13th International Semantic Web Conference (ISWC), Riva del Garda, Italy*, volume 8796 of *Lecture Notes in Computer Science (LNCS)*, pages 164–179, 2014.

[44] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen. S2RDF: RDF Querying with SPARQL on Spark. *Proceedings of the VLDB Endowment (PVLDB)*, 9(10):804–815, 2016.

[45] J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *PVLDB*, 6(14):1906–1917, 2013.

[46] L. Shala. Distributed Processing of RDFPath Queries. M.Sc. thesis, University Freiburg, 2016.

[47] M. Shaw, P. Koutris, B. Howe, and D. Suciu. Optimizing large-scale semi-naïve datalog evaluation in hadoop. In *Datalog in Academia and Industry, Datalog 2.0, Vienna, Austria, September 11-13*, pages 165–176, 2012.

[48] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A Warehousing Solution over a Map-Reduce Framework. *Proc. VLDB Endow.*, 2(2):1626–1629, 2009.

[49] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, and H. E. Bal. WebPIE: A Web-scale Parallel Inference Engine using MapReduce. *J. Web Sem.*, 10:59–75, 2012.

[50] T. White. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (2. ed.)*. O'Reilly, 2011.

[51] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Fast and Interactive Analytics Over Hadoop Data with Spark. *USENIX ;login:*, 34(4):45–51, 2012.