Benchmarking graph databased with gMark

Advokaat, N.

Award date:
2016

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Web Engineering Research Group

Benchmarking Graph Databases with gMark

Nicky Advokaat

Supervisors:
prof. dr. A. Bonifati Lyon 1 University
dr. G.H.L. Fletcher

Assessment committee:
prof. dr. A. Bonifati Lyon 1 University
dr. G.H.L. Fletcher
dr. N. Sidorova
dr. ir. J. Vanschoren

Eindhoven, December 2015

Abstract

Graphs are used to model data in many different areas such as social networks and protein interaction networks. The traditional relational database model is conceptually not very well suited for hosting large graphs and path queries can not naturally expressed, causing specialized graph databases to become more prevalent. Developers of database systems often perform benchmarks to evaluate the performance of their systems. For this purpose large amounts of graph data is required, however there is a limited amount of real life data. Therefore synthetic data is generated that mimics properties found in the real life graph.

In this work we will measure, analyze, and compare the performance of four database systems on synthetic graph data and queries. A detailed study is performed on recursive queries. As a side topic we will attempt to generate synthetic graph data based on a new model that we constructed ourselves, which is based on a protein information network. An analysis is performed on how well this synthetic data simulates the real life dataset.

Our results show graph databases generally do not perform better than a relational database on non-recursive regular path queries. We will demonstrate that we can capture the essential properties of a real life dataset and leverage this to generate synthetic data resembling the original graph.

Keywords: graph database, benchmark, regular path queries, protein network

Preface

This thesis was performed within the Web Engineering group of the Eindhoven University of Technology, in collaboration with INRIA Nord Europe, University Lyon 1, and the University of Oxford.

I would like to express my gratitude to my daily supervisor George Fletcher for guiding me through the project, always being enthusiastic and positive. Also, I want to thank Angela Bonifati for her support, guidance, and feedback throughout the project. Furthermore I'd like to thank the other members of the gMark team: Guillaume Bagan, Radu Ciucanu, and Aurélien Lemay, working together with an actual research group was a great experience and has taught me a lot. Finally I want to thank Natalia Sidorova and Joaquin Vanschoren for being part of the assessment committee and the feedback they gave me.

Nicky Advokaat
December 2015

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Motivation Any kind of data can be modeled as a graph, a data structure consisting of objects called vertices, and their relationships to other vertices called edges. Recent years saw a significant increase of large graphs such as social networks, the semantic web, and citation networks, and thereby caused a growing (scientific) interest in this kind of data. These developments caused a rise of database systems that have graph structure at the core of their design principle, unlike the more traditional SQL based relational database management systems which rely on tuples grouped in relationships. These graph databases have various design philosophies and implementations; data storage can be based on common graph structures such as RDF or as a custom graph implementation, and queries can in various languages such as SPARQL, Datalog or Regular Path Queries. Developers of these systems all make various claims about their performance, so we wanted to perform an independent benchmarking study comparing query execution performance on graph data between various systems, which will be the main contribution of this work.

To this end we require a large amount of graph data. However benchmark data is not always available in sufficient quantities, it often comes in different graph dimensions and has different properties than desired, and usually does not include a relevant set of queries. This gave us the idea to perform our benchmark using software generated synthetic graphs, enabling us to generate as much data as many graphs as we want, in any size, and having the properties we require.

Problem statement Given a large amount of synthetic graph data sets mimicking different domains and having various sizes, accompanying sets of regular path queries for each domain, and a number of graph database systems, we want to determine which system has the best overall performance measured in query execution time. The graph database systems we consider are DatalogSystem, GraphSystem, and SPARQLSystem(*). Besides these three systems, we will also use the relational DBMS PostgreSQL [1] in our experiments. Thereby we can also answer the question: do graph databases actually perform better than relational databases on regular path queries over graph data? Specific emphasis will be placed on the class of recursive queries because they have a distinct behavior and are generally more difficult to solve.

The synthetic data is created by the benchmark generation tool gMark [2], we decided to use this tool rather than other benchmark suites [3, 4, 5, 6, 7] because gMark can model any domain using a custom schema, generates not only graphs but also queries, and does not require an instance of the graph to generate queries. In order to generate graphs and queries gMark requires as input a schema definition modeling some domain, for example a bibliographical citation network. The schemas used to generate data for our benchmarks are pre supplied by gMark. This inspired us to investigate whether we can model and generate data simulating a new domain: the protein sequence and annotation database UniProt [8]. To answer this question we basically have to perform two steps; first we must extract properties from the original graph data such that we can create the schema, then we must analyze the generated graph to see how well it resembles the original graph.

Contributions Our contributions are thus a detailed benchmark study on three graph database systems using synthetic graph data and queries, and at the same time a comparison between graph and relational databases, where we make a distinction between recursive and non-recursive queries. Secondly we will demonstrate the generation of synthetic data based on a protein network, a domain for which, to our knowledge this, has not yet been attempted.

Thesis outline In Chapter 2 concepts and definitions used in this work are defined and explained. The main contribution of this work is given in Chapter 3, the benchmark study. Chapter 4 provides a more detailed study on recursive queries. In Chapter 5 we will model and generate synthetic graph data based on a protein graph network. Chapter 6 concludes the paper and summarizes the results. Appendix A demonstrates how a query can be translated into the query language used by each of the database systems used in this work. Finally, Appendix B contains the source of the schema used in Chapter 5.

*__Disclaimer__ Three of the database systems subjected to the benchmark in this work have been anonymized in order to protect the interests of their respective developers. The names used are fictitious and references that could reveal their identity are hidden. All three of these graph databases are leading examples of the major current approaches to graph query processing.

Chapter 2

Preliminaries

This chapter provides background information on the material presented in the following chapters. First we will present our notion of graphs and define our query language. An overview of the systems that will be subjected to the benchmark is given in Section 2.2, including details about their graph implementation and query language. In Sections 2.3 and 2.4 we will discuss the benchmark generation tool gMark and how it is used in our experiments. Finally in Section 2.5 we will discuss relevant literature and position this work amongst them.

2.1 Domain and query language

In this work we will be dealing exclusively with directed edge labeled graphs. Formally such a graph G is defined as $G = (V, E, \Sigma, F)$, where V is a set of vertices and $E \subseteq V \times V$ is a set of edges between vertices in V. The set of labels is denoted by Σ, and the injective function $F : E \to \Sigma$ maps edges to labels. An example graph G is shown in Figure 2.1 where $V = \{1, 2, 3, 4\}$, $E = \{(1, 2), (1, 3), (2, 3), (3, 4)\}$, $\Sigma = \{a, b, c\}$, and $E = \{(1, 2) \to a, (1, 3) \to a, (2, 3) \to b, (3, 4) \to c\}$. The queries we will execute on these graphs are unions of conjunctions of regular path queries (UCRPQ). Each conjuncts consist of two variables connected by a path expression. A path expression is a regular expression over predicates and operators. Predicates can be either labels from $\Sigma = \{a, b, \ldots\}$ or negations of these $\{a^-, b^-, \ldots\}$ which denotes a reverse edge. There are three operators in our language: concatenation $(\cdot)$, disjunction $(+)$, and recursion $(*)$. We allow

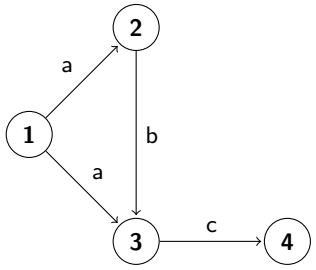

Figure 2.1: Example of a directed edge labeled graph.

recursion to occur only at the outermost level. The syntax of a query rule is defined as:

```
query = (head) <- body
 head = var,head | var
 body = conjunct,body | conjunct
conjunct = (var,path,var)
 path = (non_rec_path)* | non_rec_path
non_rec_path = ((non_rec_path) operator (non_rec_path)) | predicate
predicate = label | label⁻
label = a | b | c| ...
 var = ?x | ?y | ?z | ...
operator = · | +
```

We also pose the constraint that all variables appearing in the head must be present in the body as well. An example query rule:

$$(?x, ?z) \leftarrow (?x, (a^-)*, ?y), (?y, (b \cdot c) + d, ?z)$$

A UCRPQ query is now defined as set of query rules where the heads have the same variables, hence the Union part of the name. An example query having two query rules is:

$$(?x, ?y) \leftarrow (?x, (a + b), ?y)$$
$$(?x, ?y) \leftarrow (?x, (c)*, ?y)$$

Note that every query rule is in itself a query consisting of a single rule.

2.2 Database systems

In this work we will be using four database systems. Three of those are (anonymous)graph databases:

- **DatalogSystem**. An implementation of Datalog. Queries are expressed in the Custom-Datalog language. We implement graphs by defining a 'block' called edge:

  ```
  edge(x,y,z) -> int(x), int(y), int(z).
  ```

  Where x, y, z are the source, label, and target id's, respectively.

- **GraphSystem**. Open source native graph database. Queries are expressed in the openCypher language [9]. GraphSystem is currently one of the highest ranked graph database systems according to popularity [10].

- **SPARQLSystem**. A hybrid database server incorporating multiple classes of database systems, of which we will be using only the RDF Triple Store. Queries are written in SPARQL 1.1.

Besides these graph databases we will also have one relational database:

- **PostgreSQL** [1]. Often called Postgres, this is a traditional relational database management system. The graph will be stored in a single table with columns Source, Label, and Target. This means that for each edge in our graph there is a single entry in this table. To maximize performance SQL Indexes are created on all three columns. Indexes are implemented as B-trees ensuring that searching can always be done in $O(\log |E|)$. Queries are expressed in SQL:2011, Postgres implements recursive SQL queries.

An overview of these four systems is given in table 2.1.

System	Version	Database Type	Query Language
DatalogSystem	4.2.1	Datalog	CustomDatalog
GraphSystem	2.2.2	Graph	openCypher
PostgreSQL	9.4	Relational	SQL
SPARQLSystem	7.2	RDF	SPARQL

Table 2.1: The database systems

2.3 gMark

All the graphs and corresponding queries used in this work are generated by gMark [2], an open-source domain- and query language-independent synthetic graph benchmarking tool. The advantage of using gMark compared to other systems [3, 4, 5, 6] is that we can both have a flexible graph schema that allows us to model any domain, and that it allows us to generate queries based on this schema instead of requiring an instance of the graph.

gMark generates data based on a schema modeling a certain domain, describing nodes, relations, and other properties desired in the graph and queries. A more detailed discussion of the schema definition is given in Section 2.3.1. Throughout the following chapters we will mainly be using schemas modeling either of these domains:

- **Biblio**. A bibliographical network including nodes such as *researcher, paper, journal* and relations between them such as *author* between researchers and papers.
- **LDBC**. [4] The Linked Data Benchmark Council Social Network Benchmark, a network including *persons, messages, organizations* and relations like *likes* and *workAt*.
- **WatDiv**. [3] The Waterloo SPARQL Diversity Test Suite(WatDiv) models an online store with nodes *products, retailers, stores*, etc.

An overview of our use-case of gMark is shown in Figure 2.2. Here we can see that gMark uses the schema to generate both the graph and the queries without requiring a graph instance. The gMark query translator is used to rewrite the UCRPQ queries to the syntax of each of the four systems. The following sections provide a detailed description of the data generation, query generation, and query translation in gMark.

2.3.1 Schema definition

The input for gMark based on which it generates data is an XML file called the schema definition. It consists of two parts, the first part sets constraints during graph generation by the following variables:

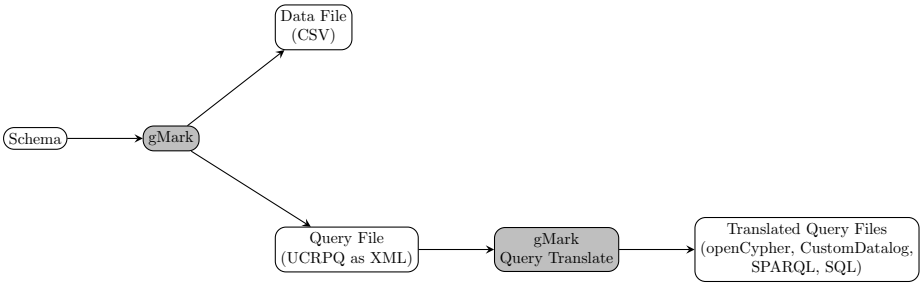

Figure 2.2: A high level overview of how gMark is used to generate graphs and queries.

- **Types.** The types of nodes that occur in the graph, such as *user* and *article*. We can indicate a proportion for each type, if we want roughly 50% of the nodes to by of type *user* we set the proportion to 0.5. It is also possible to set a fixed number, say when want exactly 30 types of articles.
- **Predicates.** The set of labels that can occur in the graph, similar to types we can set proportions or fixed amounts for the occurrence.
- **Distributions.** For each edge label we can set the probability distribution for the in and/or out degree. To do this we specify the source type, target type, and one of three distributions:
  - *Gaussian.* The normal distribution can be observed in many natural phenomena. It requires parameters for mean value μ and standard deviation σ can be specified.
  - *Uniform.* A uniform distribution between a *min* and *max* value.
  - *Zipfian.* Probability distribution used can be used to model frequency of a few popular and many less popular items. For example in the English language 'the' and 'it' occur frequently, but the vast majority of words is used infrequently. Another example is knowing people, there are some people that are known by almost everyone, but most people are not that famous. Decay in popularity is specified by α value.

