# RDFPath: Path Query Processing on Large RDF Graphs with MapReduce

Martin Przyjaciel-Zablocki, Alexander Schätzle,
Thomas Hornung, and Georg Lausen

Lehrstuhl für Datenbanken und Informationssysteme
Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 51
79110 Freiburg im Breisgau
{zablocki,schaetzl,hornungt,lausen}@informatik.uni-freiburg.de

**Abstract.** The MapReduce programming model has gained traction in different application areas in recent years, ranging from the analysis of log files to the computation of the RDFS closure. Yet, for most users the MapReduce abstraction is too low-level since even simple computations have to be expressed as Map and Reduce phases. In this paper we propose RDFPath, an expressive RDF path query language geared towards casual users that benefits from the scaling properties of the MapReduce framework by automatically transforming declarative path queries into MapReduce jobs. Our evaluation on a real world data set shows the applicability of RDFPath for investigating typical graph properties like shortest paths.

**Keywords:** MapReduce, RDFPath, RDF Query Languages, Social Network Analysis, Semantic Web.

## 1 Introduction

The proliferation of data on the Web is growing tremendously in recent years. According to Eric Schmidt, CEO of Google, more than five Exabyte of data are generated collectively *every two days*, which corresponds to the whole amount of data generated up to the year 2003[1]. Another example is Facebook with currently more than 500 million active users interacting with more than 900 million different objects like pages, groups or events.

In a Semantic Web environment this data is typically represented as a RDF graph [20], which is a natural choice for social network scenarios [21], thus facilitating exchange, interoperability, transformation and querying of data. However, management of large RDF graphs is a non-trivial task and single machine approaches are often challenged with processing queries on such graphs [30]. One solution is to use high performance clusters or to develop custom distributed systems that are commonly not very cost-efficient and also do not scale with respect to additional hardware [1,8,9].

[1] `http://techonomy.com`

The MapReduce programming model introduced by Google in [8] runs on regular off-the-shelf hardware and shows desirable scaling properties, e.g. new computing nodes can easily be added to the cluster. Additionally, the distribution of data and the parallelization of calculations is handled automatically, relieving the developer from having to deal with classical problems of distributed applications such as the synchronization of data, network protocols or fault tolerance strategies. These benefits have led to the application of this programming model to a number of problems in different areas, where large data sets have to be processed [1,2,7]. One line of research is centered around the transformation of existing algorithms into the MapReduce paradigm [19], which is a time consuming process that requires substantial technical knowledge about the framework. More in line with the approach presented in this paper is the idea to use a declarative high-level language and to provide an automatic translation into a series of Map and Reduce phases as proposed in [15,29] for SPARQL and in [24] for Pig Latin, a data processing language for arbitrary data.

**Contributions.** In this paper we present RDFPath, a declarative path query language for RDF that by design has a natural mapping to the MapReduce programming model while remaining extensible. We also give details about our system design and implementation. By its intuitive syntax, RDFPath supports the exploration of graph properties such as shortest connections between two nodes in a RDF graph. We are convinced that RDFPath is a valuable tool for the analysis of social graphs, which is highlighted by our evaluation on a real-world data set based on user profiles crawled from Last.fm. The implementation of RDFPath is available for download from our project homepage[2].

**Related Work.** There is a large body of work dealing with query languages for (RDF) graphs considering various aspects and application fields [6,10,12,17,27,34]. Besides classical proposals for graphs as introduced in [27] and in [17] with RQL, there are also many proposals for specific RDF graph languages (cf. [6,10,12] for detailed surveys). Taking this into account, we extended the proposed comparison matrix for RDF query languages from [4,5] by two additional properties, namely the support for shortest path queries and aggregate functions, as well as the additional RDF query languages SPARQL [26], RPL [34], and RDFPath, as depicted in Table 1. For a more detailed description of the properties occurring in Table 1 the interested reader is referred to [5].

According to Table 1, RDFPath has a competitive expressiveness to other RDF query languages. For the missing diameter property, which is not considered in any of the listed languages, a MapReduce solution has been proposed in [16], regardless of a syntactically useful integration into any path query language. There are also further approaches to extend SPARQL with expressive navigational capabilities such as nSPARQL [25], (C)PSPARQL [3] as well as

