

On Combinators and Single Source Context-Free Path Querying

Mikhail Nikilukin
National Research University Higher
School of Economics
Moscow, Russia
michael.nik999@gmail.com

Ekaterina Verbitskaia
JetBrains Research
St. Petersburg, Russia
ekaterina.verbitskaia@jetbrains.com

Semyon Grigorev
s.v.grigoriev@spbu.ru
semyon.grigorev@jetbrains.com
Saint Petersburg State University
St. Petersburg, Russia
JetBrains Research
St. Petersburg, Russia

ABSTRACT

Efficient context-free path querying algorithms development and its evaluation in different cases, in couple with design and development of graph query languages which support context-free constraints and its transparent integration into general-purpose languages are areas of active research. In our work we, first, explain how to use parser-combinators for context-free path querying and demonstrate how this approach can solve such problems as transparent integration with general-purpose language, type safety, composability, user-defined actions handling, and development environment support by presenting step-by-step example. Second, we evaluate combinator-based query execution procedure on two real-world RDFs in single source case, and show that, first, combinators are applicable for real-world single source CFPQ specification and processing, second, detailed analysis of single source CFPQ is required.

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 Constrained 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, 6 pages. <https://doi.org/10.1145/1122445.1122456>

1 INTRODUCTION

Context-Free Path Querying (CFPQ) is an actively developed area in graph database analysis. CFPQ is widely used for static code analysis [9, 10, 18], RDF querying [8, 17], biological data analysis [12].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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

© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...\$15.00
<https://doi.org/10.1145/1122445.1122456>

Most of research focus on developing algorithms for CFPQ evaluation [1, 3, 4, 8, 12, 16], whereas specification languages for support context-free are not investigated enough. Best to our knowledge, only one extension for Sparql supports context-free constraints: cfS-PARQL [17]. There is also a proposal for CFPQ as a part of Cypher¹ language, but there is no implementation for it yet. We believe that more research should be conducted on the specification languages for context-free constraints in graph querying.

It is worth noting that graph analysis is often only a part of a more complex system, usually implemented in a general-purpose language. Since a graph query language is unsuitable to implement a whole system, there should be means of integration of them into general-purpose programming languages. There are many ways to integrate them ranging from creating graph queries from string values of a general-purpose language to implementing a special embedded domain specific language to more sophisticated.

Although simple, the string manipulating approach does not provide a developer with any safety guarantees. There is no way to ensure that a string generated by an application is a valid query or, in case it is not, to provide any feedback. This makes string manipulating technique error prone, the code — unclear and hard to maintain.

Safety of an embedded DSL entirely depends on its implementation. Some general-purpose languages with powerful type systems (such as HASKELL, OCAML or SCALA) or the ones supporting hygienic macros (such as SCHEME or RUST) facilitate creating safe and reliable DSLs. Still, they typically lack full support of a development environment: it may be harder to debug queries or issues with composability can arise.

There is a general trend towards imposing more restricting type systems on programming languages. Among many others are typing annotations for PYTHON and TYPESCRIPT code and nullability checks in KOTLIN. Typing graphs and query languages improves readability and simplifies maintenance [13].

Parser combinators are the answer to the integration of parsing into a general-purpose programming language. Recursive descend parsers are encoded as functions of the host language, while grammar constructions such as sequencing and choice are implemented as higher-order functions. This idea was first introduced in [2] and further developed in numerous works. Notable development is monadic parser combinators [5]. In this approach, one can not only parse the input, but simultaneously run semantics calculation if

¹Proposal with path pattern syntax for openCypher: <https://github.com/thobe/openCypher/blob/rpq/cip/1.accepted/CIP2017-02-06-Path-Patterns.adoc>. It is shown that context-free constraints can be expressed with the proposed syntax. Access date: 30.03.2020

parsing succeeds. Paper [6] proposed the first monadic parser combinator library which solves the long-standing problem of inability to handle ambiguous and left-recursive grammars. The authors earlier presented a library for graph querying was developed [15] based on this work. The core idea is to use generalized parser combinators as both a way to formulate a query and to execute it. This approach inherits benefits of combinatory parsing: ease of code reuse, type safety guaranteed by the host language and, since the parser is simply a function, the integrated development support.

