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
kajigor@gmail.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 as well as 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 explain how to use parser-combinators for context-free path querying. Then we demonstrate by presenting a step-by-step example how this approach can solve a problem of transparent integration with general-purpose language and provide better type safety, composability, user-defined actions handling, and development environment support. We also evaluate a combinators-based query execution procedure on two real-world RDFs in the single source case, and show that although combinators are suitable for real-world single source CFPQ specification and processing, a 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 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, 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 also used for static code analysis [10, 11, 19], RDF querying [9, 18], biological data analysis [13].

Most of research is focused on developping algorithms for CFPQ evaluation [1, 3, 4, 9, 13, 17], whereas specification languages for context-free queries are not investigated enough. Best to our knowledge, only one extension for Sparql supports context-free constarints: cfSPARQL [18]. There is also a proposal for CFPQ as a part of Cypher[1] language, but there is no implementation for it yet. We believe that more research should be conducted on the specification languages fo 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, and even more sophisticated.

Although simple, the string manipulating approach does not provide a developper 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 powerfull 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 may 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 maintainance [14].

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 developped 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

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

[1] Proposal 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. A library for graph querying was developped [16] 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 can 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 applications, including manual data analysis. It also may be less time-intensive, since on average it needs to expore 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 for a 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. We first introduce a simple graph analysis problem and then show how to solve it by using parser combinators. In our work we use the combinators library Meerkat.Graph [2].

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 lie on the same level of the hierarchy. Namely, for the given set of relations $r = \{R_0 \ldots R_i\}$ 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: $qSameGen \to {R_0}^{-1} qSameGen?\, R_0 \mid \ldots \mid {R_i}^{-1} qSameGen?\, R_i$. Additionally, we want to calculate the length of these paths.

The first step is to specify a paths constraint. For example, we consider 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 the incoming and outgoing edges with the respective labels, ~ to concatenate subqueries, and `.?` to specify an optional subquery.

This query specifies exactly the path wanted, but is still not a solution. First of all, we cannot specify start vertex and cannot extract final vertices. Also, this query is for a single relation. To investigate hierarchy over a set of relations we need to rewrite it.

Compositionality. The first step is to generalize the query to simplify the handling of different types of relations. We introduce a helper function `reduceChoice` which takes a list of subqueries and combines them using the 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 "brackets". The brackets are passed as parameters so the query can be instantiated for different brackets.

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

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

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

We create a function `queryFromV` which takes the start vertex `startV` and a path `query` as an input, and creates a query to find all vertices with `uri` property which are reachable from the `startV` by a path from the `query` result. Finally, we collect values of `uri` for all reachable vertices by specifying a user-defined action `{case _ ~ _ ~ (v: Entity) => v.getProperty[String]("uri")}` which captures the result of query (it is a triple-sequence of subqueries results) and gets the `uri` property from the result of the 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 to calculate lengths of all paths which satisfy conditions. To do this, 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 also handles the second element of the triple in order to get access to the accumulated lengths.

[2] Meerkat.Graph repository: https://github.com/yaccconstructor/meerkat. Access date: 12.03.2020

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

Now we are ready to combine the functions and evaluate the query. First, we add a helper function makeBrs which takes a list of relation names and creates a list of pairs of subqueries which check incoming and outgoing edges respectively (pairs of brackets).

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

The main function runExample takes a list of relations, the start vertex and the graph, builds the same generation query over the given relations by using the functions described and executes it.

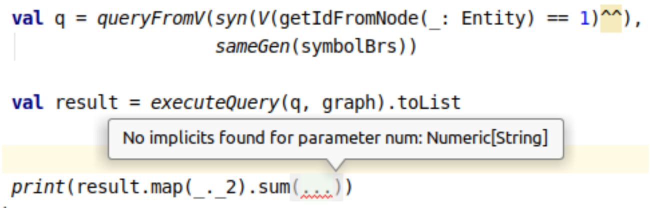
```
def runExample (brs: List[_], startVId, graph) =
    val startV = V(getIdFromNode(_: Entity) == startVId
    executeQuery(queryFromV( syn(startV)^^),
                           sameGen(makeBrs(brs))),
                 graph).toList
```

To execute the query for the vertex 1, one should call runExample as presented below.

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

Type safety. As far as queries are functions of a general-purpose language, which is used to develop the application, its compiler type checks the queries and their results statically.

Figure 1 demonstrates the example of type checking. There is an error in the way the total length of the paths is computed: the indentifiers of the final vertices are summed instead of the lengths. The compiler statically detects a type error because an integer is expected but a string is provided.