[2] `http://dbis.informatik.uni-freiburg.de/?project=DiPoS/RDFPath.html`

**Table 1.** Comparison of RDF Query Languages (adapted from [4,5])

| **Property** | **RQL** | **SeRQL, RDQL**[3]**, Triple** | **N3** | **Versa** | **RxPath** | **RPL** | **SPARQL** | | **RDFPath** |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | **1.0** | **1.1** | |
| **Adjacent nodes** | $\pm$ | $\pm$ | $\pm$ | $\pm$ | $\times$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ |
| **Adjacent edges** | $\pm$ | $\pm$ | $\times$ | $\times$ | $\times$ | $\times$ | $\surd$ | $\surd$ | $\surd$ |
| **Degree of a node** | $\pm$ | $\times$ | $\times$ | $\times$ | $\times$ | $\times$ | $\times$ | $\surd$ | $\surd$ |
| **Path** | $\times$ | $\times$ | $\times$ | $\times$ | $\pm$ | $\pm$ | $\times$ | $\pm$ | $\pm$ |
| **Fixed-length Path** | $\pm$ | $\pm$ | $\pm$ | $\times$ | $\pm$ | $\pm$ | $\surd$ | $\surd$ | $\surd$ |
| **Distance between 2 nodes** | $\times$ | $\times$ | $\times$ | $\times$ | $\times$ | $\times$ | $\times$ | $\times$ | $\pm$ |
| **Diameter** | $\times$ | $\times$ | $\times$ | $\times$ | $\times$ | $\times$ | $\times$ | $\times$ | $\times$ |
| **Shortest Paths** | $\times$ | $\times$ | $\times$ | $\times$ | $\times$ | $\times$ | $\times$ | $\times$ | $\pm$ |
| **Aggregate functions** | $\pm$ | $\times$ | $\times$ | $\times$ | $\pm$ | $\times$ | $\times$ | $\surd$ | $\pm$ |

($\times$: not supported, $\pm$: partially supported, $\surd$: fully supported)

property paths, that are a part of the proposal for SPARQL 1.1[4]. In contrast, we focus on path queries and study their implementation based on MapReduce. A more detailed discussion on SPARQL 1.0 and especially the current SPARQL 1.1 working draft can be found in the appendix.

Another area, which is related to our research, is the distributed processing of large data sets with MapReduce. *Pig* is a system for analyzing large data sets, consisting of the high-level language *Pig Latin* [24] that is automatically translated into MapReduce jobs. Furthermore there are serveral recent approches for evaluating SPARQL queries with MapReduce [15,22,29]. However, because of the limited navigational capabilities of SPARQL [25], as opposed to RDFPath, these approaches do not offer a sufficient functionality to support a broad range of analysis tasks for RDF graphs.

Besides the usage of a general purpose MapReduce cluster, some systems rely on a specialized computer cluster. Virtuoso Cluster Edition [9] is a cluster extension of the RDF Store Virtuoso and BigOWLIM[5] is a RDF database engine with extensive reasoning capabilities, both allowing to store and process billions of triples. In [32] the authors propose an extension of Sesame for querying distributed RDF repositories. However, such specialized clusters have the disadvantage that they require individual infrastructures, whereas our approach is based on a general framework that can be used for different purposes.

**Paper Structure.** Section 2 provides a brief introduction to the MapReduce framework. Section 3 introduces the RDFPath language, while Section 4 discusses the components of the implemented system and the evaluation of RDF-Path queries. Section 5 presents our system evaluation based on a real-world data set and Section 6 concludes this paper with an outlook on future work.

[3] In [14] the authors describe how to extend RDQL to support aggregates.
[4] `http://www.w3.org/TR/sparql11-query`
[5] `http://www.ontotext.com/owlim`

## 2 MapReduce

The MapReduce programming model was originally introduced by Google in 2004 [8] and enables scalable, fault tolerant and massively parallel calculations using a computer cluster. The basis of Google's MapReduce is the distributed file system GFS [11] where large files are split into equal sized blocks, spread across the cluster and fault tolerance is achieved by replication. The workflow of a MapReduce program is a sequence of MapReduce jobs each consisting of a *Map* and a *Reduce* phase separated by a so-called *Shuffle* & *Sort* phase. A user has to implement the *map* and *reduce functions* which are automatically executed in parallel on a portion of the data. The Mappers invoke the map function for every record of their input data set represented as a key-value pair. The map function outputs a list of new intermediate key-value pairs which are then sorted according to their key and distributed to the Reducers such that all values with the same key are sent to the same Reducer. The reduce function is invoked for every distinct key together with a list of all according values and outputs a list of values which can be used as input for the next MapReduce job. The signatures of the map and reduce functions are therefore as follows:

```
map:    (inKey, inValue) -> list(outKey, intermediateValue)
reduce: (outKey, list(intermediateValue)) -> list(outValue)
```

## 3 RDFPath

A RDF data set consists of a set of RDF triples in the form <subject, predicate, object> that can be interpreted as "*subject* has the property *predicate* with value *object*". It is possible to represent a RDF data set as directed, labeled graph where every triple corresponds to an edge (predicate) from one node (subject) to another node (object). For clarity of presentation, we use a simplified RDF notation without URI prefixes in the following. Strings and numbers are mapped to their corresponding datatypes in RDF.

Executing path queries on very large RDF data sets like social network graphs with billions of entries is a non-trivial task that typically requires many resources and computational power [1,8,9,21,30]. RDFPath is a declarative RDF path query language, inspired by XPath and designed especially with regard to the MapReduce model. A query in RDFPath is composed by a sequence of *location steps* where the output of the $i^{th}$ location step is used as input for the $(i+1)^{th}$ location step. Conceptually, a location step adds one or more additional edges and nodes to an intermediate path that can be restricted by filters. The result of a query is a set of paths, consisting of edges and nodes of the given RDF graph. In the following we give an example-driven introduction to RDFPath.