Besides integration, it is also capable to compute both the single-source and all pairs semantics, as well as execute user actions. The single-source semantics is relevant to many real-world application, including manual data analysis. It also may be less time-intensive, since on average it needs to expose only a subgraph of the input graph. Many querying algorithms are only capable to compute all pairs reachability which makes them unsuitable for some applications.

In this paper we make the following contributions.

- We demonstrate how to use combinatory-based graph querying on example.
- We illustrate such features of the approach as type-safety, flexibility (composability and generics), IDE support and computing user-defined actions.
- We evaluate single-source context-free path querying on some real-world RDFs.
 - Based on our evaluation, the most common case in RDF context-free querying is when the number of paths in the answer set is big, but they are small.
 - We demonstrate that the single-source CFPQ can feasibly be used to evaluate such queries.
 - We conclude that there is a need to further detailed analysis of both theoretical time and space complexity of single-source CFPQ.

2 EXAMPLE OF CFPQ WITH COMBINATORS

In this section we demonstrate the 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 work we use Meerkat.Graph combinators library.

Problem statement. Suppose we have an RDF graph and want to analyze hierarchical dependencies over different types of relations. Our goal is for the given object to find all objects which lies on the same level of the hierarchy. Namely, for the given set of relations $r = \{R_0 \dots R_i\}$ and for the given vertex v we want to find all vertices reachable from v by paths which specified by the following context-free grammar in EBNF: $qSameGen \rightarrow R_0^{-1} qSameGen? R_0 | \dots | R_i^{-1} qSameGen? R_i$. Additionally, we want to calculate the length of these paths.

The first step is to specify paths constraint. For example, we fix relation to be `skos__narrowerTransitive`. Then constraint may be specified in terms of combinators as follows:

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

Here we use `inE` and `outE` to specify incoming and outgoing edges with the respective labels, `~` to concatenate subqueries, and `.?` to specify zero or one repetition of the subquery.

This query specifies exactly the 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 is for one specified relation. To investigate hierarchy over a set of relations we need to rewrite it.

Compositionality. The first step is a generalization of the query to simplify the handling of different types of relations. To do it we introduce a helper function `reduceChoice` which takes a list of subqueries 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 the new version of `sameGen` to combine subqueries for different types of braces. To make it possible to use different types of braces without query rewriting we pass braces as a parameter.

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

Now we are ready to provide the 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 that, 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 reachable from the specified start 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 subqueries 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")})
```

User-defined actions. The final step is to extend the query with the calculation of lengths of all paths which satisfied conditions. To do it we equip `sameGen` 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
        })
    })
```

The `queryFromV` now handles not only the third element but also the second one in order to get access to accumulated lengths.

```
def queryFromV(startV, query) =
  syn(startV ~ query ~ uriV &
    {case _ ~ (len:Int) ~ (v:Entity) =>
```

```
(len, v.getProperty[String]("uri"))))
```

Now we are ready to bring all functions together and evaluate the query. To do it first we add a helper function `makeBrs` which takes a list of relation names and create a list of pairs of subqueries which check incoming and outgoing respectively (pairs of brackets).

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

We use this function in the main function `runExample` which takes a list of relations, start vertex and the graph, build the same generation query over given relations by using specified functions and execute this query from the given vertex for the given graph.

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

Finally, to execute the query that we want from the vertex 1 we should call `runExample` as presented below.

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

Type safety. As far as queries are expressed in terms of functions of general-purpose language which you use for target application development, the compiler provides static type checking of queries and its results.

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

```
val q = queryFromV(syn(V(getIdFromNode(_: Entity) == 1) ^^),
  sameGen(symbolBrs))

val result = executeQuery(q, graph).toList

print(result.map(_._2).sum(.....))
```

No implicits found for parameter num: Numeric[String]

Figure 1: Error notification in a query in IDEA IDE

IDE Support. Since you can use IDE for development, you get all features for query development, such as syntax highlighting, code navigation, autocompletion, without any additional effort. An example of autocompletion suggestions for a vertex is presented in figure 2.

3 EVALUATION

We evaluate Meerkat.Graph on single-source context-free path querying scenario. For evaluation, we use a PC with Ubuntu 18.04 installed. It has Intel core i7-6700 CPU, 3.4GHz, DDR4 32Gb RAM. The Neo4j database is embedded into the application.

The dataset contains two real-world RDFs: Geospecies which contains information about the biological hierarchy² and Enzyme

²Geospecies RDF: <https://old.datahub.io/dataset/geospecies>. Access date: 12.03.2020.