  An example setting would be: "From source *researcher* to target *paper* we want the indistribution to be Gaussian ($\mu = 3, \sigma = 1$) and the out distribution Zipfian ($\alpha = 2.5$)."

A set of queries is defined in a workload describing properties the query should have and the number of queries to be generated. We can have multiple workloads in on schema, and set the following parameters:

- **Size.** Positive integer for the number of queries to be generated. When the other parameters allow it gMark tries to avoid duplicate queries.
- **Conjuncts.** Set the minimum and maximum number of conjuncts a query can have.
- **Disjuncts.** Min and max values for the number of disjuncts ($+$) in a query.
- **Length.** Min and max values for the length of paths connected by concatenation ($\cdot$).
- **Recursion.** A number between 0 and 1 determining the probability of the query containing a star ($*$). A query can contain at most one star, which must appear at the outermost level of a path.
- **Arity.** Number of variables in the head of the query. In our experiments we will almost exclusively be using binary queries, i.e. queries with arity two.
- **Selectivity.** Determines the expected number of results by a query on a graph. We consider three classes:
  - *Constant.* The number of results selected by this query does not significantly grow as the graph increases in size.
  - *Linear.* Number of results grows at roughly the same rate as the graph does.
  - *Quadratic.* Growth of results is quadratic to graph growth.

2.3.2 Data generation

Given a schema definition and a positive integer n indicating the target graph size $|V|$ gMark generates an edge list as a CSV file. Each line in this file corresponds to one edge of the graph formatted as (source, label, target).

Once we have generated the data there we might have to perform a simple formatting step to be able to insert it into each of the databases. DatalogSystem and Postgres accept the CSV format (source,label,target) directly, such as:

```
1343,5,201
```

GraphSystem poses the constraint that labels must be strings, so we simply append a character in front of the label id:

```
1343,"p5",201
```

SPARQLSystem requires valid RDF in N-Triples format so we create a dummy name-space:

```
<http://example.org/gmark/o1343> <http://example.org/gmark/p5> <http://example.org/gmark/o201> .
```

When we have the graph data correctly formatted, inserting it into the database is quite straightforward for each of the systems.

2.3.3 Query generation and translation

Besides generating graphs, gMark can also generate the corresponding queries given the schema definition. gMark is designed to be language independent and broadly applicable, so generated queries are stored as an XML formatted representation of UCRPQ. A query translator is provided by gMark which can translate from this file to all of the four languages as listed in table2.1.

For CustomDatalog, SPARQL, and SQL the queries can readily be translated, for openCypher there are some limitations:

- We can not express inverse relations in openCypher for example:

  $$(?x, ?y) \leftarrow (?x, a^-, ?y)$$

  We work around this by ignoring all inverses.

- Paths can have a length of at most one under Kleene star, this is handled by truncating to the first symbol. For example:

  $$(?x, ?y) \leftarrow (?x, (a \cdot b)*, ?y)$$

  Can not be expressed in openCypher and is simplified to:

  $$(?x, ?y) \leftarrow (?x, a*, ?y)$$

  Which in openCypher look like:

  ```
  MATCH(x)-[:a*]->(y) RETURN x.id, y.id
  ```

- openCypher has slightly different semantics for pattern matching. When we take a graph $G = (V, E)$ with $|E| = 2$ as follows:

  $$\begin{array}{ccc} 1 & 0 & 2 \\ 1 & 0 & 3 \end{array}$$

  And execute against it the simple query:

  $$(?x, ?y, ?z) \leftarrow (?x, 0, ?y), (?x, 0, ?z)$$

  In DatalogSystem, Postgres, and SPARQLSystem the result set is:

  $$\begin{array}{ccc} 1 & 2 & 2 \\ 1 & 2 & 3 \\ 1 & 3 & 2 \\ 1 & 3 & 3 \end{array}$$

However in GraphSystem the result is:

1 3 2
1 2 3

This effectively means GraphSystem does not allow y and z to point to the same node identifier.

These are simply limitations of the openCypher language, so we will have to accept this and account for it in our experiments.

For some concrete examples of queries translated in each language please refer to Appendix A.

2.4 Evaluation of data generated by gMark

The previous section described how gMark can be used to generate data for our benchmark. In this section we will give a brief analysis of the quality and properties of the data.

Table 2.2 shows execution times of gMark for generating graphs based on the three schemas described in Section 2.3 and four increasing graph sizes. The reported numbers are averages of 3 separate executions, and were acquired on the system described in Table 3.1. The size of a graph in gMark is indicated by the number of nodes, so the difference between the generation times for different schema definitions can be explained by the density of the graph. The most important implication is that generation graphs will be no issue, since the largest graph size used in the benchmark study is 64K.

	100K	1M	10M	100M
Biblio	0m0.057s	0m0.638s	0m8.344s	1m28.725s
LDBC	0m0.225s	0m1.451s	0m23.018s	3m11.318s
Watdiv	0m2.163s	0m25.032s	4m10.988s	113m31.078s

Table 2.2: Execution time of gMark to generate graphs

In Table 2.3 the actual number of nodes and edges is displayed for three different sizes for the schemas Biblio, LDBC, and Watdiv. The number of nodes for LDBC and Watdiv are quite accurate, however for Biblio there is a large discrepancy. Compared to the number of nodes Watdiv has the fastest growing number of edges, indicating that the graph is more dense, which again explains the generation time difference in Table 2.2.

Schema	**Size**	**#Nodes**	**#Edges**
Biblio	10K	5679	16726
Biblio	100K	52337	167154
Biblio	1M	494831	1673839
LDBC	10K	10172	753638
LDBC	100K	100172	11601280
LDBC	1M	1000172	251748503
Watdiv	10K	10318	501999
Watdiv	100K	100318	15226729
Watdiv	1M	1000318	466858577

Table 2.3: For the three gMark schemas, the input size (which should incidate the number of nodes) and the actual number of nodes and edges.

As explained in Section 2.3.1 the schema definition contains a list of types and predicates, and specifies distributions of edges between nodes of each type. To verify the accuracy of these

distributions we generated a graph based on the Biblio schema and analyzed the distributions. The histogram in Figure 2.3 shows the distribution of in-degrees of edge type *authors*, which connects *researchers* to *papers*. In the schema definition the in-distribution has been set to Gaussian with $\mu = 3$ and $\sigma = 1$, as we can see in the histogram this is indeed quite accurate. The count for degree 0 is not shown because this also includes all vertices not of type *paper*, this is because the graph that gMark creates only has id's for nodes, so we can not know which type it has. The out-degree of *authors* is defined as Zipfian with parameters $\alpha = 1.5$ and plotted in Figure 2.4.

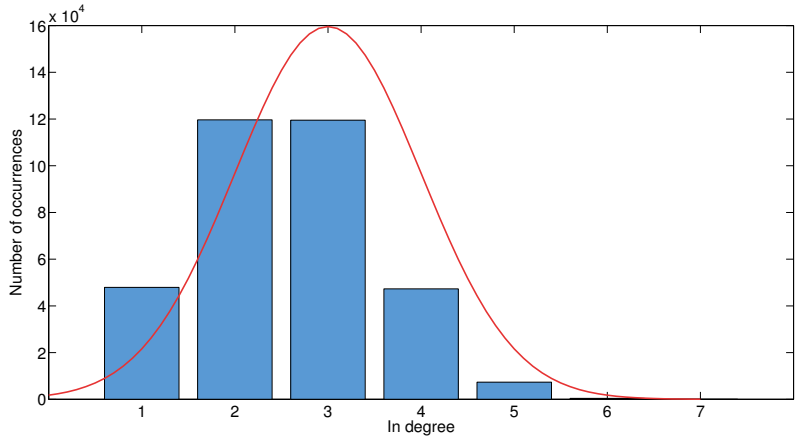

Figure 2.3: Histogram of the in-degree of the *authors* relation in Biblio scenario. The red line represent the intended distribution from the schema definition which is Normal $\mu = 3, \sigma = 1$.

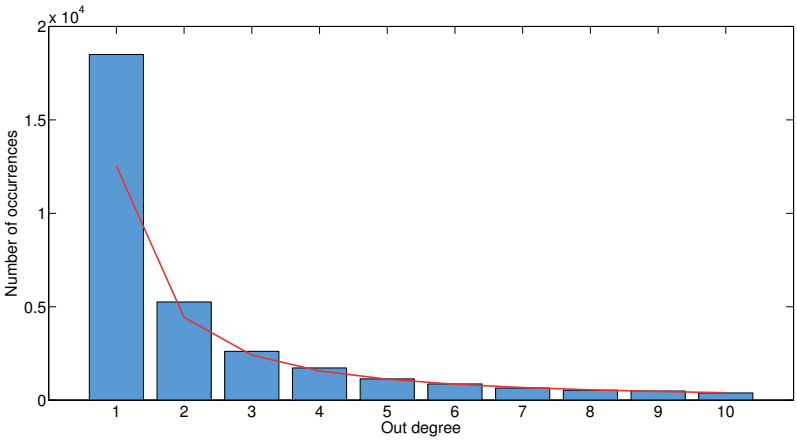

Figure 2.4: Histogram of the out-degree of the *authors* relation in Biblio scenario.

Figure 2.5 shows the in degree of *published-in*, which should be $\mu = 4$ and $\sigma = 1$. Finally Figure 2.6 shows the out distribution of published-in, which is supposed to be uniform(0,1). The distribution between 0 and 1 is wrong here, but no higher values than 1 occur.

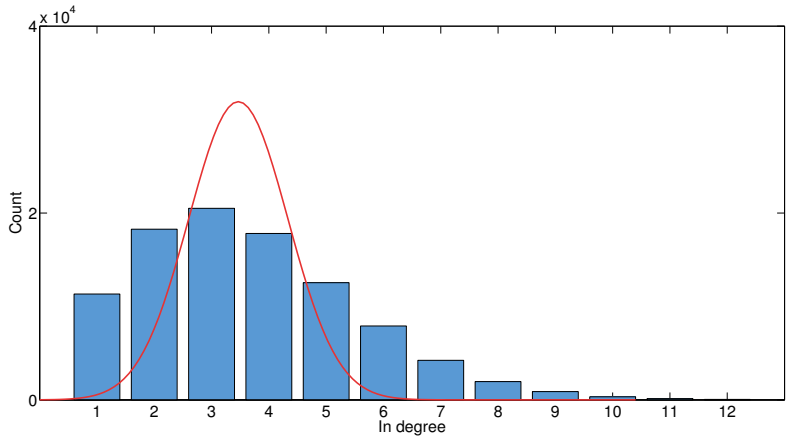

Figure 2.5: Histogram of the in-degree of the *published-in* relation in Biblio scenario.

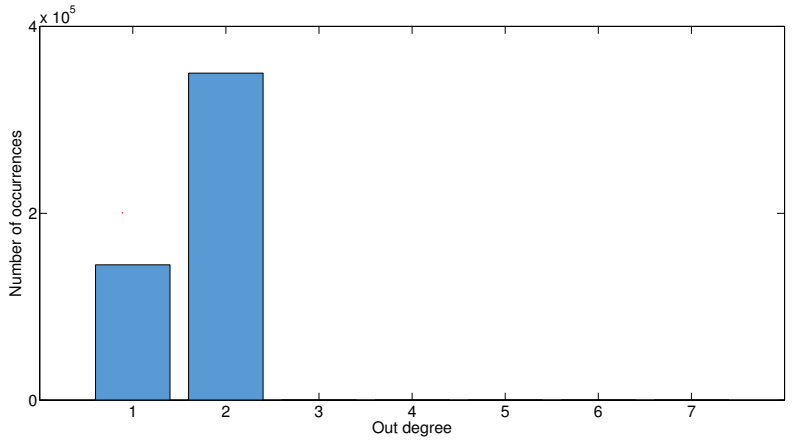

Figure 2.6: Histogram of the out-degree of the *published-in* relation in Biblio scenario.

gMark uses a best-effort heuristic to generate data satisfying the constraints specified in the schema definition. Some degree of error is therefore to be expected, however overall the results are very reasonable. The degree distributions we have seen are close to the target, and the total number of vertices does not have to be exact as long as it scales linearly(generating a graph twice as large has twice as many vertices). We conclude that the gMark graph generation has a sufficiently high quality for our intended purposes.

2.5 Related work

A comparison between a graph database(Neo4j) and a relational database(MySQL) is reported in [11], concluding that the graph database performs better on queries of a structural type. An empirical comparison between four graph databases using a custom benchmark framework is made in [12]. Instead of analyzing graph databases by running benchmarks on them, a systematic comparison is made in [13] by reviewing a number of systems by their underlying graph models, storage features, and the query language operation and manipulation features.