### 3.1 RDFPath By Example

**Start Node.** The start node is the first part of a RDFPath query, separated by `"::"` from the rest of the query and specifies the starting point for the evaluation of a path query as shown in Query 1. Using the symbol `"*"` indicates an arbitrary start node where every subject with the denoted predicate of the first location step is considered (see Query 2).

```
Chris :: knows                                              (1)
    *  :: knows                                             (2)
```

**Location Step.** Location steps are the basic navigational component in RDF-Path, specifying the next edge to follow in the query evaluation process. The usage of multiple location steps, separated by `">"`, defines the order as well as the amount of edges followed by the query (Query 3). If the same edge is used in several consecutive location steps one can use an abbreviation by specifying the number of iterations within parentheses as shown in Query 4. Instead of specifying a fixed edge, the symbol `"*"` can be used to follow an arbitrary edge as illustrated in Query 5 that determines all adjacent edges and nodes of Chris.

```
Chris :: knows > knows > age                                (3)
Chris :: knows (2) > age                                    (4)
Chris :: *                                                  (5)
```

**Filter.** Filters can be specified within any location step using square brackets. There are two types of filters to constrain the value (Queries 6, 7) or the properties (Query 8) of a node reached by the location step. Multiple filters are specified in a sequence and a path has to satisfy all filters. If a node does not have the desired property, the filter evaluates to `false`. Up to now, the following filter expressions are applicable: `equals(), prefix(), suffix(), min(), max()`.

```
Chris :: knows > age [min(18)] [max(67)]                    (6)
Chris :: * > * [equals('Peter')]                            (7)
Chris :: knows [age = min(30)] [country = prefix('D')] > name (8)
```

**Bounded Search.** This type of query starts with a fixed node and computes the shortest paths between the start node and all reachable nodes within a user-defined bound. For this purpose we extend the notation of the previously introduced abbreviations with an optional symbol `"*"`. While the abbreviations indicate a fixed length, the `"*"` symbol indicates to use the number as upper bound for the maximum search depth. As an example, in Query 9 we search for all German people with a maximum distance of three to Chris.

```
Chris :: knows [country = equals('DE')] (*3)                (9)
```

**Bounded Shortest Path.** This type of query computes the shortest path between two nodes in the graph with a user-defined maximum distance. As we are often interested in the length of the path, the query outputs the shortest distance and the corresponding path between two given nodes. To do this one has to extend a bounded search query with a final `distance()` function specifying the target node as shown in Query 10.

`Chris :: knows (*3).distance('Peter')` (10)

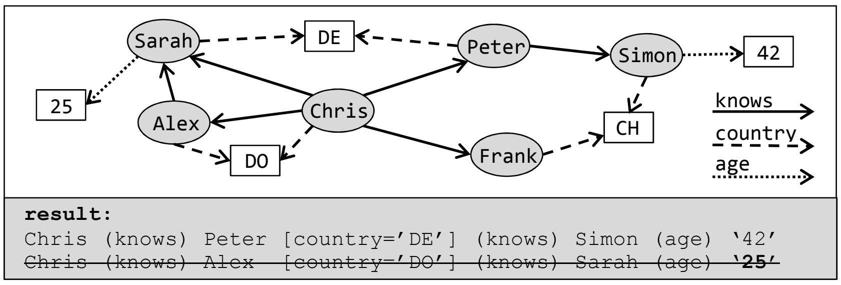
**Aggregation Functions.** It is possible to count the number of resulting paths for a query (Query 11 calculates the degree of Chris) or to apply some aggregation functions to the last nodes of the paths, respectively. The following functions are available: `count()`, `sum()`, `avg()`, `min()` and `max()`. It should be noted that aggregation functions can only be applied to nodes of numeric type (e.g. `integer` or `double`) as shown in Query 12.

`Chris :: *.count()` (11)

`Chris :: knows > age.avg()` (12)

**Example.** Figure 1 shows the evaluation of the last location step of Query 13 on the corresponding RDF graph. The second path is rejected as the age of Sarah does not satisfy the filter condition.

`Chris :: knows [country=prefix('D')] > knows > age [min(30)]` (13)



**result:**
`Chris (knows) Peter [country='DE'] (knows) Simon (age) '42'`
~~`Chris (knows) Alex [country='DO'] (knows) Sarah (age) '25'`~~

**Fig. 1.** RDFPath Example

## 3.2 Expressiveness

In this section we will evaluate the expressiveness of RDFPath w.r.t. the properties listed in Table 1. A detailed discussion can be found on our project homepage[6] and in [28]. Query 5 shows an example for the calculation of all *adjacent edges* and *nodes* of a node by using the symbol `"*"` instead of specifying a fixed edge. Query 11 calculates the *degree of a node* by applying the aggregation function `count()` on the resulting paths and Query 7 gives all *paths with a fixed length* of two from Chris to Peter by specifying two location steps with arbitrary edges. The properties *path*, *distance between 2 nodes* and *shortest paths* are only partially supported by RDFPath because in general to answer these properties one has to calculate paths of arbitrary length where RDFPath only supports paths of a fixed maximum length. Furthermore aggregation functions are partially supported as they can only be applied in the last location step of a query.