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

val result = executeQuery(q, graph).toList
                No implicits found for parameter num: Numeric[String]

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

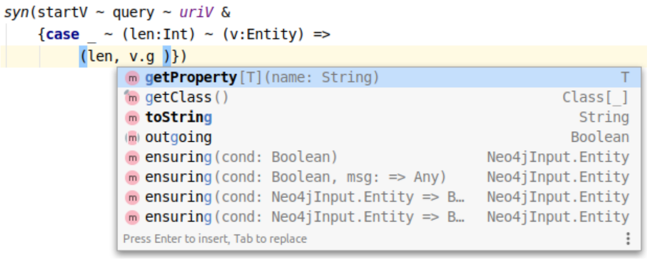
Figure 1: Error notification in a query in IntelliJ IDEA

IDE Support. By using an IDE to write queries, one can benefit from such features as syntax highlighting, code navigation, autocompletion without any additional effort. Figure 2 shows an example of autocompletion suggestion for a vertex.

3 EVALUATION

We evaluate Meerkat.Graph in the single-source context-free path querying scenario. We use a PC with Ubuntu 18.04 installed for evaluation. It has Intel core i7-6700 CPU, 3.4GHz, and 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 hierarchy[3] and *Enzyme* which is a part of the UniProt database[4]. A detailed description of these graphs is presented in the table 1. Note, that the full graphs were loaded into the database by using Neosemantix tool[5], not only the edges labeled by the relations used in queries.



```
syn(startV ~ query ~ uriV &
   {case _ ~ (len:Int) ~ (v:Entity) =>
       (len, v.g )})
           getProperty[T](name: String)                          T
           getClass()                                     Class[_]
           toString                                         String
           outgoing                                        Boolean
           ensuring(cond: Boolean)                 Neo4jInput.Entity
           ensuring(cond: Boolean, msg: => Any)    Neo4jInput.Entity
           ensuring(cond: Neo4jInput.Entity => B…  Neo4jInput.Entity
           ensuring(cond: Neo4jInput.Entity => B…  Neo4jInput.Entity
           Press Enter to insert, Tab to replace
```