There are several works discussing synthetic graph data generation and proposing new solutions, such as the Waterloo SPARQL Diversity Test Suite (WatDiv) [3], Linked Data Benchmark Council LDBC [4], and the SPARQL Performance Benchmark (SP2Bench) [5]. WatDiv features a data generator based on a schema and a query generator based on a template. This allows for some degree of freedom by customizing distributions in the graph, making Watdiv more diverse than other benchmark tools, however query generation requires a data set instance so is not entirely schema based. LDBC introduces a synthetic graph generator based on social networks, and includes three fixed workloads for which the parameters are acquired by mining the dataset. SP2Bench is a language specific benchmark platform for SPARQL queries over RDF. Other examples of synthetic benchmarks include The Berlin SPARQL Benchmark [6] and LUBM [7], however in this work we will be using gMark [2]. The advantage of gMark is that graph generation is entirely schema based, and that this schema is very flexible allowing us to define custom nodes, relations, proportions, and distributions. The query generation is also schema based, and does not depend on an instance of the graph unlike other benchmark tools. Several schemas are available, including ones that actually model LDBC and WatDiv. gMark is also language independent; it includes query translators to different languages including SPARQL and SQL.

Chapter 3

Benchmark on non-recursive query workloads

This chapter contains the main contribution of this work, the benchmark study. Here we will stress test the four database systems on various sets of queries and increasing graph sizes. The queries used are all non-recursive, recursive queries are treated separately in Chapter 4. A detailed discussion of how the benchmark is set up is given in Section 3.1. The results are presented and discussed in Section 3.2.

All experiments are performed on a system with a Intel Core i6 and 6GB of main memory running on Ubuntu 14.04 as described in Table 3.1.

Processor	Intel Core i7 920
RAM	6GB
Operating System	Ubuntu 14.04 64 bit

Table 3.1: System specifications

3.1 Parameters

The benchmark study will be performed on data generated by gMark based on the Biblio schema as described in Section 2.3. For the execution of each single query we measure two things: the number of rows returned, which should be the same across all systems, and the time it took to return the result. Unless indicated otherwise execution times will be expressed in seconds(s). Furthermore we will use the following settings and variables during the benchmarking procedure:

- **Graph sizes**. Different sizes of graphs are used, this way we can analyze the behavior of each system e.g. how much the graph size affects the execution time of a query. The size of a graph is indicated by the number of nodes, because this is how the size is indicated in gMark. We use the sizes:
  - 2000
  - 4000
  - 8000
  - 16000
  - 32000
  - 64000
- **Query classes**. The class of a query describes the range of properties it can have as described in the schema definition, see Section 2.3.1. We have the following three classes of non-recursive queries:
  - **Len**. Varying path lengths, no disjuncts, no conjuncts.
  - **Dis**. Disjuncts, no conjuncts.
  - **Con**. Conjuncts and disjuncts.
- **Selectivity classes**. The selectivity of a query indicates how the number of returned results grows with the size of the graph. The 3 selectivity classes are:
  - **Constant**. Number of results does not grow significantly when graph size grows.
  - **Linear**. Number of results scales linearly with the graph size.
  - **Quadratic**. Number of results scales quadratically with the graph size.
- **Number of queries**. As described in the above points a query has both a class and a selectivity class. We can use gMark to generate 10 instances of each combination, giving us 90 queries in total.
- **Repetitions**. Each single query execution is repeated 6 times to account for fluctuations in system performance. From these 6 the first 'cold' run is dropped. Then from the remaining 5 'warm' executions the highest and lowest times are dropped. The final result is the average of the remaining three values. Between the executions of different queries the databases are closed and reopened to clear any caching.

Summarizing, we generate a list of 90 queries, that are evaluated 6 times each on graphs of 6 different sizes, on 4 different systems.

3.2 Results

Figure 3.1 shows the average query execution times on constant queries. The labels on the x-axis are combinations of the 4 systems ((PostgreSQL(P), SPARQLSystem(S), GraphSystem(G) and DatalogSystem(D)) and the 3 query classes Len, Dis, Con. For each of these 12 combinations there is a group of 6 colored bars indicating the average execution time on each graph size. Execution

times are measured in seconds, note that the vertical axis is log scale. The figure indicates that P has the best average execution times overall. However, S and D do have a better behavior as the graphs grows. Queries of the class Con are more difficult than both Len and Dis for all systems.

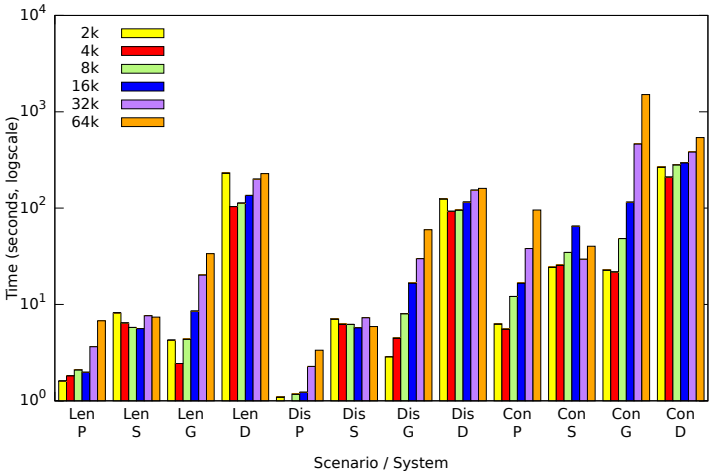

Figure 3.1: Histogram showing average execution times of constant queries. The bars are grouped by query class and database system, giving the result for each of the six graph sizes.

Figure 3.2 shows the same picture for queries of linear selectivity class. It shows the behavior on these queries is very similar to constant queries of Figure 3.1. P has the best overall performance, while S and D scale better. Query classes Len and Dis behave similar, Con is more difficult.

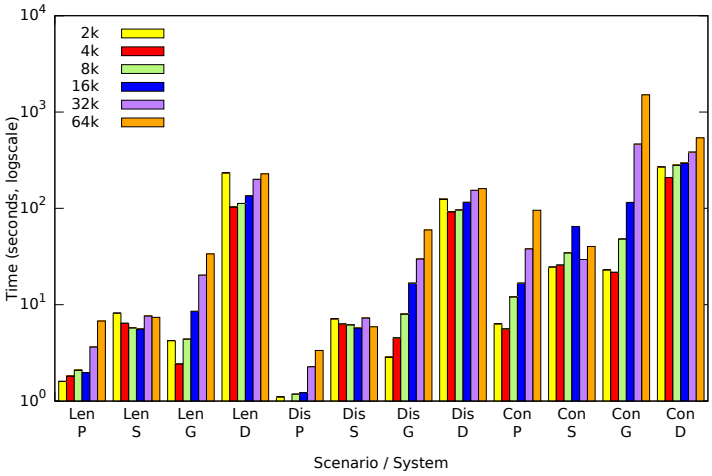

Figure 3.2: Histogram showing average execution times of linear queries.

In Figure 3.3 we can see the results on quadratic selectivity queries. These queries are much more difficult for each of the systems (note the difference in y-scale w.r.t. the other figures). Also the growth of execution time as the graph size increases seems to be exponential for all systems. There is no clear winner here but both S and D seem to perform better than P and G.

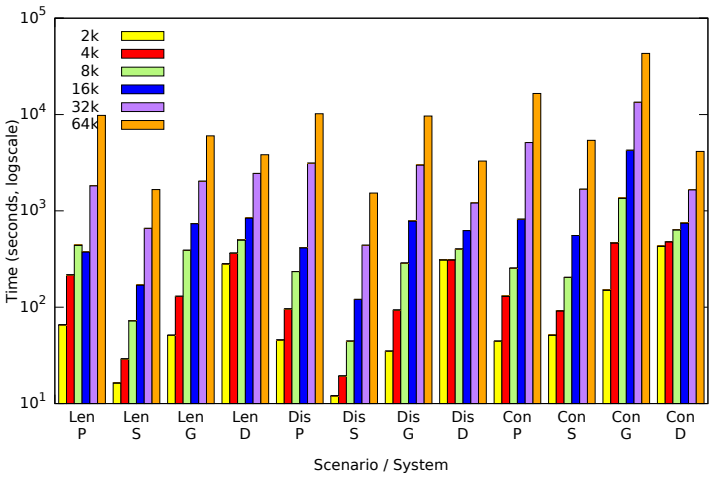

Figure 3.3: Histogram showing average execution times of quadratic queries.

3.3 Summary

We have performed a benchmark study on four database systems, using graphs of varying sizes based on the Biblio scenario.

Three classes of queries were used, of which the base case is a path of varying length. Adding disjuncts to the queries did not have a significant impact, however adding more conjuncts did noticeably increase query execution time.

But the most important result of this chapter is that none of the graph databases perform better than the relational database Postgres. This is very surprising, since graph databases are designed for this task. One might wonder why graph databases should be used at all, besides for the benefit of having a more natural query language. Our final conclusion is therefore that there is still a lot of progress to be made in the field of graph databases.

Chapter 4

Benchmark on recursive query workloads

This chapter provides a detailed study on recursive queries. These are treated separately because the behavior and performance on our database systems is very different. We will first discuss the implementation and the parameters we used, then we will analyze the results. Finally we will draw our conclusions and determine which system has the overall best performance on recursive queries.

4.1 Implementation and limitations

This section discusses how recursion is implemented in each language, some of the limitations, and how we deal with these limitations. Since it is a superset of Datalog, recursive queries can be naturally expressed in CustomDatalog. In SQL:2011 we can use the `WITH RECURSIVE` keyword which is supported by Postgres. GraphSystem only partially supports recursion, as discussed in Section 2.3.3, we can only have a single symbol under a star in openCypher. We work around this by truncating to the first symbol. SPARQLSystem claims to implement SPARQL 1.1 in which we can express recursion by simply using a star. But there's a catch, SPARQLSystem needs a starting points for the recursion. For example the recursive SPARQL query:

```
SELECT DISTINCT ?x0 ?x1 WHERE { { ?x0 (:p0)* ?x1 . } }
```

Results in an error because there is only one expression under a star and no starting point. This actually means that SPARQLSystem does not fully implement SPARQL 1.1. We can however easily fix this by expressing the query as:

```
SELECT DISTINCT ?x0 ?x1 WHERE { { ?x0 ?y ?z . ?x0 (:p0)* ?x1 . } }
```

Which is semantically the exact same query, but internally selects all edges as starting point for recursion. Summarizing, we can translate recursive UCRPQ queries to each query language of the respective systems, with the limitation of openCypher only supporting a single symbol under a star. Examples translations of recursive queries in each language are given in Appendix A.

4.2 Parameters

The benchmark performed in this chapter is very similar to those in Chapter 3, but with recursive queries. We will use the same setup for each of the systems to report for each query the number of results and execution time in milliseconds, which is again the average of the last 5 of 6 total runs. Furthermore, we will use the following parameters:

- **Graph sizes** The same graph sizes are used but we only go up to $32k$ because recursive queries are often harder to solve, and we would therefore get too many failures on 64k. The graphs are generated by gMark based on the Biblio schema definition.

- **Query classes** In the benchmark study query classes Len Dis and Con were used, and selectivity classes constant linear and quadratic. Neither of these will be used in this benchmark, instead queries are categorized by the following recursive structures:

  - **Single** A single symbol under a star.

    $$(?x, ?y) \leftarrow (?x, (a)*, ?y)$$

  - **Concat** A path of two or more symbols under a star.

    $$(?x, ?y) \leftarrow (?x, (a \cdot b)*, ?y)$$

  - **Disj** One or more disjunctions of single symbols under a star.

    $$(?x, ?y) \leftarrow (?x, (a + b)*, ?y)$$

  - **Conj** One or more conjuncts outside of the star.

    $$(?x, ?z) \leftarrow (?x, (a)*, ?y), (?y, b, ?z)$$

  - **DisjConcat** Combination of Concat and Disj.

    $$(?x, ?y) \leftarrow (?x, ((a \cdot b) + (c \cdot d))*, ?y)$$

  Having these classes can help us in determining why recursive queries are more difficult to compute. Since we truncate to the first symbol under a star in GraphSystem we basically only have classes Single and Conj for this system. Therefore we will be limited in comparing this system to the others.

- **Number of queries** We will have size queries of each class, giving a total of $6 \cdot 5 = 30$ queries.

4.3 Results

In this section we will discuss the results of the benchmark study on recursive queries.

Recall from Section 4.1 that we had to rewrite queries in SPARQLSystem because the system requires an internal starting point for the recursion. We did this by selecting all edges and assigning this to the starting variable. It turns out that this fix does not works for queries of class DisjConcat, we can give no reasonable explanation for this behavior, especially because Disj and Concat do succeed separately. SPARQLSystem did successfully compute the results for the four other classes, so from we will discuss these and ignore DisjConcat.