[6] `http://dbis.informatik.uni-freiburg.de/?project=DiPoS/RDFPath.html`

## 4 Query Evaluation

We implemented RDFPath based on the well-known *Apache Hadoop Framework*, an open source implementation of Google's MapReduce and GFS. Our system loads the considered RDF graph into the Hadoop Distributed File System (HDFS) once in advance, translates RDFPath queries into a sequence of MapReduce jobs, executes them in the framework and stores the results again in HDFS. A location step in RDFPath mostly follows a fixed edge (predicate) which means that only a portion of the RDF graph has to be considered in many cases. In these cases, it is advantageous to read only those triples concerning the relevant edge which can be achieved by partitioning the triples of the RDF graph according to their predicates. This principle is also knows as *vertical partitioning* [2] and forms the basis of our data model. Hence an input RDF graph is loaded in advance to apply vertical partitioning and store resulting partitions in HDFS. Certainly, in the case of a not fixed edge (`"*"` symbol), all partitions must be considered and we cannot benefit from this strategy.

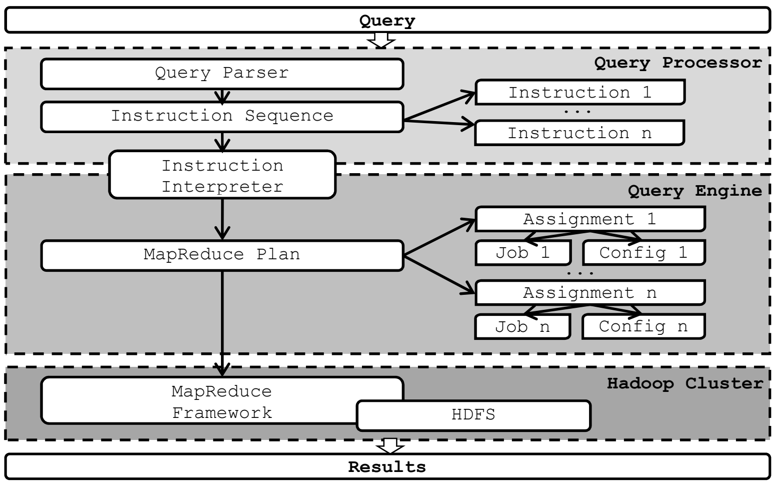


**Fig. 2.** Query Processing

The Query Processor parses the query and generates a general execution plan that consists of a sequence of instructions where each instruction describes e.g. the application of a filter, join or aggregation function. In the next step, the general execution plan is mapped to a specific MapReduce plan that consists of a sequence of MapReduce assignments. An assignment encapsulates the specific MapReduce job together with a job configuration. The Query Engine runs the MapReduce jobs in sequence, collects information about the computation process like time and storage utilization and cleans up temporary files. A schematic representation of this procedure is shown in Figure 2.

Fault-tolerance, i.e. relaunching failed tasks, is managed by the Hadoop framework automatically. Currently there are no join-ordering optimizations implemented to determine the best join execution order as proposed in [32], for example. A query in RDFPath is processed sequentially from left to right. Although queries in RDFPath do not need to have a fixed start node, this is often advisable as arbitrary start nodes dramatically increase the number of intermediate results in general. For queries with a fixed start node, a processing order from left to right can often cut down the costs for processing a sequence of joins, as it usually corresponds to the most selective join-order. In other cases, e.g. fixed start node combined with fixed end node for computing shortest paths, it is likely that join-ordering optimizations could have a significant impact on query evaluation time and space [32].

## 4.1 Mapping of Location Steps to MapReduce Jobs

A query in RDFPath is composed of a sequence of location steps that is translated into a sequence of MapReduce jobs automatically. As illustrated in Figure 3 a location step corresponds to a join in MapReduce between an intermediate set of paths and the corresponding RDF graph partition. Joins are implemented as so-called *Reduce-Side-Joins* since the assumption of the more efficient *Map-Side-Joins* that both inputs must be sorted is not fulfilled in general. The principles of Reduce-Side-Joins can be looked up in [19,33]. Filters are applied in the Map phase by rejecting all triples that do not satisfy the filter conditions and aggregation functions are computed in parallel in the Reduce phase of the last location step. The computation of shortest paths is based on a parallel breadth-first search approach and requires at most one additional MapReduce job for selecting shortest paths. This selection is usually applied in the subsequent location step. If there is no subsequent location step, an additional MapReduce job becomes necessary. We also implemented a mechanism to detect cycles when extending an intermediate path where the user can decide at runtime whether (1) cycles are allowed, (2) only allowed if the cycle contains two or more distinct edges or (3) not allowed at all. Considering Figure 3 the given query requires two joins and is therefore mapped into a MapReduce plan that consists of two MapReduce jobs. While the first job computes all friends of "Chris" that can be reached by following the edge `knows` at most two times, the second MapReduce job follows the edge `country` and restricts the value to "DE".