Figure 2: Query auto-completion in IntelliJ IDEA

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

The queries for the evaluation are versions of the same-generation query — a classical context-free query useful for hierarchy analysis. All queries in our evaluation are created using the 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_TRANSITIVEY :: Nil))
    queryFromV(startV, q)
```

It is plain to see that once the set of appropriate functions is created, new queries can be easily constructed.

We run every query from each vertex of each graph and measure elapsed time and required memory by using ScalaMeter library[6].

The results of the 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 are presented in Appendix A. For each query result size we average the time and memory required, and for each set of points we also draw the linear approximation of this set. While memory consumption is quite stable, time measurements contain some outliers which are not significant. We provide a standart boxplot for Q_3 in figure 9.

We also collect the distribution of paths length which shown in figure 3. Paths distribution shows that most paths in the queries results are short: the longest consists of 10 edges.   Figures 4, 8

[3] *Geospecies* RDF: https://old.datahub.io/dataset/geospecies. Access date: 12.03.2020.

[4] Protein sequences data base: https://www.uniprot.org/. RDFs are available here: ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/rdf. Access date: 12.03.2020.

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

[6] ScalaMeter library Web page: https://scalameter.github.io/. Access date: 12.03.2020.

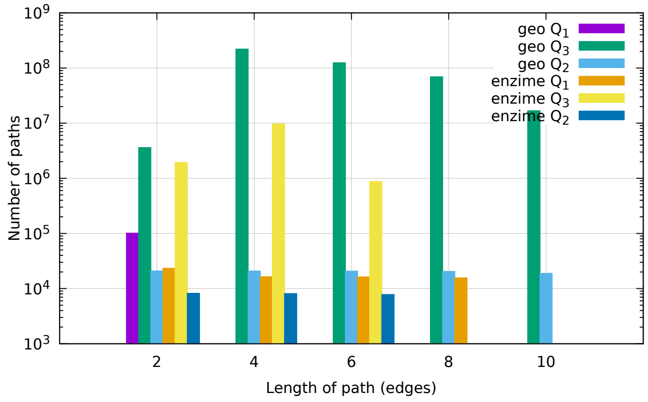


Figure 3: Paths length distribution

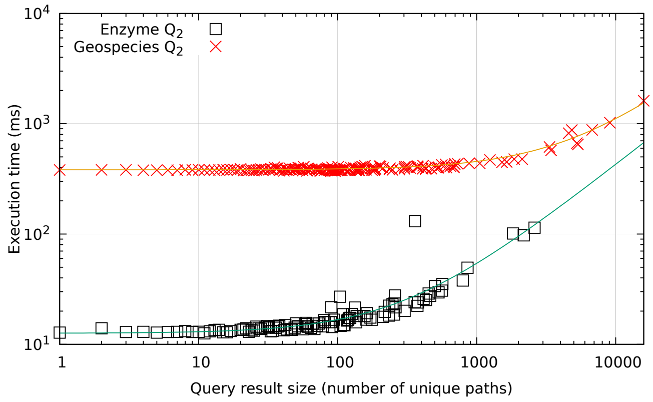


Figure 4: Execution time for Q_2 query

and 6 show the dependency of the evaluation time on the query result size (the number of unique paths). We can see that query evaluation time is linear on result size. The time required to evaluate a query for one specific vertex is small: for Q_2 and *Enzyme* RDF 15051 queries (99.75%) were executed in less then 20ms, and only 3 queries required more than 100ms. Figures 5, 10, and 7 show dependency of memory required for the evaluation on the query result size. We can see, that memory consumption is linear on result size, and is relatively small (does not exceed 512 Mb even for the larger results).

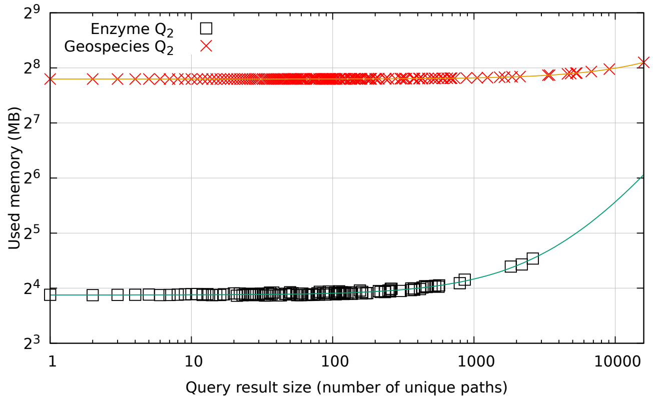


Figure 5: Memory consumption for Q_2 query

We can see, that both the time and memory consumption depend on the input graph size, and there is a constant overhead independent from the query and the query result size. We believe the reason to be that Meerket.Graph is implemented on top of Neo4j and thus cannot use internal data structures and create an optimal query execution plan. For example, it performs a linear scan over all vertices for each query to find the start vertex. It is the reason for the time difference between runs for the *Enzyme* and *Geospecies* datasets. To prevent such issues, query execution mechanisms should be deeply integrated into the database engine.

Finally, we can conclude that confext-free path querying in the single-source scenario can be efficiently evaluated in case when the number of paths in the answer is big but their length is relatively small, while all pairs scenario is still complicated [8]. We can also see that while the theoretical time and space complexity of CFPQ algorithms is at least cubic, in the scenario demonstrated the real execution time and used memory are linear. Thus it is necessary to provide detailed time and space complexity analysis of the algorithms.