We will see in the results that we will shortly discuss, that Postgres is relatively slow on recursive queries. On several occurrences over 10 minutes is required so compute a single query, there were even two queries that failed on graph size 32k because of an out of memory exception, these queries will be ignored in the presented figures.

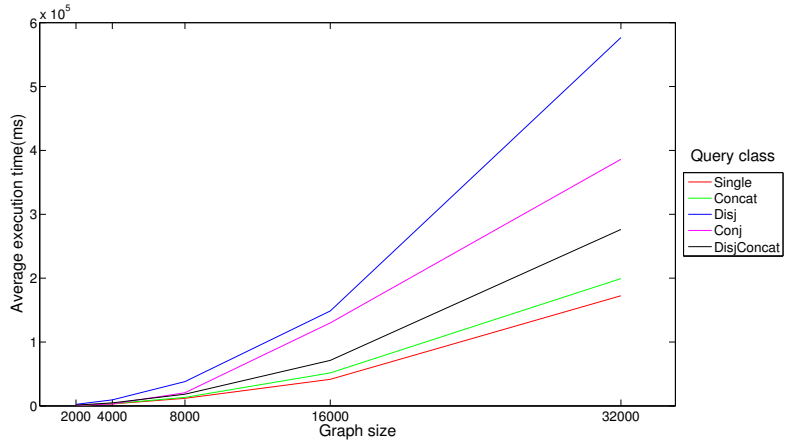

Figure 4.1: Plot of the average execution time of each query class in Postgres.

Figure 4.1 shows the execution times of queries for each graph size, where the results are grouped and averaged by class. Class Concat unsurprisingly appears to be the easiest class, followed by Concat. The longest execution times are taken by Disj. Interesting to not is that DisjConcat is faster than Disj. Overall we can see that Postgres scales quadratically with respect to graph size.

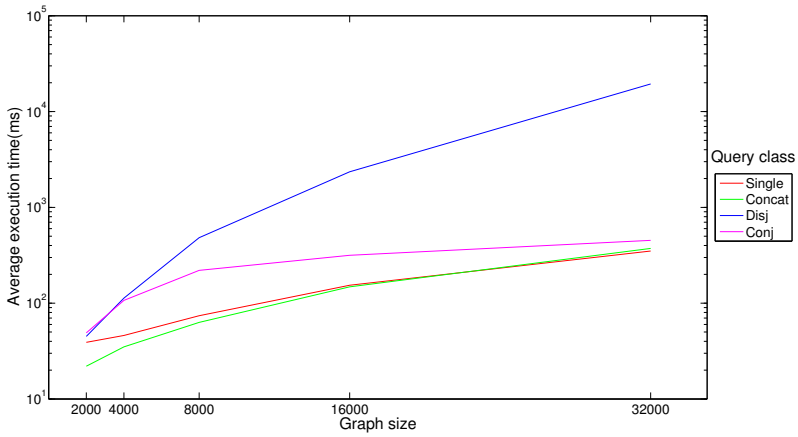

Figure 4.2: Plot of the average execution time of each query class in SPARQLSystem. Class 4 is missing because these queries can't be computed. The y-axis has log-scale.

The same plot is drawn for SPARQLSystemin Figure 4.2. Note that class DisjConcat is missing because these queries can not be computed. Likewise what we saw for Postgres the Disj queries take most time, so much even that the y-axis has to be drawn in log scale to ensure all data is visible. The three remaining classes are relatively close together. Although it is a bit harder to see because of the log y-axis, the recursive queries on SPARQLSystem scale quadratically to graph size.

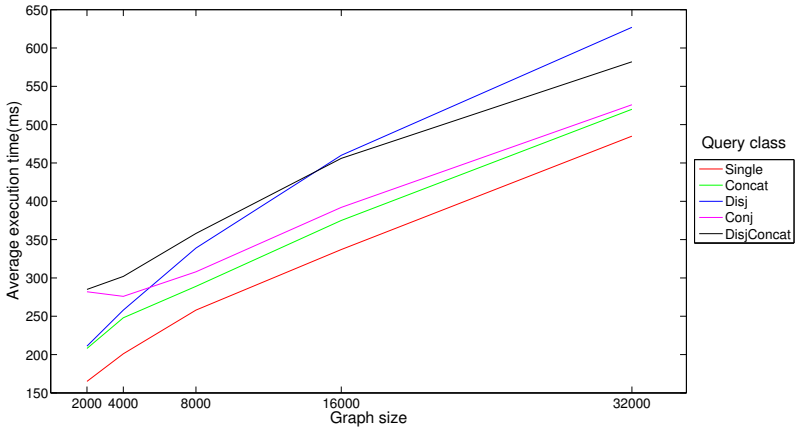

Figure 4.3: Plot of the average execution time of each query class in DatalogSystem.

Again the same plot, this time for DatalogSystem is shown in Figure 4.3. The disjunctive classes Disj and DisjConcat have the highest average execution time, although the difference towards the other classes is much smaller than it was on the other systems. DatalogSystem seems to scale very well compared to Postgres and SPARQLSystemsince most of the classes are close to linear.

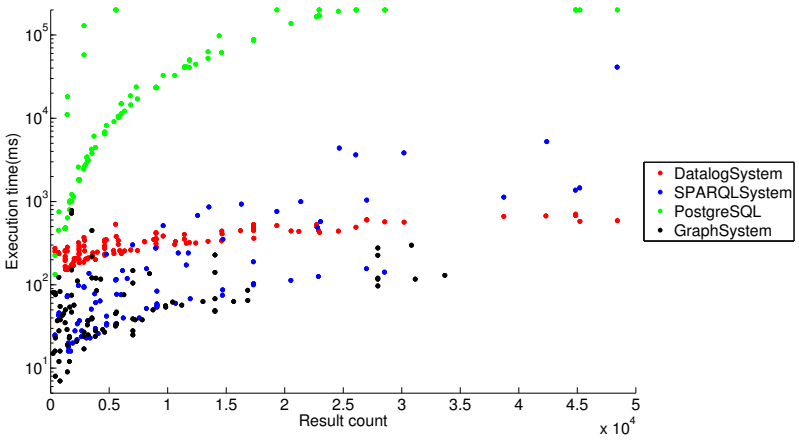

Figure 4.4: Scatter plot of result count versus execution time. The y-axis is in log scale.

The scatter plot in Figure 4.4 combines the results of the systems, but instead of using graph size as x-axis, we use the size of the result set. The y-axis indicates the execution time using log scale. For all systems there is a strong correlation between the result size and time. Postgres is by far the slowest and scales worst, the timings are up to two orders of magnitude larger than those of the other systems. GraphSystem seems to perform well but can not be fairly compared because of the different queries, this is also visible in the figure; the data points do not line up with the other systems on the x-axis. DatalogSystem and SPARQLSystem are somewhat comparable, SPARQLSystem is faster when the result size is smaller, however DatalogSystem scales better just like we have already discussed and seen in Figures 4.3 and 4.1.

4.4 Discussion

In the previous section we demonstrated that Postgres performs poorly on recursive queries compared to the other systems. Recursive SQL queries are expressed using so called common table expressions in which a table can refer to itself. An example calculating the sum of the integers 1 to 100:

```
WITH RECURSIVE t(n) AS (
  VALUES (1)
UNION ALL
  SELECT n+1
  FROM t
  WHERE n < 100
)
SELECT sum(n) FROM t;
```

According to the Postgres documentation on recursive queries [14] this is executed as follows:

1. Evaluate the non-recursive term. Include all remaining rows in the result of the recursive query, and also place them in a temporary working table.
2. So long as the working table is not empty, repeat these steps:
   (a) Evaluate the recursive term, substituting the current contents of the working table for the recursive self-reference. Include all remaining rows in the result of the recursive query, and also place them in a temporary intermediate table.
   (b) Replace the contents of the working table with the contents of the intermediate table, then empty the intermediate table.

The second step is quite an expensive operation. In our scenario where we do not know the number of recursive steps and where the queries can have high selectivity, this can become a problem.

Another way to find information about the performance of Postgres is the SQL EXPLAIN command, which gives us information about the query execution plan. The recursive part of a query unfolds as follows:

```
->  Recursive Union  (cost=0.00..191385945.16 rows=5419966614 width=8)
  ->  CTE Scan on c0 c0_1  (cost=0.00..2082.28 rows=104114 width=8)
  ->  Merge Join  (cost=165535.89..8298453.06 rows=541986250 width=8)
    Merge Cond: (head.trg = tail.src)
    ->  Sort  (cost=12183.54..12443.82 rows=104114 width=8)
      Sort Key: head.trg
      ->  CTE Scan on c0 head  (cost=0.00..2082.28 rows=104114 width=8)
    ->  Materialize  (cost=153352.35..158558.05 rows=1041140 width=8)
      ->  Sort  (cost=153352.35..155955.20 rows=1041140 width=8)
      Sort Key: tail.src
```

We can see the operations that are applied, including an estimate of the given cost. In each step of the recursion a sort and a join is performed, causing the estimated cost to explode.

The intended use case for recursion in SQL in hierarchical data, such as tables with subcomponent relations. This type of recursion is only a couple of layers deep, whereas in our use case the recursion can go as deep as spanning the entire graph. Therefore we conclude Postgres had bad performance on recursive queries over graph data because SQL was never designed to do so.

4.5 Summary

We have performed a benchmark study using recursive queries of five classes on the four database systems DatalogSystem GraphSystem PostgreSQL, and SPARQLSystem.

There is a wide variance in performance between all systems on recursive queries, and generally recursive queries are more difficult than non-recursive queries. We have seen that especially recursion over a disjunction is a hard problem for all systems.

The execution times of GraphSystem were reasonable, but this was partially because we could not express most of the recursive queries in openCypher. Therefore the results are too weak to draw a fair comparison. Although we managed to express all queries in SQL, Postgres was simply too slow with computation times of over 10 minutes on a single query and even some failures. We conclude that SQL is not suitable for recursive queries over graph data. We found out that SPARQLSystem does not fully implement SPARQL 1.1, so we had to adapt our queries to this which enabled us to execute the majority of the queries. SPARQLSystem turned out to perform well, however it did not scale very well for graph size. Finally, the Datalog like syntax of DatalogSystem naturally support recursion, query execution times are quite fast, and of all systems DatalogSystem scales best. Therefore we conclude DatalogSystem is the clear winner on recursive queries.

Chapter 5

Modeling UniProt protein database in gMark

In the previous chapters we demonstrated how gMark can be used as a benchmark generation tool based on some schema definition that models a certain domain. The schemas we used (Biblio, LDBC, Watdiv) were predefined and supplied by gMark. In this chapter we will attempt to model a new domain by ourselves, and experimentally verify its effectiveness. The schema we will construct is based on (a subset of) the Uniprot [8] database, more specifically the UniProtKB Swiss-Prot dataset [1]. The Universal Protein Resource (UniProt) is a collaboration between three institutions that aims to bring together knowledge about protein sequences, functions, and other information. A protein is basically a large molecule consisting of chains of amino acids that plays essential roles in living organisms such as DNA replication. For this work however no understanding of biochemistry is required, we will regard this data just like we would any graph structure.

The remainder of this chapter is structured as follows, in the first part we will define a model for the Uniprot domain and mine the Uniprot database to discover relative proportions of node types and distributions of edge degrees. In the second part we will construct a schema definition based on our findings and generate a graph instance using gMark, we will then analyze this graph to verify how well the original graph is simulated, like we did in Section 2.4.

[1] http://www.uniprot.org/uniprot/

5.1 Analysis of UniProtKB data

In this section we gather all the meta data from the UniProt graph required to construct a gMark schema definition. We will start by determining the node types and edge labels. Then we will count for each of the types and relations the number of occurrences in the graph in order to establish their relative proportions. Finally we will gather statistics about the in and out degrees for each relation, and try to fit a statistical distribution to them.

5.1.1 Domain model

The UniProt database contains millions of proteins that are annotated with hundred of properties and relations. Modeling the entire domain would take too much time, so we choose to only consider a subset consisting of 7 node types and 7 relations. We choose nodes that are occur frequently enough in the dataset, and can be modeled as distinct entities. Many attributes are descriptive properties, links to other databases, or occur too infrequently to perform a sound statistical analysis on. Therefore we believe this subset is a sufficiently accurate representation of the UniProt domain for our purposes. The model including the node types and their relations are represented in Figure 5.1, the 7 types of nodes we consider are:

- **Protein.** The main entity in the Uniprot database is the *protein*, a large molecule occurring in living organisms. A *protein* can have an *Interacts* relations to another *protein*.
- **Gene.** A region of DNA that stores the genetic code of a *protein*.
- **Organism.** A living being in which a certain *protein* has been found. Can be animals but also bacteria or viruses.
- **Article.** Scientific article referencing a *protein*.
- **Keyword.** Term describing an important property or categorization of a *protein*, such as *transmembrane helix* or *host membrane*, but the meaning of these terms is not relevant to our interests.
- **Author.** Person that has contributed to writing an article.
- **Journal.** An *article* has usually been published in a *journal*.

