On Combinators and Single Source Context-Free Path Querying

Mikhail Nikilukin
Inria Paris-Rocquencourt
Rocquencourt, France
trovato@corporation.com

Ekaterina Verbitskaia
The Thørväld Group
Hekla, Iceland
larst@affiliation.org

Semyon Grigorev
Rajiv Gandhi University
Doimukh, Arunachal Pradesh, India
larst@affiliation.org

ABSTRACT

A clear and well-documented LaTeX document is presented as an article formatted for publication by ACM in a conference proceedings or journal publication. Based on the "acmart" document class, this article presents and explains many of the common variations, as well as many of the formatting elements an author may use in the preparation of the documentation of their work.

CCS CONCEPTS

• Information systems → Graph-based database models; **Query languages for non-relational engines**; **• Theory of computation →** *Grammars and context-free languages*; **• Software and its engineering →** *Functional languages.*

KEYWORDS

Graph Database, Context-Free Path Querying, Parser Combinators, Single-Source Path Querying, CFPQ, Language Constarined Path Querying

ACM Reference Format:
Mikhail Nikilukin, Ekaterina Verbitskaia, and Semyon Grigorev. 2018. On Combinators and Single Source Context-Free Path Querying. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Context-Free path querying (CFPQ) is an actively developed area in graph datatbase analysis. CFPQ is widely used for static code analysis [?], RDF querying [?], biological data analysis [?].

While lots of research aimed to CFPQ evalustion algorithm develomplent [?], languages which supports context-free constraints specification are not investigated anough. In our knolage, only extension for Sparql CfSparql [?] supports context-free constarints. There is also proposal for Cypher[1] which is not implemneted yet. So, ways to envolve context-free constraints for graph querying should be investigated.

Note, that graph analysis often is only a part of more complex solution. So, graph query languages should be integrated with general-purpose programming languages. Typing [1].

[1]!!!

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

Combinators can solve these problems EV!!! [2].

Single source scenario instead of traditional all pairs. Also useful. For manual data analysis. Some of algorithms inheritantly calculate only all pairs reachability.

In this paper we make the following contributions.

- Introduce example and show how to use combinators for context-free path querying. We demonstarte main features of combinator-based approach such as type-safety, flexibility (compositionality and generics), IDE support and user-defined actions.
- We evaluate single source context-free path querying on some real-world RDFs. We find that the case when number of paths in answer in big, but length of these paths is relatevely small is the main case in classical RDF context-free queryes. And we show thst in this case single-source CFPQ can be evaluated in reasonable time and space. Also our evaluation demonstrates that detailed analysis of theoretical time and space complexity of CFPQ algorithms id required.

2 COMBINATORS FOR CONTEXT-FREE PATH QUERYING

In this section we demonstrate main features of combinators in the context of context-free path querying and integration with general-purpose programming languages. To do it we first introduce a simple graph analysis problem and then show how to solve it by using parser combinators In our wirk we use Merrkst.Graph combinators library.

2.1 Problem Statement

Suppose we have an RDF graph and whatn to analize hierarchical dependencies over different types of relations. Our goal is for fhe given object to find all objects which lies on the same lavel of hierarchy!!! !!!!!

2.2 Simple Solution

```
val rName = "skos__narrowerTransitive"
def qSameGen () =
    syn(inE((_: Entity).label() == rName) ~ qSameGen().? ~
        outE((_: Entity).label() == rName))
```

This query specifies exactly path we want, but still not a solution. First of all, we can not specify start vertex and can not extract final vertices. Also, this query for one specified relation. If we want to investigate hierarchy over othe relations, we need to rewrite this query.

2.3 Compositionality

First step is a generalization of the same generation query to simplfy handling of different types of relations. To do it we introduce a helper function `reduceChoice` which takes a list of subqueryes and combine them by using alternation operation.

```
def reduceChoice(qs: List[_]) = {
  qs match {
     case x :: Nil => x
     case x :: y :: qs => syn(qs.foldLeft(x | y)(_ | _))
  }
}
```

After that we use this function in new version of `sameGen` to combine subqueryes for different types of braces. To make it possible to use differetn types of braces without query rewriting we pass barces as a parameter.

```
def sameGen(brs: List[(_,_)]) =
    reduceChoice( brs.map {
      case (lbr, rbr) => syn(lbr ~ sameGen(brs).? ~ rbr)
      })
```