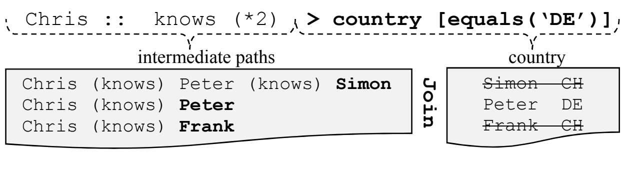


**Fig. 3.** Joins and Location Steps

## 5 Evaluation

We evaluated our implementation on two different data sources to investigate the scalability behavior. First, we used artificial data produced by the SP$^2$Bench generator [31] which allows to generate arbitrary large RDF documents that contain bibliographic information about synthetic publications. The generated RDF documents contained up to 1.6 billion RDF triples. Second, we collected 225 million RDF triples of real world data from the online music service Last.fm that are accessible via a public API. Due to space limitations we only discuss some results for the Last.fm dataset, which is a more appropriate choice for path queries and can also be interpreted in a more intuitive way. Figure 4 illustrates the dataset.

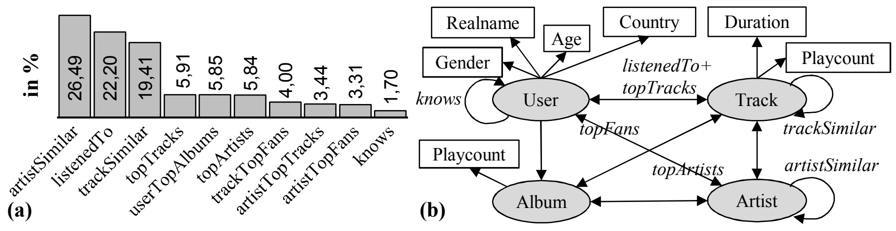


**Fig. 4.** (a) Histogram of Last.fm data (b) Simplified RDF graph of the Last.fm data set. The missing edge labels are named like the target nodes. In the case of ambiguity, the edge label is extended by the type of source (e.g. `trackPlaycount` and `albumPlaycount`).

We used a cluster of ten Dell PowerEdge R200 servers connected via a gigabit network and Cloudera's Distribution for Hadoop 3 Beta (CDH3). Each server had a Dual Core 3,16 GHz processor, 4 GB RAM and 1 TB harddisk. One of the servers was exclusively used to distribute the MapReduce jobs (Jobtracker) and store the metadata of the file system (Namenode). Query 1 to 3 were evaluated on a fixed cluster size of 9 nodes with varying dataset sizes, whereas Query 4 and 5 used a fixed dataset size of 200 million triples while varying the number of cluster nodes.

**Query 1.** Starting from a given track this query determines the album name for all similar tracks that can be reached by following the edge `trackSimilar` at most four times. The overall execution times of this query are shown in the left diagram of Figure 5 and exhibit a linear scaling behavior in the size of the graph. Furthermore it turns out that this is also the case for the amount of transferred data (SHUFFLE), intermediate data (LOCAL) and data stored in HDFS. These values are shown in the right diagram of Figure 5. We conclude that the execution time is mainly influenced by the number of intermediate results stored locally as well as the transferred data between the machines.

**Query 2.** Starting from all tracks of Michael Jackson that are on the album "Thriller" the query determines all similar tracks that have a minimum duration

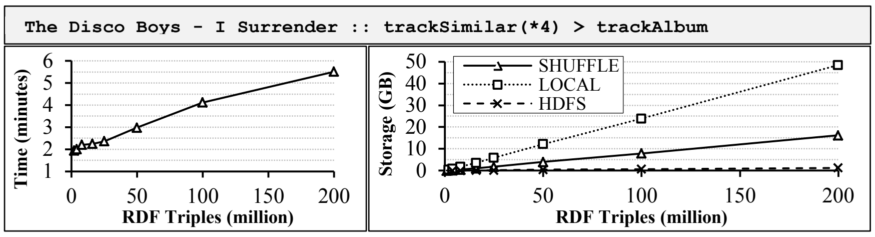


**Fig. 5.** Query 1

of 50 seconds. The last location step then looks for the top fans of these tracks who live in Germany. The idea behind this query was to have a look at the impacts of using filters to reduce the amount of intermediate results. The number of results to the query and therefore the used HDFS storage do not increase significantly with the size of the graph as the tracks of the album "Thriller" are fixed. This also explains the execution times of the query as illustrated in the left diagram of Figure 6 and confirms that the execution time is mainly determined by the amount of intermediate results.