A visual representation of these types and the relationships they can have to one another is given in Figure 5.1.

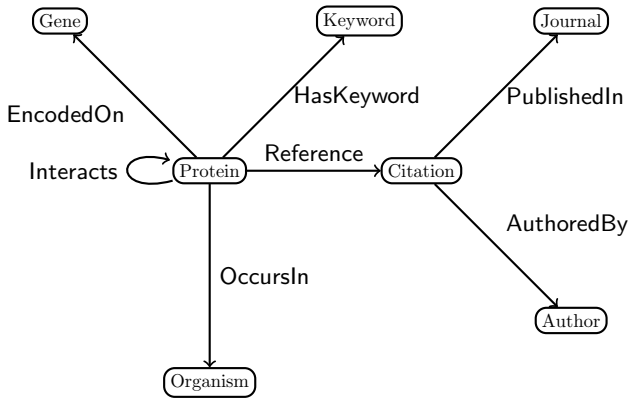

Figure 5.1: The node types and relations in the UniProt scheme.

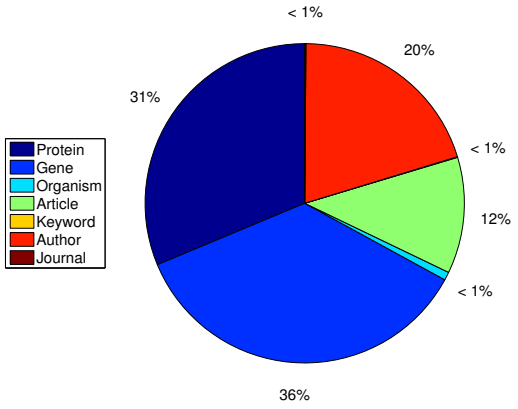

Figure 5.2: Pie chart displaying the relative count of each vertex type.

5.1.2 Type proportions

The schema definition in gMark requires a list of types and for each type a proportion, which can either be defined as a ratio of the total number of nodes or as a fixed value. The relative occurrence of the seven node types discussed in the previous section is displayed in Figure 5.2. We decide to implement the proportions of node types Journal, Organism, and Keyword as fixed values because when the database grows by adding new proteins, it will probably refer to the existing Keywords and Organisms, and its articles will most likely be published in existing journals. The proportions of the other types will likely remain roughly the same as the graph grows, so will be implemented as a ratio of the total amount of nodes based on the numbers in Figure 5.1: Protein(32%), Gene(36%), Author(20%), Article(12%).

Type	**Count**	**Proportion**	**Implemented**
Protein	549.646	0,313	0,32
Gene	628.192	0,358	0,36
Organism	14.579	0,008	Fixed(15.000)
Article	206.767	0,118	0,12
Keyword	1.166	0,001	Fixed(1.000)
Author	354.287	0,202	0,20
Journal	2.082	0,001	Fixed(2.000)

Table 5.1: For each type in the UniProt graph the total number of occurrences, the proportion of the total number of nodes, and the implementation in the schema definition. The implementation can be either a proportion or a fixed value.

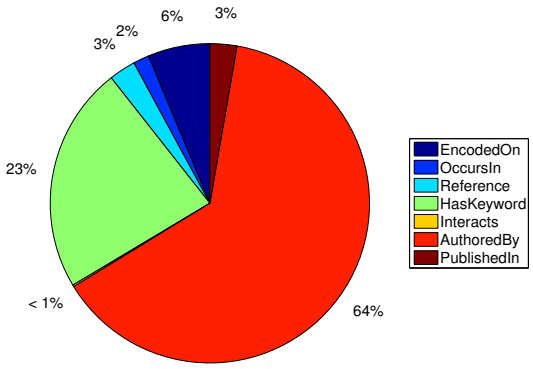

Figure 5.3: Pie chart displaying the relative count of each edge type.

5.1.3 Relation proportions and distributions

The pie chart in Figure 5.3 shows the proportion of relationships in the UniProt dataset, the percentages indicated in this chart can be readily used in the schema. We need not only the proportions of edges in order to create the schema definition, but also the probabilistic distributions of the in and out degree of each relation. We gather statistics from the dataset and then try to fit either Normal(Gaussian), Zipfian, or Uniform distributions to them including their respective parameters. In the remainder of this section we will first discuss the statistics and distributions of the out degrees, and thereafter those of the in degrees.

Edge	**Distribution**	**Parameters**	**Figure**
Interacts	Normal	$\mu = 0,11, \sigma = 1,68$	5.4
EncodedOn	Normal	$\mu = 1,95, \sigma = 0,99$	5.5
OccursIn	Uniform	min=1, max=1	
HasKeyword	Normal	$\mu = 7,13, \sigma = 3,61$	5.6
Reference	Normal	$\mu = 1,68, \sigma = 2,97$	5.7
AuthoredBy	Normal	$\mu = 5,87, \sigma = 4,99$	5.8
PublishedIn	Uniform	min=1, max=1	

Table 5.2: The distributions fitted to out degrees of each relation.

The out degree distributions fitted to the data are shown in Table 5.2. For *EncodedOn*, *HasKeyword*, *Interacts*, *Reference*, and *AuthoredBy* we fitted normal distributions with mean and standard deviation as indicated in the table. The rightmost column links to the Figure in which the data and its matching distribution are plotted. For example to argue that *EncodedOn* has indeed a normal distributed out degree with $\mu = 1,95$ and $\sigma = 0,99$ we refer to figure 5.5 which plots both a histogram of out degree counts and the function of the distribution. This distribution for *EncodedOn* fits quite nicely, however some other distributions are harder to fit. For example Reference does not have a nice Normal behavior but can't be modeled as Zipfian or Uniform either. The two remaining relations, *OccursIn* and *PublishedIn* are both constant 1 and therefore modeled as uniform distributions with min=1 and max=1.

Edge	**Distribution**	**Parameters**	**Figure**
Interacts	Normal	$\mu = 0,00187$ and $\sigma = 0,04321$	5.9
EncodedOn	Zipfian	$\alpha = 3,31$	5.10
OccursIn	Zipfian	$\alpha = 1,62$	5.11
HasKeyword	Zipfian	$\alpha = 0,20$	5.12
Reference	Zipfian	$\alpha = 2,53$	5.13
AuthoredBy	Zipfian	$\alpha = 1,78$	5.14
PublishedIn	Zipfian	$\alpha = 1,37$	5.15

Table 5.3: The distributions fitted to in degrees of each relation.

The in degree distributions are displayed in Table 5.4 in the same manner as we did for the out degrees. Whereas for the out degrees most of the relations were normally distributed, the in degrees are mostly Zipfian. The only relation that is not Zipfian is *Interacts*, for which vertices have either in degree 0 or 1. A uniform distribution is not applicable because in degree 0 far outweighs in degree 1. It would be nice is we could set some proportion for the degree like we did for the node type occurrences, but since this is not possible in gMark we decide to use a normal distribution. All the other in distributions are Zipfian, most of which can be fitted quite nicely. There is one exception; *Keywords*, which is quite difficult to fit as can be seen in Figure 5.12. Normal and uniform distributions would have an even bigger error so Zipfian is the best solution here.

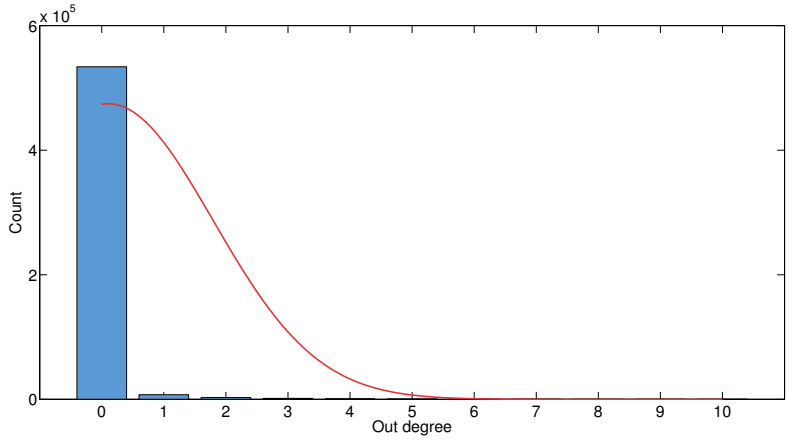

Figure 5.4: Histogram of the out degree of *Interacts*, and the fitted normal distribution with $\mu = 0,11$ and $\sigma = 1,68$.

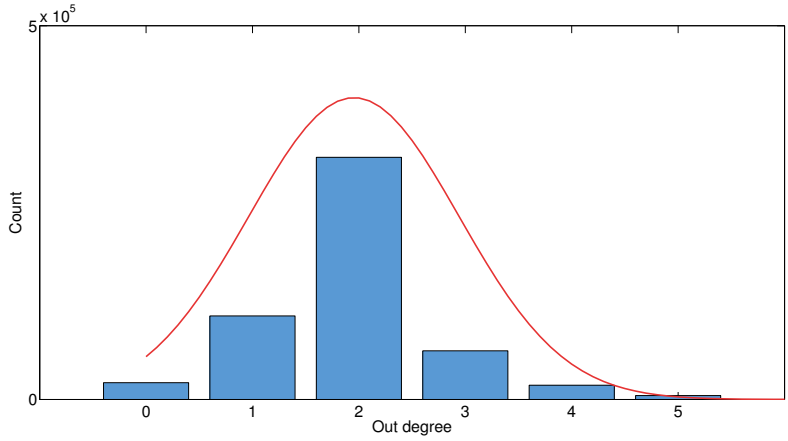

Figure 5.5: Histogram of the out degree of *EncodedOn*, and the fitted normal distribution with $\mu = 1,95$ and $\sigma = 0,99$.

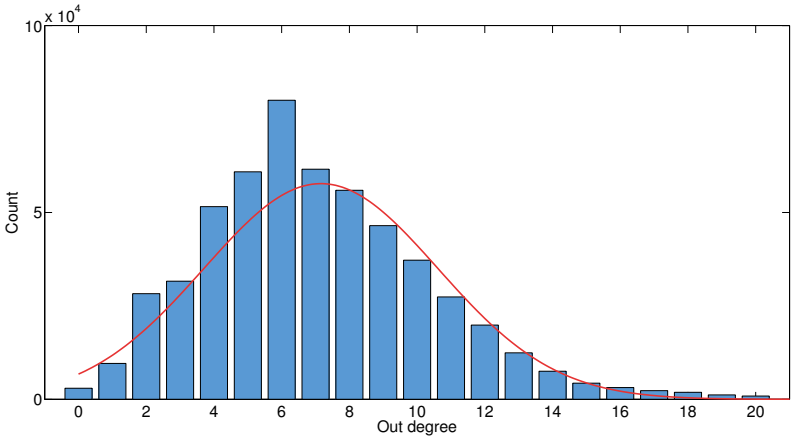

Figure 5.6: Histogram of the out degree of *HasKeyword*, and the fitted normal distribution with $\mu = 7,13$ and $\sigma = 3,61$.

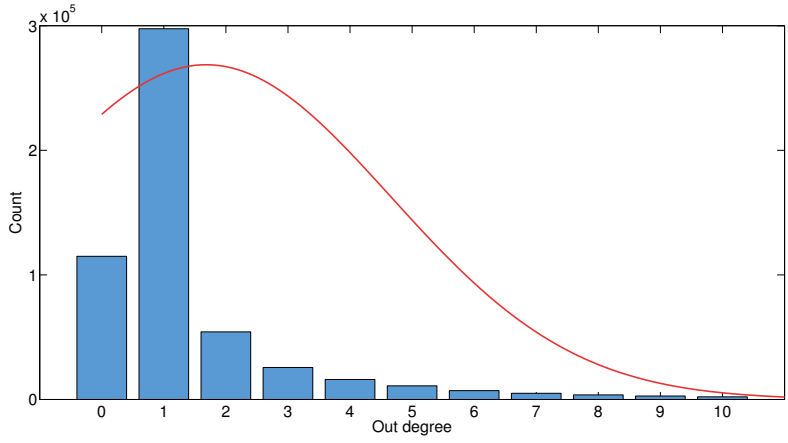

Figure 5.7: Histogram of the out degree of *Reference*, and the fitted normal distribution with $\mu = 1,68$ and $\sigma = 2,97$

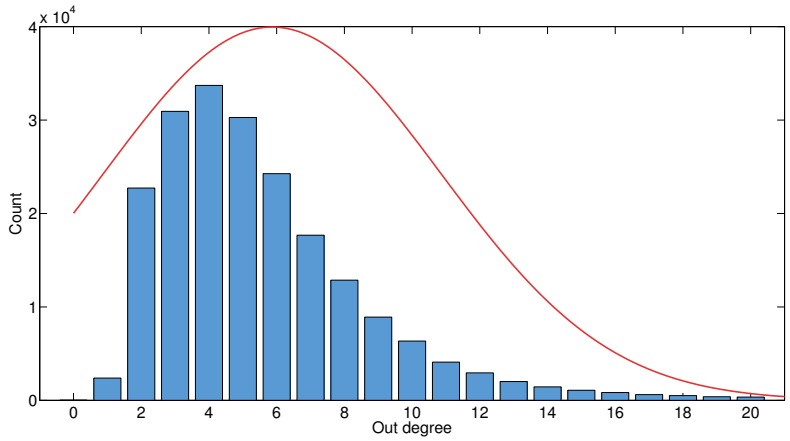

Figure 5.8: Histogram of the out degree of *AuthoredBy*, and the fitted normal distribution with $\mu = 5,87$ and $\sigma = 4,99$.