Now we are ready to provide ability to specify start vertex and collect information of final vertices. First of all, we provide a filter to select only vertices with `uri` property.

```
val uriV = syn(V((_: Entity).hasProperty("uri")) ^^)
```

After thet we create a function which takes two parameters, start vertex and a path query, and create a new query to find all vertices with `uri` property which are reachcbl from the specified strat vertex by specified path. Finally we collect values of `uri` for all reachable vertices. To do it we specify user-defined action `{case _ ~ _ ~ (v: Entity) => v.getProperty[String]("uri")}` which captures result of query (it is a triple-sequence of subqueryes results) and gets the `uri` property form result of last subquery.

```
def queryFromV (startV, query) =
  syn(startV ~ query ~ uriV &
      {case _ ~ _ ~ (v: Entity) =>
             v.getProperty[String]("uri")})
```

2.4 User-Defined Actions and Advanced Results Processing

Final step is to extend the query with calculation of lengths of all paths which satisfie conditions. To do it we equip `sameGen` query with additional user-defined actions.

```
def sameGen(brs: List[(_,_)]) =
    reduceChoice(
      brs.map {
        case (lbr, rbr) =>
          syn((lbr ~ (sameGen(brs).?) ~ rbr) & {
            case _~Nil~_ => 2
            case _~((x:Int)::Nil)~_ =>  x + 2
          })
      })
```

Top level query function now handles not only theard element, but also the second one in order to get access to accamulated lenghts of paths.

```
def queryFromV(startV, query) =
    syn(startV ~ query ~ uriV &
      {case _ ~ (len:Int) ~ (v:Entity) =>
        (len, v.getProperty[String]("uri"))})

def makeBrs (brs:List[_]) =
    brs.map(name =>
       (syn(inE((_: Entity).label() == name) ^^),
        syn(outE((_: Entity).label() == name) ^^)))
      .toList

def runExample (brs: List[_], startVId, graph) =
    val startV = V(getIdFromNode(_: Entity) == startVId
    executeQuery(queryFromV( syn(startV)^^),
                                  sameGen(makeBrs(brs))),
                  graph).toList

runExample(RdfConstants.RDFS__SUB_CLASS_OF :: Nil, 1, graph)
```

2.5 Type Safety

If subqueryes are composed incorrectly, then

In example showed in figure 1, elements of pair wich represents query result are used incorrectly: we want to find total length of all paths but sum final vertices' identifiers instead of lenghths. As a result, compiler statically detect a error because integer expected instead of string.

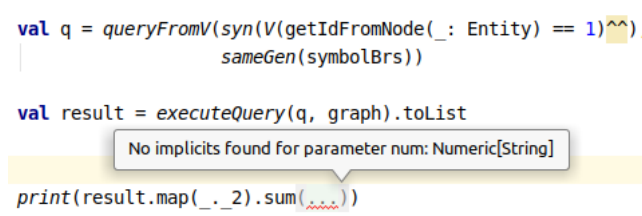


Figure 1: !!!

2.6 IDE Support

Since you can use IDE for development, you get all features for qury development, such as syntah highkighting, code navigation, autocompletion, without any additional effor. An example of autocompletion suggestions for vertex is presented in figure 2.

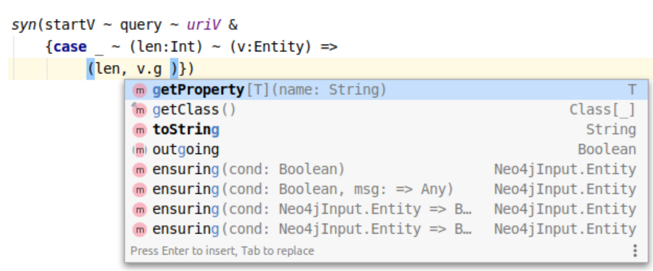
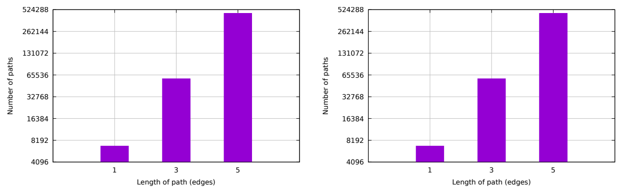


Figure 2: !!!!

3 EVALUATION

We evaluate Meerkat.Graph on single source context-free path querying scenario. For evaluation we use Neo4j graph databese which was run on PC with the folloeing configuration.