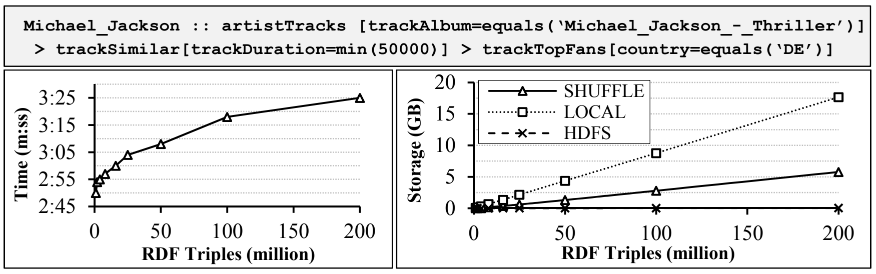


**Fig. 6.** Query 2

**Query 3.** These queries determine the friends of Chris reached by following an increasing number of edges. The first query starts by following the edge `knows` once and the last query ends by following the edge `knows` at most ten times. This corresponds to the computation of the *Friend of a Friend*[7] paths starting from Chris with an increasing maximum distance. The left chart of Figure 7 illustrates the percentage of reached people, in accordance to the maximum Friend of a Friend distance, where the total percentage represents all reachable people. Starting with a fixed person we can reach over 98% of all reachable persons by following the edge `knows` seven times which corresponds to the well-known *six degrees of separation* paradigm [18]. The right chart of Figure 7 shows the execution times depending on the maximum Friend of a Friend distance. We

[7] `http://www.foaf-project.org`

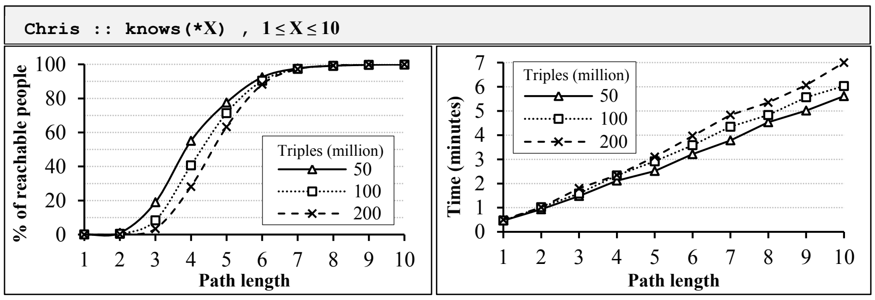


**Fig. 7.** Query 3

can observe a linear scaling behaviour that is mainly determined by the number of joins rather than computation and data transfer time.

**Query 4. & 5.** Query 4 is a kind of recommendation query that gives, for every user, those tracks that are similar to the tracks the user listened to. On the other hand, Query 5 is an analytical query where we want to know, for every artist, where the top Fans of a similar artist come from. The execution times of both queries for a fixed input size of 200 million triples and a variable number of nodes in the cluster are shown in the left diagram of Figure 8. We can observe that the overall execution times for Query 4 as well as for Query 5 improve with the number of nodes but the benefit of an additional node decreases continuously which is an expectable behaviour of the MapReduce framework. The storage utilization for both queries is given in the right diagram of Figure 8. The values for different numbers of nodes were almost equal in size, with a maximum deviation of 1% for Shuffle & HDFS storage and 9% for local storage.

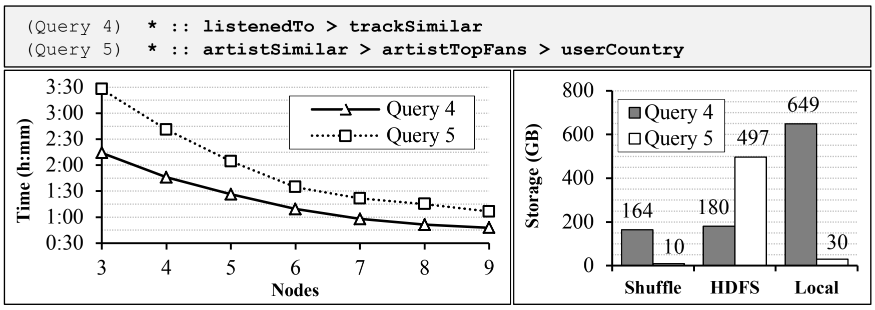


**Fig. 8.** Query 4 & 5

**Results.** Our evaluation shows that RDFPath allows to express and compute interesting graph issues such as Friend of a Friend queries, small world properties like six degrees of separation or the *Erdös number*[8] on large RDF graphs.

[8] `http://www.oakland.edu/enp`

The execution times for the surveyed queries on real-world data from Last.fm scale linear in the size of the graph where the number of joins as well as the amount of data, that must be stored (local/HDFS) and transferred over the network, determine the complexity of a query. Taking this into account it is promising to observe an almost constant storage utilization with an increasing number of nodes. On the other hand, adding additional nodes can improve the overall executions time significantly, which shows that RDFPath benefits from the horizontal scaling properties of MapReduce.

## 6 Conclusion

The amount of available Semantic Web data is growing constantly, calling for solutions that are able to scale accordingly. The RDF query language RDFPath, that is presented in this paper, was designed with this constraint in mind and combines an intuitive syntax for path queries with an effective execution strategy using MapReduce. Our evaluation confirms that both large RDF graphs can be handled while scaling linear with the size of the graph and that RDFPath can be used to investigate graph properties such as a variant of the famous six degrees of separation paradigm typically encountered in social graphs.