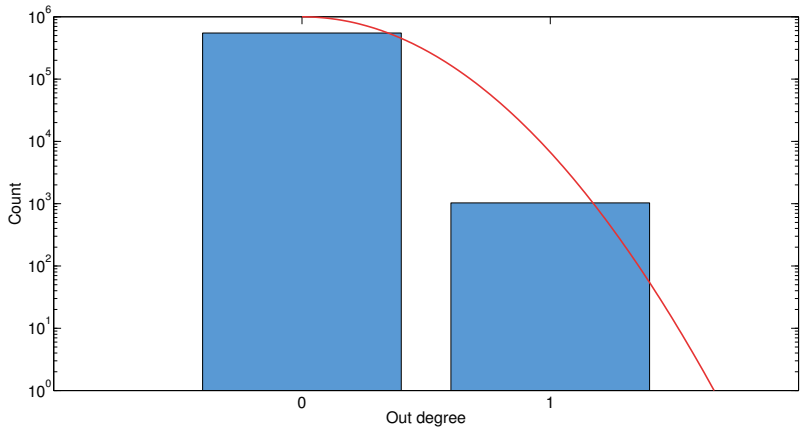

Figure 5.9: Histogram of the in degree of *Interacts*, and the fitted normal distribution with $\mu = 0,00187$ and $\sigma = 0,04321$. Note that the Y-axis is in log scale, the count of in degree 0 and 1 are 548618 and 1028, respectively.

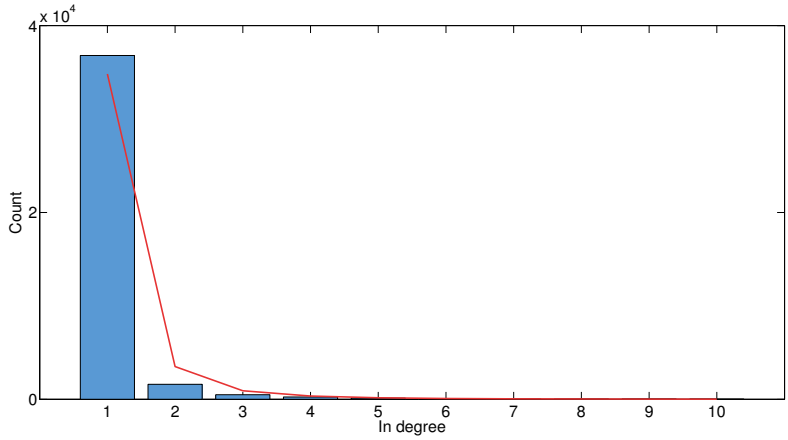

Figure 5.10: Histogram of the in degree of *EncodedOn*, and the fitted Zipfian distribution with $\alpha = 3,31$.

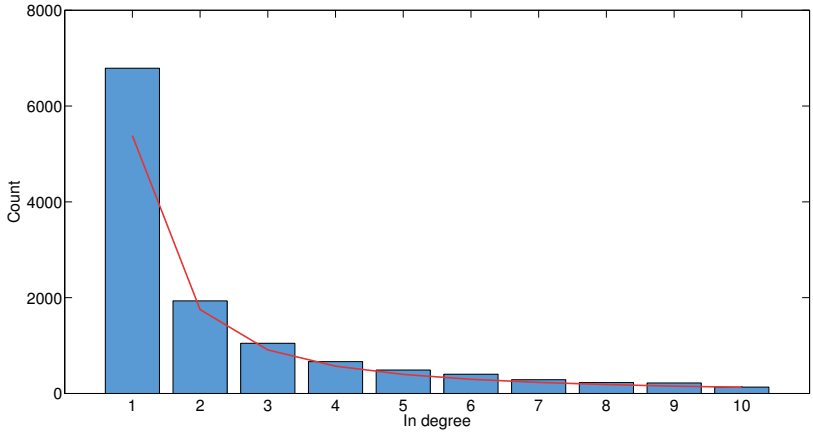

Figure 5.11: Histogram of the in degree of *OccursIn*, and the fitted Zipfian distribution with $\alpha = 1,62$.

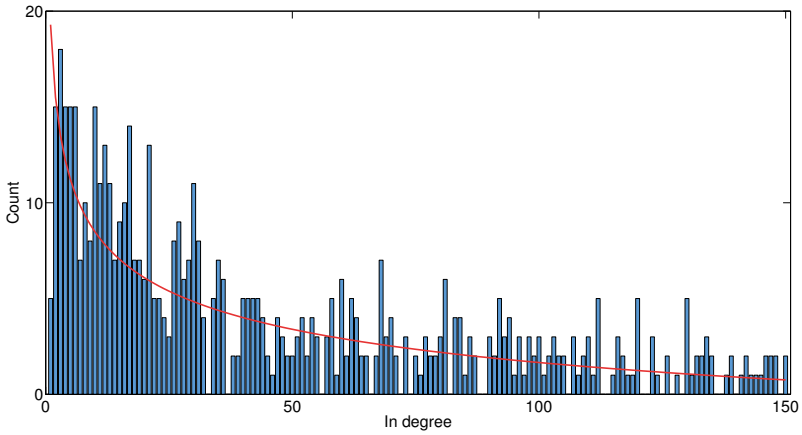

Figure 5.12: Histogram of the in degree of *HasKeyword*, and the fitted Zipfian distribution with $\alpha = 0, 20$.

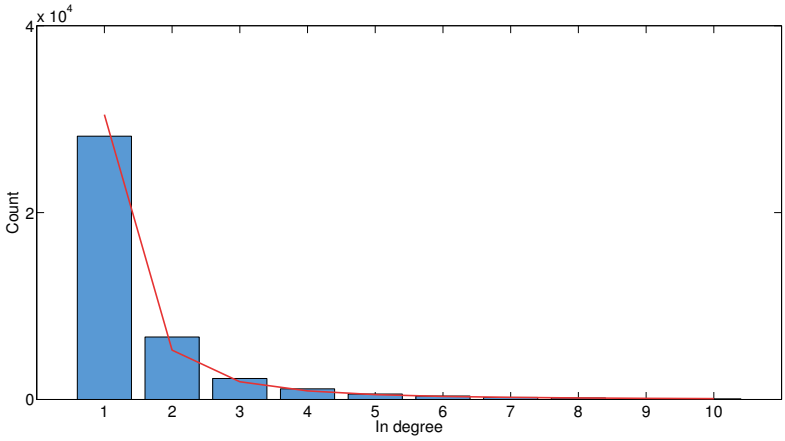

Figure 5.13: Histogram of the in degree of *Reference*, and the fitted Zipfian distribution with $\alpha = 2, 53$.

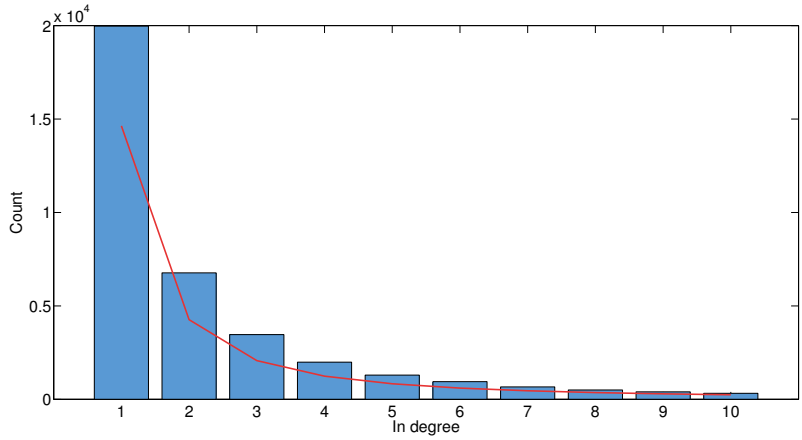

Figure 5.14: Histogram of the in degree of *AuthoredBy*, and the fitted Zipfian distribution with $\alpha = 1,78$.

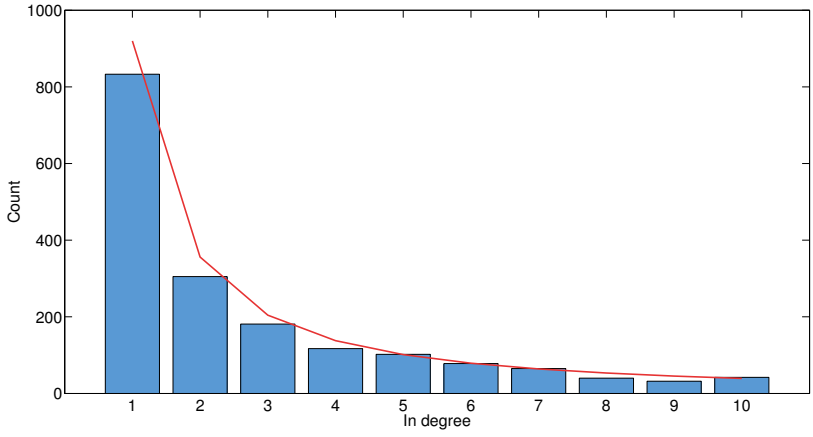

Figure 5.15: Histogram of the in degree of *PublishedIn*, and the fitted Zipfian distribution with $\alpha = 1,37$.

5.2 Evaluation of generated data

Given the types, relations, proportions, and distributions determined in the previous section, generating the schema definition in gMark is straightforward. The actual XML file is included in Appendix B. Besides this schema file gMark needs a number n for the size of the graph. The original UniProtKB dataset had roughly 1.7 million nodes, we will also take this as the value of n. For the sake of brevity from now on we will refer to the original UniProtKB graph by G_{up} and to the graph generated by gMark by G_{gm}. To gather statistics from G_{gm} we insert the graph data into a Postgres database like did in the benchmark study, having a table called edge with columns source, edge, and target. We can then execute SQL queries to get the statistics we require, for example:

- To calculate the number of edges we can simple count the number of relations in the edges table.

```
SELECT COUNT(*)
FROM edge
```

- To get the total number of vertices we combine the sources and targets.

```
SELECT COUNT(node)
FROM
(
  SELECT DISTINCT src AS node
  FROM edge
  UNION
  SELECT DISTINCT trg AS node
  FROM edge
) AS derivedTable
```

- To determine the out degrees, count for each vertex the number of outgoing edges grouped by their label.

```
WITH nodes(node) AS (
  SELECT node
  FROM (
    SELECT DISTINCT src AS node
    FROM edge
    UNION
    SELECT DISTINCT trg AS node
    FROM edge
  )
),
labels(label) AS (
  SELECT DISTINCT label
  FROM edge
),
combined(node,label) AS (
  SELECT nodes.node AS node, labels.label AS label
  FROM nodes, labels
)
SELECT combined.node, combined.label, COUNT(edge.src)
FROM combined
LEFT JOIN edge ON edge.src = combined.node AND edge.label = combined.label
GROUP BY combined.node, combined.label
```

```
ORDER BY combined.node, combined.label
```

5.2.1 Analysis of proportions

The generated graph G_{gm} has $|V| = 881.809$ and $|E| = 5.739.431$, so the number of nodes is only half of the $n = 1.7$ million we wanted. We do not know why gMark does not generate a larger graph, but it is not an issue for us since we will mostly discuss proportions and distributions and not concrete values. As discussed in Section 2.3 gMark generates graphs in CSV format as triples (sourceID, label, targetID), so in the graph file one line might look like:

```
1343,5,201
```

One major implication of this format is that we can no longer verify the type of a node, we know only it's ID. Therefore we can not evaluate the quality of gMark on the proportions of node types. We can however still check the edge proportions, because there the label identifier is maintained. In Table 5.4 we can see for each relation the number of occurrences and it's proportion in the both G_{up} and G_{gm}. The first thing to note is that the *Interacts* relation is not present in G_{gm}, apparently the value $0,002$ is too small to be picked up by gMark. Furthermore the proportions for *HasKeyword* and *AuthoredBy* are significantly different than they were intended to be. The other relations are relatively close.

Edge	Count	Proportion	Actual Count	Actual Proportion
Interacts	60.060	0,002	0	0
EncodedOn	2.147.060	0,063	149.688	0,026
OccursIn	549.646	0,016	527.000	0,092
Reference	925.144	0,027	172.019	0,030
HasKeyword	7.842.186	0,230	3.518.923	0,613
AuthoredBy	21.701.347	0,637	1.167.801	0,203
PublishedIn	925.143	0,027	204.000	0,036

Table 5.4: For each edge the count in the original graph G_{up}, the proportion used in the schema definition, and the actual count and proportion found in the generated graph G_{gm}.