- CPU
- RAM
- OS
- JVM



(a) Enzime (b) Geospecies

Figure 3: Paths length destribution

Neo4j is integreted into application !!!!

Dataset contains two real-world RDFs: Geospecies which contains information about biological hierrarchy[2] and Enzime which is a part of UniProt database[3]. Detailed description of these graphs is presented in table 1. Note, that graphs was loaded into database fully, not only edges which laballed by relations used inqueryes.

Graph	#Vertices	#Edges	#NT	#BT
Enzime				
Geospecies				

Table 1: Details of graphs

Queries for evaluation are versions of same-generation query — classical context-free query which is useful for hierarchy analysis. We equip queryes with user-defined actions for end verties saving, paths length calcualtion and unique path counting. To demonstarte power of combinators, we use the function !!! defied above to create queries.

For each graph and each query we run this query form each vertex from graph and measure elapsed time and required memory by using !!! tool. Note, that mesured memory is allocated by JVM, not really used.

Enzime RDF querying. We evaluate two queryes: Q_1 — same genaration over !!!! relation

```
def q1 (startV) =
    val q =
        sameGen(makeBrs(RdfConstants.RDFS__SUB_CLASS_OF ::
                        RdfConstants.RDF__TYPE :: Nil))
    queryFromV(startV, q)
```

and Q_2 — same generation over !!!

```
def sameGen(brs) =
  reduceChoice(
    brs.map {case (lbr, rbr) =>
      lbr ~ syn(sameGen(brs).?) ~ rbr})
```

.

Results of evaluation are presented in figures 4 and 5. Also we collect paths length destribution which is showed in figure 3. We can see that prvided datasets contain relatively short paths which satisfie queryes.

Figure 4 shows dependency of query evaluation time on query answer size in terms of number of edge-different !!! paths. First of

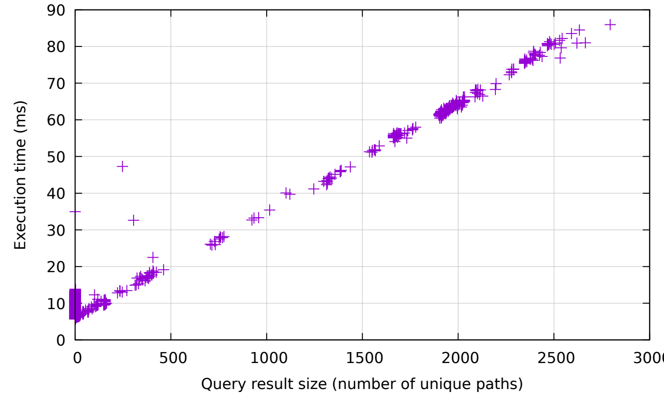


Figure 4: Query execution time for Enzime dataset and queryes Q_1 and Q_2

all, we can see that evaluation time is linear on answer size. Also we can see, that time which requred to evaluate query for one specific vertex is relatively small. In our case it is less than 90ms.

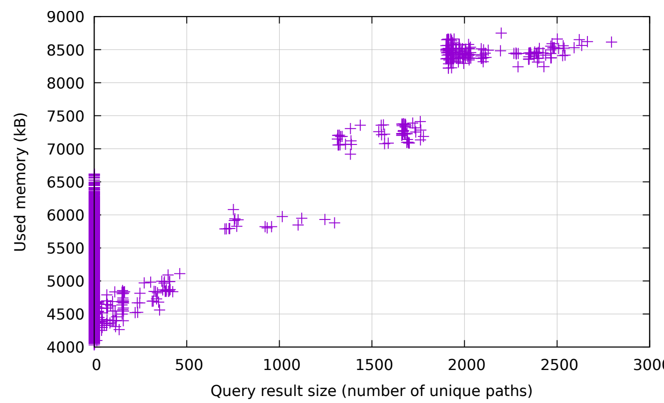


Figure 5: Query required memory for Enzime dataset and queryes Q_1 and Q_2

Figure 5 shows dependency of memory requred to evaluate qurey on query answer size in terms of number of uniqie paths.

Geospecies RDF querying.

Here we can see !!!!