4 CONCLUSION AND FUTURE WORK

We show that single-source context-free path queries can feasibly be evaluated for real-world graphs. We demonstrate that the combinators-based approach to CFPQ is 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 powerful way to specify context-free queries, the technique seem hard for many users, but it may be a native way for functional query languages, like RQL [7]. On the other hand, other algorithms for CFPQs should be applicable for single-source path querying (GLL-based [3, 9] or GLR-based [12, 15]) and we believe that they can be integrated with the existing graph databases in a more convenient way.

We should investigate more datasets to detect other shapes of query results. For example, when a number of resulting paths in the single-source querying is small, but the paths are relatively long. The first question here is which data analysis tasks lead to such scenario. Also, we should provide detailed theoretical analysis of single-source CFPQ.

One important direction for future research is to optimize the performance of the proposed solution. We believe that it is possible to reduce the use of graph size-dependent structures and thus make query execution time depend only on the size of the result. One possible solution is a deep integration with the Neo4j infrastructure to utilize its cache system.

Another direction is combinators library improvement. It is necessary to make combinators syntax more user-friendly and to create a set of query templates (such as 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 Afroozeh, 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] Gregory Karvounarakis, Aimilia Magkanaraki, Sophia Alexaki, Vassilis Christophides, Dimitris Plexousakis, Michel Scholl, and Karsten Tolle. 2004. *RQL: A Functional Query Language for RDF*. Springer Berlin Heidelberg, Berlin, Heidelberg, 435–465. https://doi.org/10.1007/978-3-662-05372-0_18

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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.

[13] 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

[14] 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

[15] 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.

[16] 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

[17] 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.

[18] 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.

[19] 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

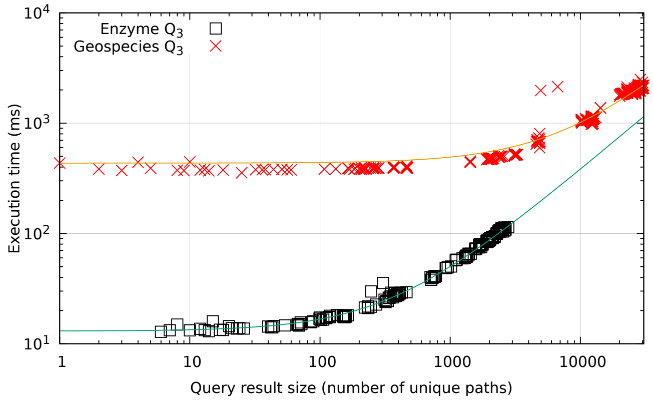


Figure 8: Execution time for Q_3 query

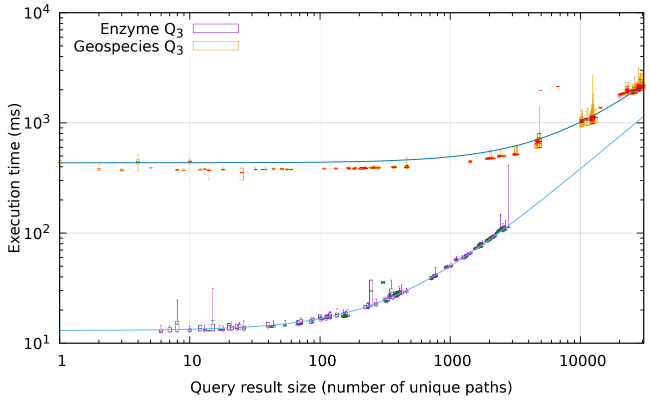


Figure 9: Execution time for Q_3 query (boxplot)

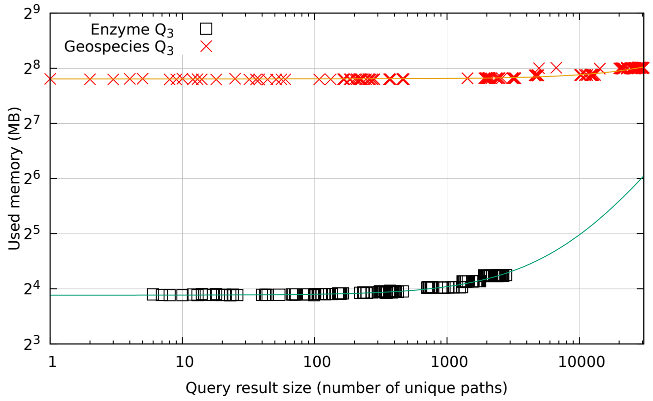


Figure 10: Memory consumption for Q_3 query

A ADDITIONAL EVALUATION RESULTS

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

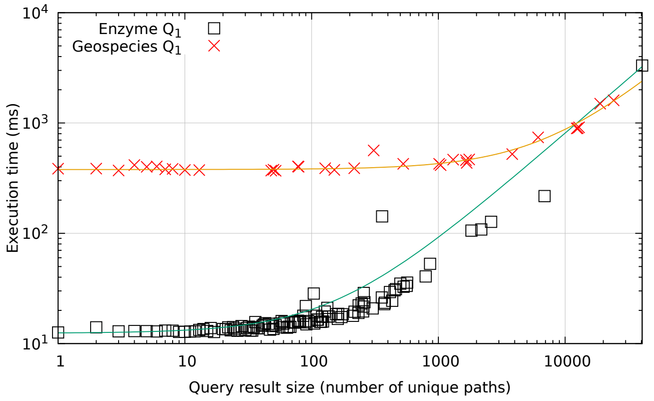


Figure 6: Execution time for Q_1 query

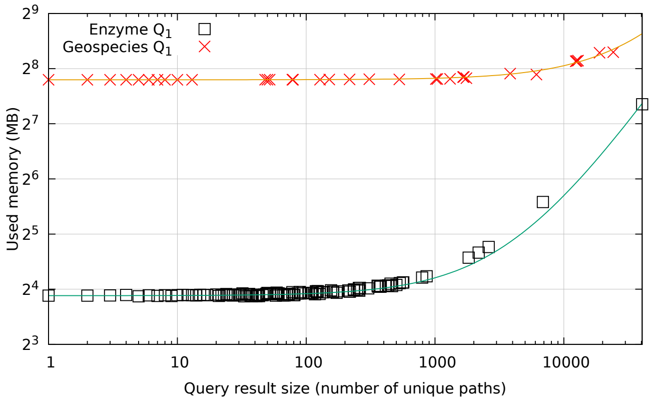


Figure 7: Memory consumption for Q_1 query

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

Time measuremet results for Q_3 query are provided in two ways: only average values for each query result size (figure 8) and standard boxplot to provide information about vales 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 jus 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 depend on query: for similar result sizes reqired time and memory are similar for all qieryes.