5.2.2 Analysis of distributions

Back in Section 5.1.3 we fitted statistical distributions to the edge degrees found in G_{up}, in this section we will gather statistics from G_{gm} and analyze how well the data matches the intended distributions. We will first discuss the out degree distributions and then move on to the in degrees.

For the out degrees we fitted five normal distributions and two uniform distributions. The latter are *OccursIn* and *PusblishedIn* both with min=1 and max=1, their out degree frequencies are displayed in Table 5.5. Recall that we can only check the id's in G_{gm} and not the node types, so we can not confirm that this distribution matches. All we know is that for these two relations there are no nodes with out degrees 2 or greater, but there might be proteins without an *OccursIn* relation to an organism, or articles not published in a journal. Judging by the ration between degrees 0 and 1 however, it seems quite probable that this not the case and the distributions are correct.

Degree	OccursIn	PublishedIn
0	354.809	677.809
1	527.000	204.000

Table 5.5: Frequency table of the out degree of *OccursIn* and *PublishedIn* in G_{gm}, both were classified as Uniform(1,1) in G_{up}.

The *Interacts* relation does not occur in G_{gm} so we are left with the four normal distributions in Table 5.6. When we look at the figures refered to in the rightmost column we can see that the actual distributions are very close to the target. Note that we left out degree=0 in these figures because this number would include all vertices that can not have the relation, and this would skew the data. Overall we can conclude that except for the missing *Interacts* relation, capturing the out degrees in a gMark schema definition has worked very well.

Relation	**Target parameters**	**Figure**
EncodedOn	$\mu = 1,95; \sigma = 0,99$	5.16
HasKeyword	$\mu = 7,13; \sigma = 3,61$	5.17
Reference	$\mu = 1,68; \sigma = 2,97$	5.18
AuthoredBy	$\mu = 5,87; \sigma = 4,99$	5.19

Table 5.6: For each of the normally distributed out degrees the estimated μ, σ in G_{up}, and the corresponding figure plotting the distribution in G_{gm}.

Now we can move on to the in degrees. Table 5.7 shows the Zipfian α of each relation for both G_{up} and G_{gm}. The only relation missing from this Table is *Interacts*, which had a Normal distribution in G_{up}. But since this relation is entirely absent from G_{gm} we only need to discuss the Zipfian in degrees. The table indicates that *AuthoredBy* is matched almost perfectly. The relations *EncodedOn*, *OccursIn*, *Reference*, and *PublishedIn* have some varying degrees of error, looking at the referenced figures however they are still quite reasonable. The most troublesome relation is *HasKeyword* for which a Zipfian distribution can not be fitted. This distribution was already difficult to match in the original graph G_{up}, which resulted in a very low α value.

Relation	**Target Alpha**	**Actual Alpha**	**Figure**
EncodedOn	3,31	2,41	5.20
OccursIn	1,62	1,52	5.21
Reference	2,53	1,85	5.22
HasKeyword	0,2	/	//
AuthoredBy	1,78	1,79	5.23
PublishedIn	1,37	1,79	5.24

Table 5.7: For each relation the estimated α for the Zipfian in-degree distribution in G_{up}, and the actual α found in G_{gm}.

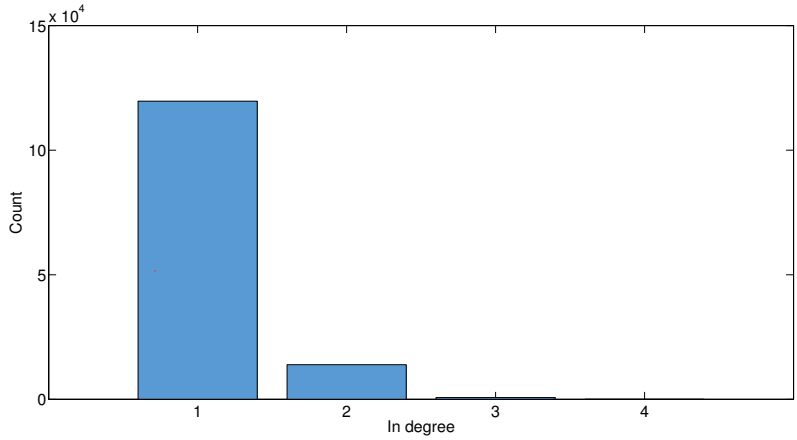

Figure 5.16: Histogram of the out degree of *EncodedOn* in G_{gm}, the specified distribution from G_{up} is Normal $\mu = 1,68$ and $\sigma = 2,97$.

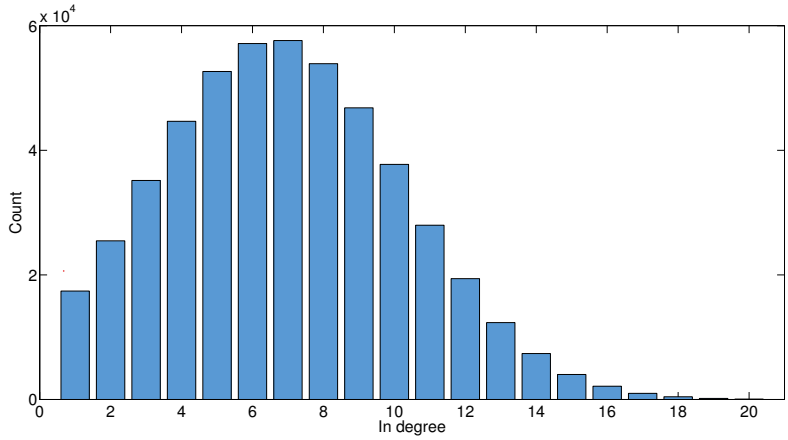

Figure 5.17: Histogram of the out degree of *HasKeyword* in G_{gm}, the specified distribution from G_{up} is Normal $\mu = 7,13$ and $\sigma = 3,61$.

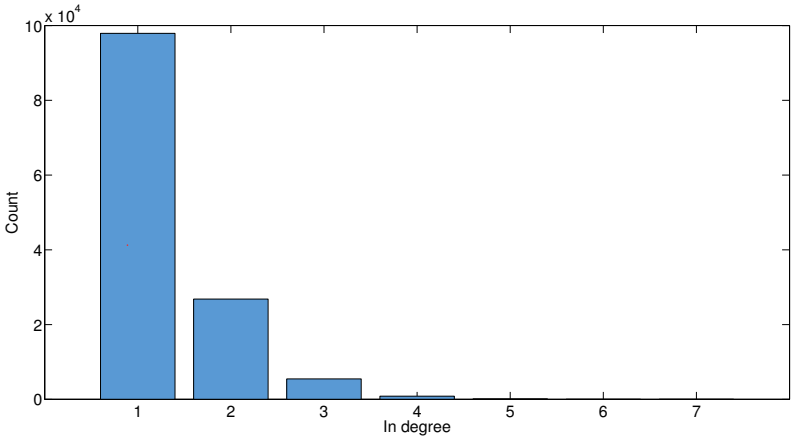

Figure 5.18: Histogram of the out degree of *Reference* in G_{gm}, the specified distribution from G_{up} is Normal $\mu = 1,68$ and $\sigma = 2,97$.

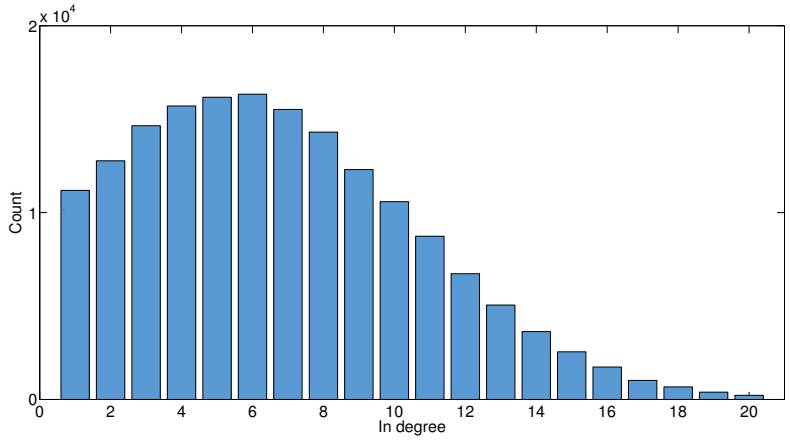

Figure 5.19: Histogram of the out degree of *AuthoredBy* in G_{gm}, the specified distribution from G_{up} is Normal $\mu = 5,87$ and $\sigma = 4,99$.

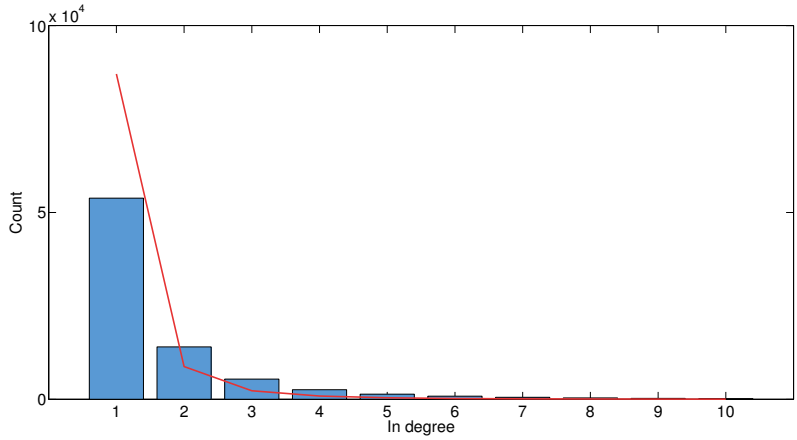

Figure 5.20: Histogram of the in degree of *EncodedOn* in G_{gm}, also plotted is the specified distribution from G_{up}, which is Zipfian $\alpha = 3,31$.

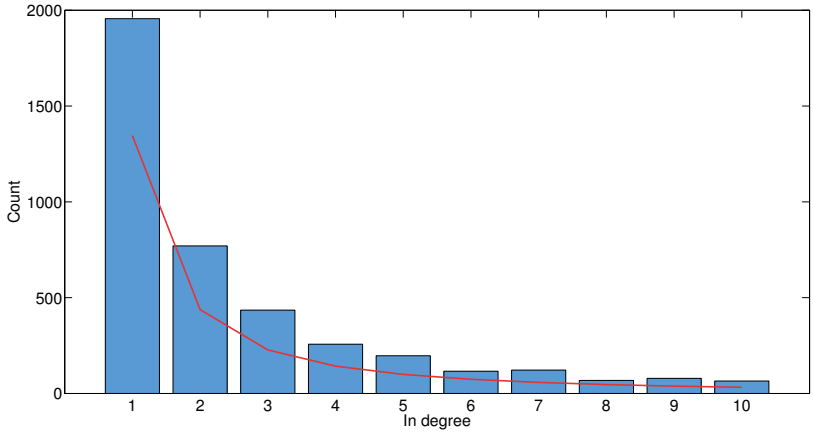

Figure 5.21: Histogram of the in degree of *OccursIn* in G_{gm}, also plotted is the specified distribution from G_{up}, which is Zipfian $\alpha = 1,62$.

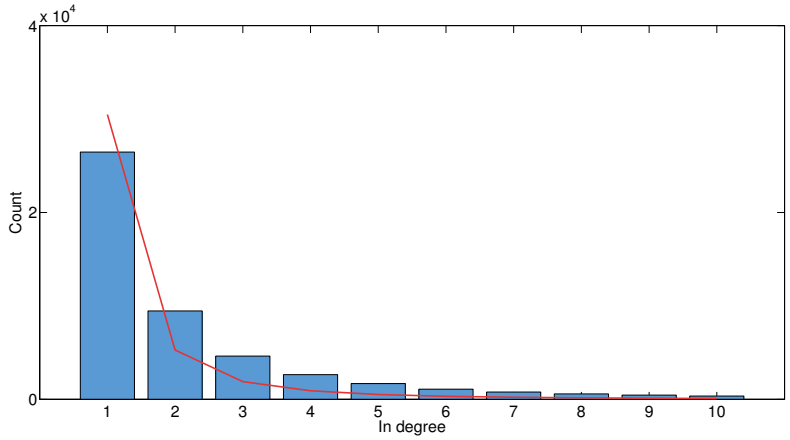

Figure 5.22: Histogram of the in degree of *Reference* in G_{gm}, also plotted is the specified distribution from G_{up}, which is Zipfian $\alpha = 2, 53$.

