Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 164

Systems Group, Department of Computer Science, ETH Zurich

A High-level Graph Query Language Interface for Differential Dataflow

by

Lukas Striebel

Supervised by

Prof. Timothy Roscoe
Dr. Ioannis Liagouris
Dr. Desislava Demitrova
Moritz Hoffmann

October 2016 - April 2017

Informatik
Computer Science

Abstract

In today's business world, Social Networks have become one of the most important if not the most important platforms to advertise your product and gain new customers. Graph databases have been increasing in popularity in comparison to traditional relational databases. This tendency has given birth to the need of powerful and graph datawarehouses and query evaluators. Thus, we present cool program.

Contents

1 Introduction

1.1 Motivation

1.2 Structure of the Thesis

The Thesis has the following layout:
First, we give an overview of the technologies used in the Thesis.
We continue by giving a detailed explanation of the query parser and query evaluator. Next, we describe the experiments that were run in order to determine the performance of the program. Finally, we draw our conclusion and show potential future work.

2 Background

In this section, all the relevant technologies used in the Thesis are explained.

2.1 The Property Graph Query Language

The Property Graph Query Language (PGQL) was developed by Oskar van Rest at Oracle. [8] PGQL enables developers to write intuitive path queries over a property graph. A PGQL query consists of 4 clauses, two of them are optional.

- Path Clause (optional)
- Select Clause (required)
- Where Clause (required)
- Solution Modifier Clause (optional)

In the Path Clause custom path patterns are defined, which are to be used again in the Where Clause.
Example:

```
PATH connects_to := (:Generator) -[:has_connector]-> (:Connector WITH status =
    'OPERATIVE') <-[:has_connector]- (:Generator)
```

The Select Clause bears great similarity to the SQL one. Here, the set of properties one wishes to retrieve is defined. Possible Selections are either

- everything, indicated by the use of *
- certain attributes of edges and/or vertices
- Aggregation of certain attributes of edges and/or vertices.

Currently, there are five types of aggregations supported:

- COUNT, returns the number of tuples in the solution
- MAX, returns the maximum value of an attribute in any tuple. The specified attribute has to be numeric.
- MIN, returns the minimum value of an attribute in any tuple. The specified attribute has to be numeric.
- SUM, returns the sum of an attribute over all the tuples. The specified attribute has to be numeric.

- AVG, returns the average of an attribute over all the tuple. The specified attribute has to be numeric.

Example:

```
SELECT v.name, AVG(v.age)
```

The most complex part of the query is the Where Clause. In this section, all the requirements that the edges and vertices of the result set have to fulfill are specified. **Example:**

```
WHERE v.name = 'Alice', v.age > 30
```

2.2 Dataflow Computation

Dataflow oriented computation is a young field in Computer Science.

2.3 Differential Dataflow

Differential Dataflow was developed by Frank McSherry at Microsoft. [5] Rather than rewriting the entire data every time a change occurs, Differential Dataflow only keeps track of the changes made to the data. This allows for huge timesavings when executing the same operation multiple times on different data, since the results of previous computations can be reused.

2.4 The Rust Programming Language

Rust is a fairly new programming language. It's origin lies with the Mozilla Company, where former employee Graydon Hoare started development in 2006. In the 2009, Mozilla offically began sponsering th eproject.
Rust has been praised as it combines . The Rust compiler is self-hosted, meaning it is written in Rust as well. Rust won the Award for Most Loved Programming Language in 2016, host by the Stack Overflow Community. Rust is completly opensource, and a big part of the code was written by members of the community.

3 Parser

In the following, we describe how our parser is build. The Parser is build using nom, [1]. Nom was developed by Geoffroy Couprie. It is a byte oriented, zero copy, streaming Parser Library written in Rust. The library provides macros, functions and enums which faciliate the parsing process.

The most commonly used macros were:

- do_parse!: Takes a list of parsers as inputs and applies them sequentially, finally returns a tupel.
- alt_complete!: Takes a set of parsers as input and applies them until one succeeds. Returns the result of the first successful parser.
- opt!: Takes a parsers as input and makes it optional. Returns None if parser was unsuccessful or Some otherwise.
- tag!: Parses a specific String. Aborts if the String is not found.
- named!: Faciliates the creation of new custom parsers.
- many0!: Applies the parser 0 or more times.
- many1!: Applies the parser 1 or more times.

As mentioned in Section 2.1, the Parser has to recognize the following 4 clauses: Path definitions, select, where, solution modifier.
The main parse function therefore looks:

```
named!(pgql_query<Query>,
    do_parse!(
   paths: opt!(paths) >>
   space >>
   select: select_clause >>
        space >>
        vvhere: where_clause >>
   space >>
        solmod: opt!(solutionModifier) >>
        (Query { select: select, vvhere: vvhere, paths: paths, solmod: solmod})
```

3.1 Path Clause

The Path Clause is a list of definitions. Each Path definition starts with a name, followed by ':=' and then a path description. The path defined in this clause can then be reused in the Where Clause. One of many difficulties encountered while writing the parser, is the ability to differentiate between a single path definition and multiple ones, seperated by commas.

3.2 Select Clause

The Select Clause of PGQL is very similar to the SQL one. Started by the keyword ‘Select’, a list of attributes is provided. Attributes may be renamed with the keyword ‘as’. Aggregate functions like ‘Sum’, ‘Avg’ etc. may also be accessed.

3.3 Where Clause

The Where Clause consists of a list of Constraints. Each constraint defines either a path or value requirement that has to be fulfilled by the respective vertex or edge.

3.3.1 Path Constraints

Path Constraints are patterns, which are matched against the graph. If a set vertices does not fit the pattern, it is discarded.
A typical Path Constraint requires a vertex to have certain edges to other vertices. For example, the constraint: (v) -¿ (u) requires that the vertex v has a direct edge to the vertex v.