As future work we plan to extend RDFPath with more powerful language constructs geared towards the analysis of social graphs, e.g. to express the full list of desiderata stated in [21]. In parallel, we are optimizing our implementation on the system level by incorporating current results for the efficient computation of joins with MapReduce [7,23].

## References

1. Abadi, D.J.: Tradeoffs between Parallel Database Systems, Hadoop, and HadoopDB as Platforms for Petabyte-Scale Analysis. In: Gertz, M., Ludäscher, B. (eds.) SSDBM 2010. LNCS, vol. 6187, pp. 1–3. Springer, Heidelberg (2010)
2. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: VLDB, pp. 411–422 (2007)
3. Alkhateeb, F., Baget, J.F., Euzenat, J.: Extending sparql with regular expression patterns (for querying rdf). J. Web Sem. 7(2), 57–73 (2009)
4. Angles, R., Gutierrez, C.: Querying RDF Data from a Graph Database Perspective. In: Gómez-Pérez, A., Euzenat, J. (eds.) ESWC 2005. LNCS, vol. 3532, pp. 346–360. Springer, Heidelberg (2005)
5. Angles, R., Gutierrez, C., Hayes, J.: RDF Query Languages Need Support for Graph Properties. Tech. Rep. TR/DCC-2004-3, University of Chile (June 2004)
6. Bailey, J., Bry, F., Furche, T., Schaffert, S.: Web and Semantic Web Query Languages: A Survey. In: Eisinger, N., Małuszyński, J. (eds.) Reasoning Web. LNCS, vol. 3564, pp. 35–133. Springer, Heidelberg (2005)
7. Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y.: A Comparison of Join Algorithms for Log Processing in MapReduce. In: SIGMOD Conference, pp. 975–986 (2010)
8. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI, pp. 137–150 (2004)

9. Erling, O., Mikhailov, I.: Towards Web Scale RDF. In: Proc. SSWS (2008)
10. Furche, T., Linse, B., Bry, F., Plexousakis, D., Gottlob, G.: RDF Querying: Language Constructs and Evaluation Methods Compared. In: Barahona, P., Bry, F., Franconi, E., Henze, N., Sattler, U. (eds.) Reasoning Web 2006. LNCS, vol. 4126, pp. 1–52. Springer, Heidelberg (2006)
11. Ghemawat, S., Gobioff, H., Leung, S.T.: The Google File System. In: Proc. SOSP, pp. 29–43 (2003)
12. Haase, P., Broekstra, J., Eberhart, A., Volz, R.: A Comparison of RDF Query Languages. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 502–517. Springer, Heidelberg (2004)
13. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. W3C Working Draft (May 2011), `http://www.w3.org/TR/sparql11-query/`
14. Hung, E., Deng, Y., Subrahmanian, V.S.: RDF Aggregate Queries and Views. In: ICDE, pp. 717–728 (2005)
15. Husain, M.F., Khan, L., Kantarcioglu, M., Thuraisingham, B.: Data Intensive Query Processing for Large RDF Graphs Using Cloud Computing Tools. In: Proc. CLOUD, pp. 1–10. IEEE (2010)
16. Kang, U., Tsourakakis, C.E., Faloutsos, C.: PEGASUS: A Peta-Scale Graph Mining System. In: ICDM, pp. 229–238 (2009)
17. Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M.: RQL: A Declarative Query Language for RDF. In: WWW, pp. 592–603 (2002)
18. Leskovec, J., Horvitz, E.: Planetary-Scale Views on a Large Instant-Messaging Network. In: Proc. WWW 2008, pp. 915–924 (2008)
19. Lin, J., Dyer, C.: Data-intensive text processing with MapReduce. Synthesis Lectures on Human Language Technologies 3(1), 1–177 (2010)
20. Manola, F., Miller, E.: RDF Primer (2004), `http://www.w3.org/TR/rdf-primer/`
21. Martín, M.S., Gutierrez, C.: Representing, Querying and Transforming Social Networks with RDF/SPARQL. In: Aroyo, L., Traverso, P., Ciravegna, F., Cimiano, P., Heath, T., Hyvönen, E., Mizoguchi, R., Oren, E., Sabou, M., Simperl, E. (eds.) ESWC 2009. LNCS, vol. 5554, pp. 293–307. Springer, Heidelberg (2009)
22. Myung, J., Yeon, J., Lee, S.: SPARQL Basic Graph Pattern Processing with Iterative MapReduce. In: Proc. MDAC 2010, pp. 1–6. ACM (2010)
23. Okcan, A., Riedewald, M.: Processing Theta-Joins using MapReduce. In: SIGMOD Conference, pp. 949–960 (2011)
24. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: A Not-So-Foreign Language for Data Processing. In: SIGMOD, pp. 1099–1110 (2008)
25. Pérez, J., Arenas, M., Gutierrez, C.: nSPARQL: A navigational language for RDF. In: Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T., Thirunarayan, K. (eds.) ISWC 2008. LNCS, vol. 5318, pp. 66–81. Springer, Heidelberg (2008)
26. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Trans. Database Syst. 34(3) (2009)
27. Pratt, T.W., Friedman, D.P.: A Language Extension for Graph Processing and Its Formal Semantics. Commun. ACM 14(7), 460–467 (1971)
28. Przyjaciel-Zablocki, M.: RDFPath: Verteilte Analyse von RDF-Graphen. Master's thesis, Albert-Ludwigs-Universität Freiburg (2010)
29. Schätzle, A., Przyjaciel-Zablocki, M., Lausen, G.: PigSPARQL: Mapping SPARQL to Pig Latin. In: Proceedings of the International Workshop on Semantic Web Information Management, SWIM 2011, pp. 4:1–4:8. ACM (2011)