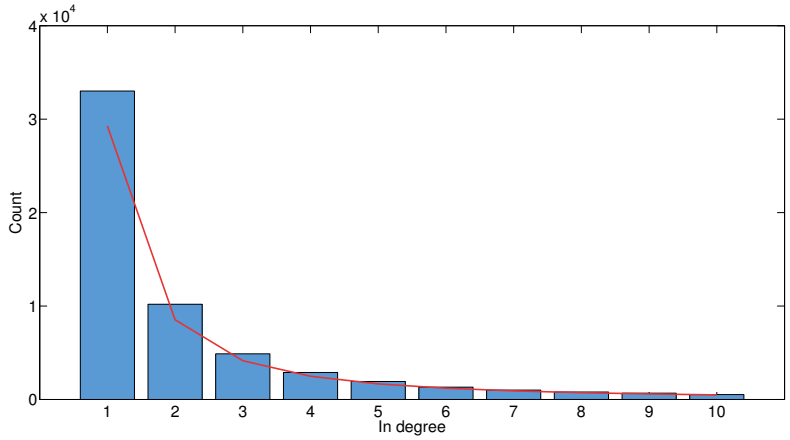

Figure 5.23: Histogram of the in degree of *AuthoredBy* in G_{gm}, also plotted is the specified distribution from G_{up}, which is Zipfian $\alpha = 1, 78$.

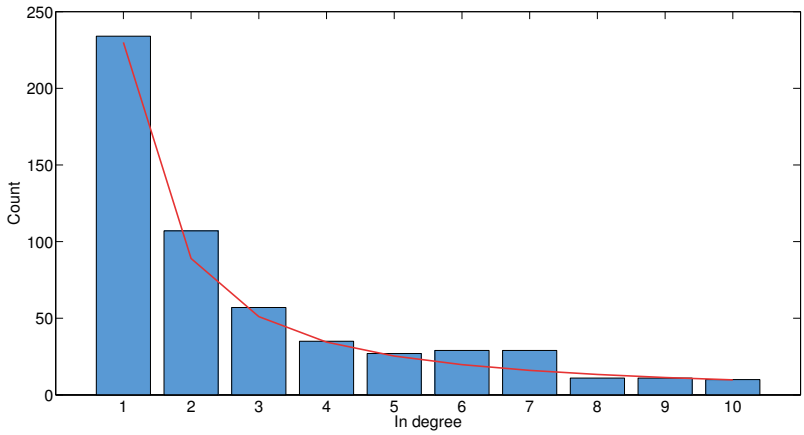

Figure 5.24: Histogram of the in degree of *PublishedIn* in G_{gm}, also plotted is the specified distribution from G_{up}, which is Zipfian $\alpha = 1,37$.

5.3 Summary

In this chapter we have generated synthetic data simulating the UniProtKB protein database. This process consisted of two parts, in the first part we performed some data mining on the real UniProt graph to discover the different node types and edges, their relative proportions, and the degree distributions of each edge type. To each distribution we fitted one of the three statistical distributions supported by gMark. The second part consisted of implementing the properties we discovered earlier in a gMark schema definition, generating a graph based on this schema, and finally analyzing this graph to see if it shares the selected properties with the original graph.

We conclude that the gMark schema definition is sufficient in capturing the most important properties of the data, and that the gMark data generator can effectively generate a graph satisfying the constraints defined in the schema.

Chapter 6

Conclusions

This chapter concludes the thesis, in Section 6.1 we summarize the work we have done and our final conclusions. Section 6.2 discusses the limitations we faced during the development of this work. Finally, section 6.3 proposes suggestions for future extensions of our research.

6.1 Contributions

The main contribution of this thesis is the benchmark study, in which we have compared performance of four database systems on the execution of regular path queries over graph data. A clear winner could not be called, however none of the graph databases performed better than the relational SQL database Postgres. This is a surprising result, we expected graph databases would perform better on a task they were designed to do. Therefore we conclude that developers of graph databases have a lot of work ahead of them in improving the performance of their systems.

Recursive queries were treated in a separate benchmark study. Here the winner is Datalog-System which was amongst the fastest systems and scaled the best w.r.t. graph size. Whereas Postgres performed very well on the non-recursive queries, it was by far the slowest on recursive queries.

Furthermore we have demonstrated that we can model a new schema definition based on the UniProtKB database, and use this to generate synthetic data that resembles characteristics of the original graph. Although we could not always find distributions with a tight fit and there was some degree of error in node proportions, we believe that the gMark schema definition sufficiently captures the essential data and we are confident that this procedure can be applied to other graph data sources as well.

6.2 Limitations

GraphSystem's openCypher language is less expressive than the other languages, and it has slightly different semantics for pattern matching. Therefore query results sets might differ, and comparing execution times is not entirely fair.

6.3 Future work

gMark itself is still under development, future versions [2] are suggested to include a wider variety of queries, constant vertices instead of only variables, and selectivity estimation on n-ary queries rather than only binary queries. Performing the benchmark study using a version of gMark that includes these features would make it more complete.

The benchmark study we performed in this work used predefined schemas supplied by gMark. We demonstrated that we can generate a graph using a schema definition we created ourselves

based on the UniProtKB database. It would be interesting to see if we could also generate queries for UniProt and then perform a benchmark based on this scenario.

Constructing a schema definition for the UniProt database required a lot of manual labor, first in gathering the required properties from the graph, second in implementing this in an XML file. Therefore our final suggestion for future work is to automate this process by writing software that does some of this work for us; mining the original graph for us and then constructing a schema definition based on this. We probably still need to do some manual selection and quality control, but this could significantly speed up the process and allow us to quickly generate benchmark data for many more graph domains.

Bibliography

[1] "http://www.postgresql.org/."

[2] G. Bagan, A. Bonifati, R. Ciucanu, G. Fletcher, A. Lemay, and N. Advokaat, "Controlling diversity in benchmarking graph databases," *arXiv:1511.08386*, November 2015.

[3] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, "Diversified stress testing of rdf data management systems," *ISWC*, pp. 197–212, 2014.

[4] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz., "The ldbc social network benchmark: Interactive workload," *SIGMOD*, pp. 619–630, 2015.

[5] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "Sp2bench: A sparql performance benchmark," *ICDE*, pp. 222–233, 2009.

[6] C. Bizer and A. Schultz, "The berlin sparql benchmark," *Int. J. Semantic Web Inf. Syst.*, pp. 1–24, 2009.

[7] Y. Guo, Z. Pan, and J. Heflin, "Lubm: A benchmark for owl knowledge base systems," *J. Web Sem.*, pp. 158–182, 2005.

[8] T. U. Consortium, "Uniprot: a hub for protein information," *Nucleic Acids Res.*, vol. 43, no. D204-D212, 2015.

[9] "http://www.opencypher.org/."

[10] "http://db-engines.com/en/ranking/graph+dbms."

[11] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database," *ACM SE '10 Proceedings of the 48th Annual Southeast Regional Conference*, no. 42, 2010.

[12] S. Jouili and V. Vansteenberghe, "An empirical comparison of graph databases," *Social Computing (SocialCom), 2013 International Conference on*, pp. 708–715, Sept 2013.

[13] R. Angles, "A comparison of current graph database modelsa," *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pp. 171 – 177, 2012.

[14] "http://www.postgresql.org/docs/8.4/static/queries-with.html."

Appendix A

Examples of Query translations

This appendix shows an example of a UCRPQ query translated in openCypher CustomDatalog SPARQL, and SQL. We consider the query:

$$(?x, ?z) \leftarrow (?x, a^{-}, ?y), (?y, (b + c)*, ?z)$$

openCypher. As explained in Section 2.3.3 the maximum path length under a star is one.

```
MATCH (y)-[:b|c*]->(z), (x)<-[:a]-(y) RETURN x.id,z.id;
```

CustomDatalog

```
gmarkSubquery00(x, y) <- core:edge:edge(x1, a, x0), x = x0, y = x1.
gmarkSubquery01(x, y) <- core:edge:edge(x0, b, x1), x = x0, y = x1.
gmarkSubquery01(x, y) <- core:edge:edge(x0, c, x1), x = x0, y = x1.
gmarkSubquery01(x, x) <- core:edge:edge(x, r, y).
gmarkSubquery01(y, y) <- core:edge:edge(x, r, y).
gmarkSubquery01(x, y) <- gmarkSubquery01(x, z), gmarkSubquery01(z, y).
query(x, z) <- gmarkSubquery00(x, y), gmarkSubquery01(y, z).
```

SPARQL

```
PREFIX : <http://example.org/gmark/>
SELECT DISTINCT ?x ?z
WHERE {
 ?x (^:a) ?y . ?y ((:b)|(:c))* ?z .
}
```

SQL

```
WITH RECURSIVE
c0(src, trg) AS
((
  SELECT s0.src, s0.trg
  FROM (SELECT trg as src, src as trg, label FROM edge) as s0
  WHERE s0.label = a
)),
c1(src, trg) AS
((
  SELECT edge.src, edge.src
  FROM edge UNION
  SELECT edge.trg, edge.trg
  FROM edge UNION
```

```
  SELECT s0.src, s0.trg
  FROM edge s0
  WHERE s0.label = b UNION
  SELECT s0.src, s0.trg
  FROM edge s0
  WHERE s0.label = c
)),
c2(src, trg) AS
(
 SELECT src, trg
 FROM c1 UNION SELECT head.src, tail.trg
 FROM c1 as head, c2 as tail
 WHERE head.trg = tail.src
)
SELECT DISTINCT c0.src, c1.trg
FROM c0, c1, c2
WHERE c1.src = c0.trg;
```

Appendix B

UniProt schema definition for gMark

```
<generator>
  <graph>
    <nodes>1700000</nodes>
    <output ntriples="1" />
  </graph>

  <predicates>
    <size>7</size>

    <alias symbol="0">Interacts</alias>
    <proportion symbol="0">0.002</proportion>

    <alias symbol="1">EncodedOn</alias>
    <proportion symbol="1">0.063</proportion>

    <alias symbol="2">OccursIn</alias>
    <proportion symbol="2">0.016</proportion>

    <alias symbol="3">Reference</alias>
    <proportion symbol="3">0.027</proportion>

    <alias symbol="4">HasKeyword</alias>
    <proportion symbol="3">0.230</proportion>

    <alias symbol="5">AuthoredBy</alias>
    <proportion symbol="3">0.637</proportion>

    <alias symbol="6">PublishedIn</alias>
    <proportion symbol="3">0.027</proportion>
  </predicates>

  <types>
    <size>7</size>
    <alias type="0">Protein</alias>
    <proportion type="0">0.31</proportion>

    <alias type="1">Gene</alias>
    <proportion type="1">0.36</proportion>

    <alias type="2">Organism</alias>
    <fixed type="2">15000</fixed>

    <alias type="3">Article</alias>
    <proportion type="3">0.12</proportion>
```

```
    <alias type="4">Keyword</alias>
    <fixed type="4">1200</fixed>

    <alias type="5">Author</alias>
    <proportion type="5">0.21</proportion>

    <alias type="6">Journal</alias>
    <fixed type="6">2000</fixed>
  </types>

  <schema>
    <source type="0"> <!-- Protein -->
      <target type="0" symbol="0" multiplicity="*"> <!-- Interacts -->
      <indistribution type="gaussian">
        <mu>0.00187</mu>
        <sigma>0.04321</sigma>
        </indistribution>
        <outdistribution type="gaussian">
        <mu>0.11</mu>
        <sigma>1.68</sigma>
        </outdistribution>
      </target>
      <target type="1" symbol="1" multiplicity="*"> <!-- EncodedOn -->
        <indistribution type="zipfian">
        <alpha>3.31</alpha>
        </indistribution>
        <outdistribution type="gaussian">
        <mu>1.95</mu>
        <sigma>0.99</sigma>
        </outdistribution>
      </target>
      <target type="2" symbol="2" multiplicity="*"> <!-- OccursIn -->
        <indistribution type="zipfian">
        <alpha>1.62</alpha>
        </indistribution>
        <outdistribution type="uniform">
        <min>1</min>
        <max>1</max>
        </outdistribution>
      </target>
      <target type="3" symbol="3" multiplicity="*"> <!-- Reference -->
        <indistribution type="zipfian">
        <alpha>2.53</alpha>
        </indistribution>
        <outdistribution type="gaussian">
        <mu>1.68</mu>
        <sigma>2.97</sigma>
        </outdistribution>
      </target>
      <target type="4" symbol="4" multiplicity="*"> <!-- HasKeyword -->
      <indistribution type="zipfian">
        <alpha>0.20</alpha>
        </indistribution>
        <outdistribution type="gaussian">
        <mu>7.13</mu>
        <sigma>3.61</sigma>
        </outdistribution>
      </target>
    </source>
    <source type="3"> <!-- Article -->
      <target type="5" symbol="5" multiplicity="*"> <!-- AuthoredBy -->
        <indistribution type="zipfian">
        <alpha>1.78</alpha>
        </indistribution>
        <outdistribution type="gaussian">
        <mu>5.87</mu>
        <sigma>4.99</sigma>
```

```
        </outdistribution>
      </target>
      <target type="6" symbol="6" multiplicity="*"> <!-- PublishedIn -->
        <indistribution type="zipfian">
        <alpha>1.37</alpha>
        </indistribution>
        <outdistribution type="uniform">
        <min>1</min>
        <max>1</max>
        </outdistribution>
      </target>
    </source>
  </schema>
</generator>
```