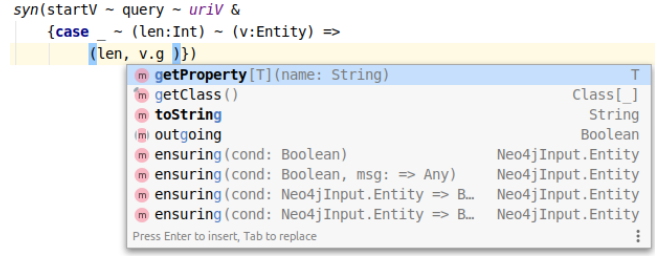


Figure 2: Query auto-completion in IDEA IDE

which is a part of the UniProt database³. A detailed description of these graphs is presented in table 1. Note, that graphs were loaded into database fully by using Neosemantix tool⁴, not only edges labeled by relations used in queries.

Graph	#V	#E	#SCO	#T	#NT	#BT
Enzyme	15088	47953	8202	15081	6819	8195
Geospecies	225134	1631525	0	89062	20830	20867

Table 1: Details of graphs

Queries for evaluation are versions of same-generation query — classical context-free query which is useful for hierarchy analysis. All queries in our evaluation created by using functions described in section 2. Namely, we create and evaluate three queries Q_1 , Q_2 and Q_3 as presented below.

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

def q2 (startV) =
  val q =
    sameGen(makeBrs(SKOS__BROADER_TRANSITIVE :: Nil))
  queryFromV(startV, q)