Finally, we can conclude that confext-free path querying in single source scenario can be efficiently evaluated by using !!! in case when number of paths in answer is big but its length is relatively small. While all pairs scenario is still hard [**?**], single source scenarion, which is useful for manual or interactive data analysis, can be !!! Also we can see that while theoretical time and space complexity of CFPQ algoritms at leas cubic, in demonstrated scenario real execution time and required memory is linear. So, it is necessary to provide detailed time and space complexity analysis of algorithms.

[2] https://old.datahub.io/dataset/geospecies. Access date: 12.11.2019.

[3] Protein sequences data base: https://www.uniprot.org/. RDFs with data are available here: ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/rdf. Access date: 12.11.2019

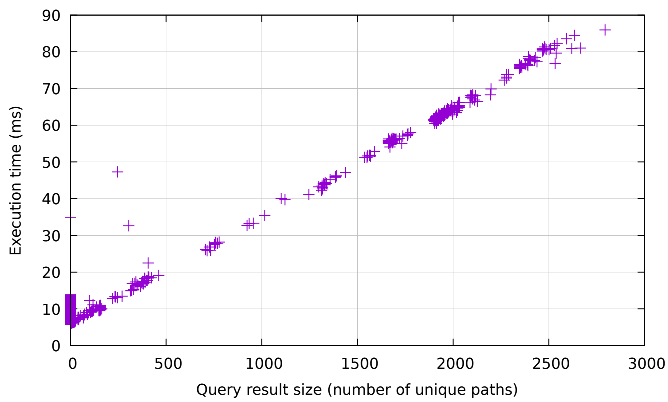


Figure 6: Query execution time for Enzime dataset and queryes Q_3 and Q_4

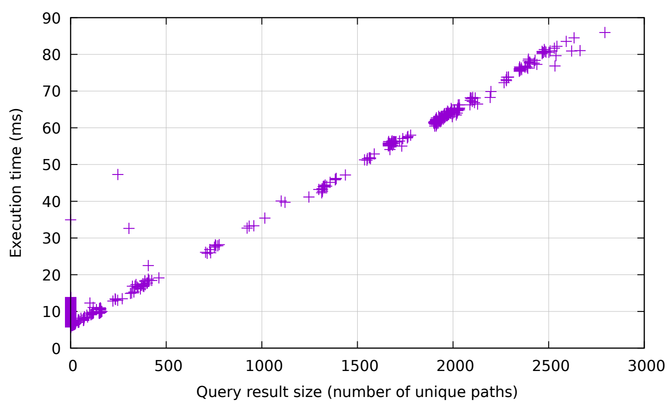


Figure 7: Query execution time for Enzime dataset and queryes Q_3 and Q_4

4 CONCLUSION AND FUTURE WORK

We show that single-source context-free path querying can be !!! We demonstrate a combinator-based approach implemented in Meerkat.Graph Scala library, but this approach can be implemented in almost any high-level programing language. While combinators is a very powerful way to specify context-free queries, it may seem hard to understand for many users. There are other algorithms for context-free path queries which should be applicable for single-source path querying and we hope that they can be integrated with the existing graph database in a more convenient way. But it is necessary more research in this direction.

We should investigate wore datasets to detect other shapes of query results. For example, we should investigate the behavior of single-source querying in the case when a number of resulting paths is small, but paths are relatively long. And the first question is which data analysis tasks lead to this scenario.

One of important direction of the future reserach is to optimize performance of proposed solution. One of possible solution is deep integration with Neo4j infrastructure to utilize cache system.

Another direction is combinators library improvement. First of all, it is necessary to make cimbinators syntax more user-friendly. Also, it is necessary to create set of query templates (see same-generation template).

ACKNOWLEDGMENTS

To Robert, for the bagels and explaining CMYK and color spaces.

REFERENCES

[1] Norbert Tausch, Michael Philippsen, and Josef Adersberger. 2011. A Statically Typed Query Language for Property Graphs. In *Proceedings of the 15th Symposium on International Database Engineering & Applications* (Lisboa, Portugal) *(IDEAS '11)*. Association for Computing Machinery, New York, NY, USA, 219–225. https://doi.org/10.1145/2076623.2076653

[2] Ekaterina Verbitskaia, Ilya Kirillov, Ilya Nozkin, and Semyon Grigorev. 2018. Parser Combinators for Context-Free Path Querying. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala* (St. Louis, MO, USA) *(Scala 2018)*. Association for Computing Machinery, New York, NY, USA, 13–23. https://doi.org/10.1145/3241653.3241655