30. Schmidt, M., Hornung, T., Küchlin, N., Lausen, G., Pinkel, C.: An Experimental Comparison of RDF Data Management Approaches in a SPARQL Benchmark Scenario. In: Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T., Thirunarayan, K. (eds.) ISWC 2008. LNCS, vol. 5318, pp. 82–97. Springer, Heidelberg (2008)
31. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP2Bench: A SPARQL Performance Benchmark. In: ICDE, pp. 222–233 (2009)
32. Stuckenschmidt, H., Vdovjak, R., Broekstra, J., Houben, G.J.: Towards distributed processing of RDF path queries. Int. J. Web Eng. Technol. 2(2/3), 207–230 (2005)
33. White, T.: Hadoop: The Definitive Guide, 1st edn. O'Reilly (2009)
34. Zauner, H., Linse, B., Furche, T., Bry, F.: A RPL Through RDF: Expressive Navigation in RDF Graphs. In: Hitzler, P., Lukasiewicz, T. (eds.) RR 2010. LNCS, vol. 6333, pp. 251–257. Springer, Heidelberg (2010)

## A Comparison with SPARQL

SPARQL 1.0 is the W3C recommended query language for RDF. Compared with RDFPath, we can note the following interesting relations: First of all, it is important to mention that SPARQL 1.0 is a general purpose query language, designed for a wide range of analysis tasks, whereas RDFPath focuses on expressing path queries. Thus, SPARQL 1.0 provides only limited navigation capabilities (e.g. no abbreviation for following the same edge). Furthermore, both approaches differ in the kind of output. The result of a query in RDFPath is a set of paths in contrast to a set of variable mappings in SPARQL. Next, SPARQL 1.0 does not support aggregate functions and shortest path expressions, whereas RDFPath supports both.

The SPARQL 1.1 [13] working draft addresses, among other things, some of the issues mentioned above: Property paths, introduced with SPARQL 1.1, add support for navigational queries similar to RDFPath but with some additional features like inverse, negated and alternative paths. They also allow to abbreviate occurrences of edges in more detail and follow paths of arbitrary length. However, property paths do not provide access to the whole path, but only to the first and last node. Accordingly, the result in SPARQL 1.1 is projected to the variables given in the query, i.e. the first and the last node of a path. In contrast, RDFPath provides access to the whole path in such a way that it is possible to express filter conditions on arbitrary location steps and to emit whole paths with all intermediate steps as output. Furthermore, predicates in property paths are always fixed, i.e. variable edges are not expressible. RDFPath provides a possibility to follow arbitrary edges and supports three types of cycle treatment. Expressing a query that determines the shortest path between two nodes is also not possible with property paths. Certainly, SPARQL 1.1 allows multiple property path expressions in one query, whereas queries in RDFPath must be composed of a single sequence of location steps. The following two examples illustrate how queries could be expressed with SPARQL 1.0, SPARQL 1.1 and RDFPath, despite of the different kind of output.

| **Example 1.** | *Friend of a Friend query starting with 'Allen'* |
|---|---|
| SPARQL 1.0 | `SELECT ?tmp1, ?tmp2, ?tmp3, ?tmp4, ?tmp5`<br>`WHERE { Allen knows ?tmp1 . ?tmp1 knows ?tmp2 .`<br>`        ?tmp2 knows ?tmp3 . ?tmp3 knows ?tmp4 .`<br>`        ?tmp5 knows ?tmp5 }` |
| SPARQL 1.1 | `SELECT  ?tmp { Allen knows{5} ?tmp }` |
| RDFPath | `Allen :: knows(5)` |

| **Example 2.** | *Friend of a Friend query with two path restrictions* |
|---|---|
| SPARQL 1.0<br>&<br>SPARQL 1.1 | `SELECT ?tmp1, ?tmp2, ?age`<br>`WHERE { Allen knows ?tmp1 . ?tmp1 age ?age .`<br>`        FILTER (?age >= 20)`<br>`        ?tmp1 knows ?tmp2 . ?tmp2 country 'DE'}` |
| RDFPath | `Allen :: knows[age=min(20)] > knows[country=equals('DE')` |