def q3 (startV) =
  val q =
    sameGen(makeBrs(SKOS__NARROWER_TRANSITIVE :: Nil))
  queryFromV(startV, q)
```

As you can see, once you create a set of appropriate functions, one can easily construct new queries.

For each graph and each query, we run this query from each vertex from graph and measure elapsed time and required memory by using ScalaMeter library⁵.

Results of evaluation are presented in figures 6 and 7 for query Q_1 , in figures 4 and 5 for query Q_2 , and in figures 8 and 10 for query Q_3 . Note that some results on time and memory measurements are presented in Appendix A. For each query result size we calculate an average value of required time and memory, and for each set

³Protein 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: 12.03.2020.

⁴Neosemantix is a Neo4j plugin for RDF to Neo4j import. Project page: <https://neo4j.com/labs/nsmtx-rdf/>. Access date: 30.03.2020.

⁵ScalaMeter library Web page: <https://scalometer.github.io/>. Access date: 12.03.2020.

of points we also draw linear approximation of this set. While memory consumption is quite stable, time measurements contain outliers but they are not sufficient. To demonstrate it we provide standard boxplot for Q_3 in figure 9. Also we collect paths length distribution which is showed in figure 3.

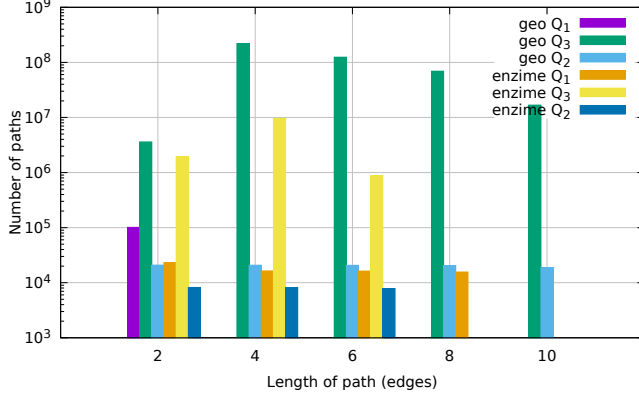


Figure 3: Paths length distribution

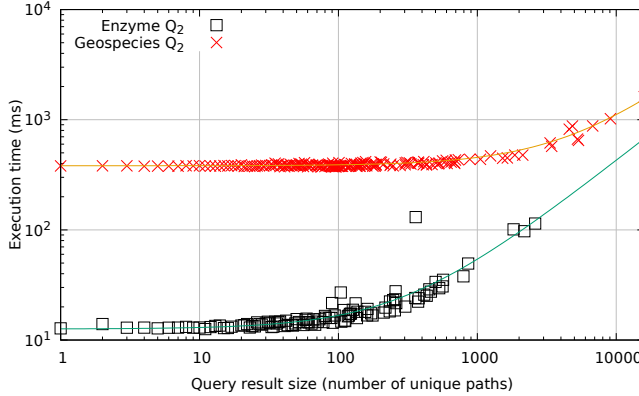


Figure 4: Execution time for Q_2 query

First of all, we can see that provided datasets contain relatively short paths which satisfy queries: the longest path for all queries contains 10 edges.

Figures 4, 8 and 6 show dependency of query evaluation time on query result size measured in terms of number of unique paths. First, we can see that query evaluation time is linear on result size. Also we can see, that time which required to evaluate query for one specific vertex is relatively small. For example, for Q_2 and Enzyme RDF 15051 queries (99.75%) were executed in less than 20ms, and only 3 queries require more than 100ms.

Figures 5, 10, and 7 show dependency of memory required to evaluate query on query result size. We can see, that memory consumption is linear on result size, and relatively small (not exceed 512 Mb even for big results).

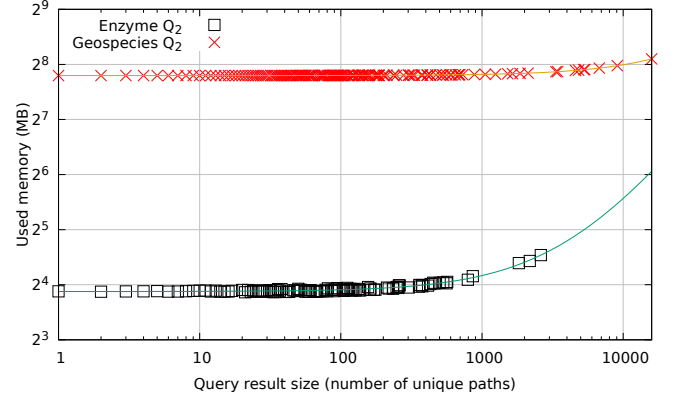


Figure 5: Memory consumption for Q_2 query

We can see, that both, time and memory consumption depend on input graph size, and its dependency looks like constant overhead which is independent from query and query result size. It is because of Meerkat.Graph is implemented on top of Neo4j and cannot use internal data structures and create an optimal query execution plan. For example, to find start vertex by predicate specified in the query it performs linear scan over all vertices for each query. It is the reason for the time difference between Enzyme and Geospecies datasets. To prevent such problems, query execution mechanisms should be deeply integrated into the database engine.

Finally, we can conclude that context-free path querying in single-source scenario can be efficiently evaluated in case when number of paths in answer is big but its length is relatively small, while all pairs scenario is still hard [7]. Also we can see that while theoretical time and space complexity of CFPQ algorithms at least 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.

4 CONCLUSION AND FUTURE WORK

We show that single-source context-free path queries can be evaluated in reasonable time for real-world graphs. Also, we demonstrate that the combinator-based approach to CFPQ is very flexible and powerful.

We demonstrate a combinator-based approach implemented in Meerkat.Graph Scala library, but this approach can be implemented in almost any high-level programming language. While combinators are 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 (GLL-based [3, 8] or GLR-based [11, 14]) 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 more 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. Also, we should provide detailed theoretical analysis of single-source CFPQ.

One of the important directions of future research is to optimize the performance of the proposed solution. We hope that it is possible to reduce the utilization of graph size-dependent structures and thus make query execution time depends only on the size of the result. One of the possible solutions is deep integration with Neo4j infrastructure to utilize cache system.

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

ACKNOWLEDGMENTS

The research was supported by the Russian Science Foundation grant 18-11-00100.

REFERENCES

- [1] Rustam Azimov and Semyon Grigorev. 2018. Context-free Path Querying by Matrix Multiplication. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (Houston, Texas) (GRADES-NDA '18). ACM, New York, NY, USA, Article 5, 10 pages. <https://doi.org/10.1145/3210259.3210264>
- [2] William Burge. [n.d.]. Recursive Programming Techniques.
- [3] Semyon Grigorev and Anastasiya Ragozina. 2017. Context-free Path Querying with Structural Representation of Result. In *Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia* (St. Petersburg, Russia) (CEE-SECR '17). ACM, New York, NY, USA, Article 10, 7 pages. <https://doi.org/10.1145/3166094.3166104>
- [4] Jelle Hellings. 2014. Conjunctive context-free path queries. In *Proceedings of ICDT'14*. 119–130.
- [5] Graham Hutton and Erik Meijer. 1996. Monadic parser combinators. (1996).
- [6] Anastasia Izmaylova, Ali Afrozeh, and Tijs van der Storm. 2016. Practical, general parser combinators. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. 1–12.
- [7] Jochem Kuijpers, George Fletcher, Nikolay Yakovets, and Tobias Lindaaker. 2019. An Experimental Study of Context-Free Path Query Evaluation Methods. In *Proceedings of the 31st International Conference on Scientific and Statistical Database Management* (Santa Cruz, CA, USA) (SSDBM '19). Association for Computing Machinery, New York, NY, USA, 121–132. <https://doi.org/10.1145/3335783.3335791>
- [8] Ciro M. Medeiros, Martin A. Musicante, and Umberto S. Costa. 2019. LL-based query answering over RDF databases. *Journal of Computer Languages* 51 (2019), 75–87. <https://doi.org/10.1016/j.cola.2019.02.002>
- [9] Thomas Reps. 1997. Program Analysis via Graph Reachability. In *Proceedings of the 1997 International Symposium on Logic Programming* (Port Washington, New York, USA) (ILPS '97). MIT Press, Cambridge, MA, USA, 5–19. <http://dl.acm.org/citation.cfm?id=271338.271343>
- [10] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. 1994. Speeding up Slicing. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering* (New Orleans, Louisiana, USA) (SIGSOFT '94). Association for Computing Machinery, New York, NY, USA, 11–20. <https://doi.org/10.1145/193173.195287>
- [11] Fred C. Santos, Umberto S. Costa, and Martin A. Musicante. 2018. A Bottom-Up Algorithm for Answering Context-Free Path Queries in Graph Databases. In *Web Engineering*, Tommi Mikkonen, Ralf Klamma, and Juan Hernández (Eds.). Springer International Publishing, Cham, 225–233.
- [12] Petteri Sevon and Lauri Eronen. 2008. Subgraph Queries by Context-free Grammars. *Journal of Integrative Bioinformatics* 5 (06 2008). <https://doi.org/10.1515/jib-2008-100>
- [13] 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>
- [14] Ekaterina Verbitskaia, Semyon Grigorev, and Dmitry Avdyukhin. 2016. Relaxed Parsing of Regular Approximations of String-Embedded Languages. In *Perspectives of System Informatics*, Manuel Mazzara and Andrei Voronkov (Eds.). Springer International Publishing, Cham, 291–302.
- [15] 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>
- [16] Charles B Ward, Nathan M Wiegand, and Phillip G Bradford. 2008. A distributed context-free language constrained shortest path algorithm. In *2008 37th International Conference on Parallel Processing*. IEEE, 373–380.
- [17] Xiaowang Zhang, Zhiyong Feng, Xin Wang, Guozheng Rao, and Wenrui Wu. 2016. Context-Free Path Queries on RDF Graphs. In *The Semantic Web – ISWC 2016*, Paul Groth, Elena Simperl, Alasdair Gray, Marta Sabou, Markus Krötzsch, Freddy Lecue, Fabian Flöck, and Yolanda Gil (Eds.). Springer International Publishing, Cham, 632–648.
- [18] Xin Zheng and Radu Rugina. 2008. Demand-driven Alias Analysis for C. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) (POPL '08). ACM, New York, NY, USA, 197–208. <https://doi.org/10.1145/1328438.1328464>

A ADDITIONAL EVALUATION RESULTS

Here we provide detailed results of time and memory consumption measurement for all queries.

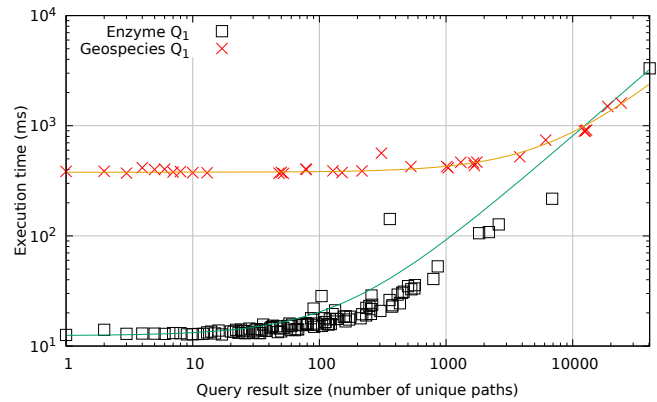


Figure 6: Execution time for Q_1 query

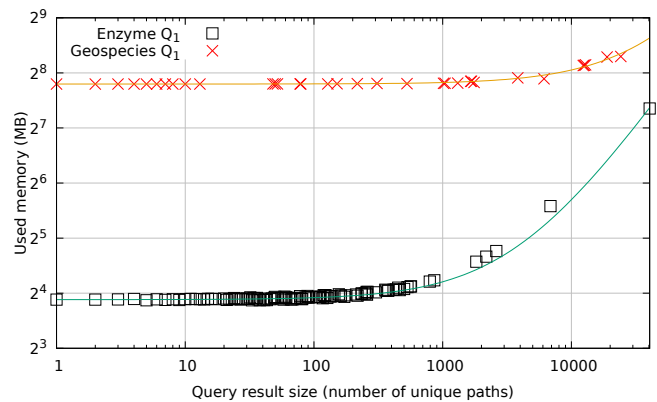
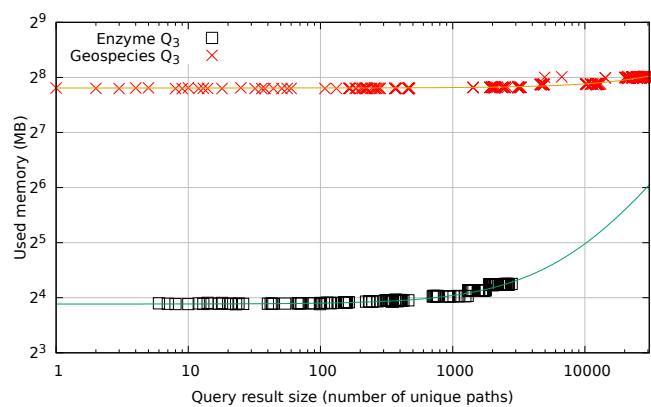
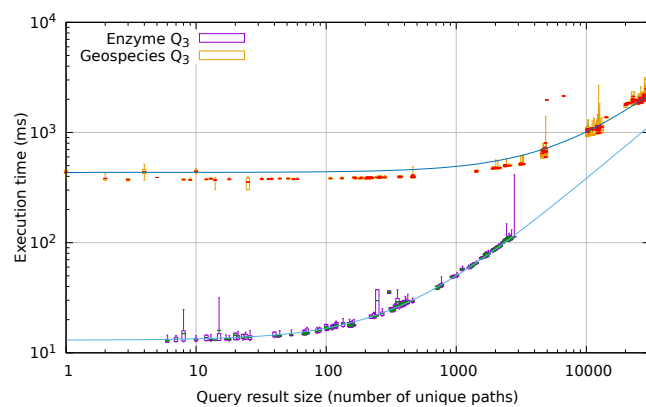
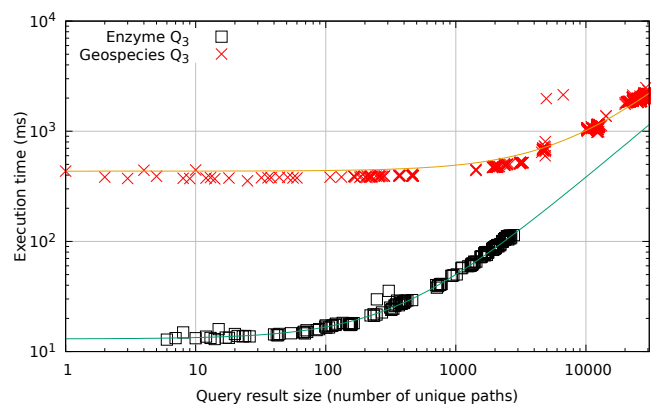


Figure 7: Memory consumption for Q_1 query

Time and memory measurements results for Q_1 query are provided in figures 6 and 7 respectively.

Time measurement results for Q_3 query are provided in two ways: only average values for each query result size (figure 8) and standard

Figure 10: Memory consumption for Q_3 queryFigure 9: Execution time for Q_3 query (boxplot)Figure 8: Execution time for Q_3 query

boxplot to provide information about values distribution (figure 9). We can see that number of outliers is small. Note that the number of measurements for each query result size is different, so in some cases we have just single points instead of boxes (it means that there is only one result of such size).

Memory consumption for Q_3 query is presented in figure 10.

Also we can see, that for provided queries and graphs time and memory consumption are not dependent on query: for similar result sizes required time and memory are similar for all queries.