3.3.2 Value Constraints

Value Constraints are constraints on attributes of the Vertex, e.g. name = 'Alice' or age <40. Every Value Constraint has to include one or multiple vertex attributes, and one or more Literals. Literals are raw values, and come in 3 types:

- Strings e.g. 'Alice'
- Floats e.g. 40
- Booleans, which are either true or false

```
pub enum Literal {
    Str(String),
    Float(f32),
    Boolean(bool),
}

named!(literal<Literal>,
    alt_complete!(
        float      => { |f| Literal::Float(f)         } |
        boolean    => { |b| Literal::Boolean(b)       } |
        string     => { |s| Literal::Str(String::from(s)) }

    )
);
```

3.4 Solution Modifier Clause

The Solution Modifier Clause consists of three parts:

- GroupBy Clause
- OrderClause
- Offset and Limit Clause

All clauses are optional and can be omitted.

4 Query Evaluation

4.1 Graph Loader

The Loading of the graph plays a major part of the entire program execution time. To load a graph from a text file into differntial Dataflow, an entirely new parser had to be written. This parser was also built using nom. In order fo it to properly recognize a graph, the graph has to be supplied in the Vertex-Edge-List format. This format gives us a text file (.txt), which contains two lists. First, all the vertices and their attributes are described. The vertices are ordered ascing according to their ID. The IDs have to start at 1 and must be continuous.

4.2 Construction of the Dataflow

5 Benchmarking

All experiments were run on server. The machine contained 48 Amd cores. The installed memory was 128 GB of RAM.
In the experiment, four different topologies were used:
Large Fattree The large fattree topology is the largest one of the four topologies. It contains 55’000 vertices, of which 2’880 are switches. Furthermore the graph posses a total of 50’000 bidirectional edges. Since one bidirectional edge is represented by two directed edges, there are a total of 100’000 edges in the collection.

Small Fattree The large fattree topology is the largest one of the four topologies. It contains 55’000 vertices, of which 2’880 are switches. Furthermore the graph posses a total of 50’000 bidirectional edges. Since one bidirectional edge is represented by two directed edges, there are a total of 100’000 edges in the collection.

Large Jellyfish The large fattree topology is the largest one of the four topologies. It contains 55’000 vertices, of which 2’880 are switches. Furthermore the graph posses a total of 50’000 bidirectional edges. Since one bidirectional edge is represented by two directed edges, there are a total of 100’000 edges in the collection.

Small Jellyfish The large fattree topology is the largest one of the four topologies. It contains 55’000 vertices, of which 2’880 are switches. Furthermore the graph posses a total of 50’000 bidirectional edges. Since one bidirectional edge is represented by two directed edges, there are a total of 100’000 edges in the collection.

6 Results and Discussion

Things to consider

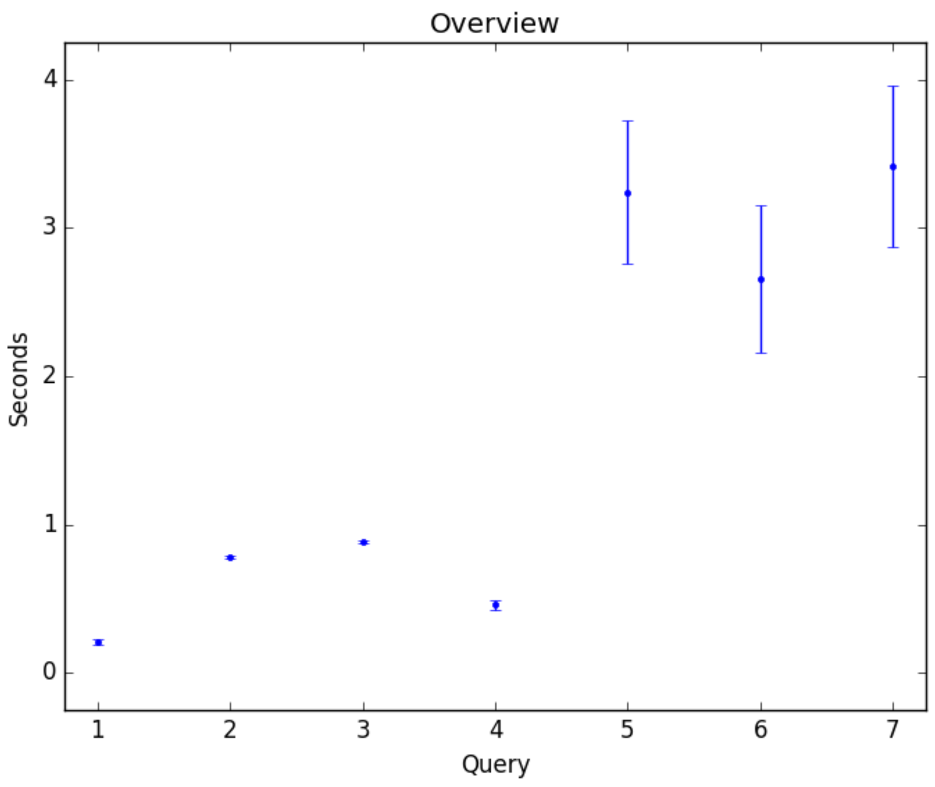
1. Weight is an attribute of the edges and (as far as I know) uniformly distributed from 1 to 9. The selection weight <2 will therefore remove around 90% of all edges., which speeds up the join process immensely.

2. Since Timely does not like HashMaps at all, attributes of vertices and edges are store in a Vector of (String, Literal) tuples. However, in order to do a selection, this Vector has to be transformed back into a HashMap, which takes some time. Therefore the more selection a query contains, the slower it should be.

3. One Edge in the query triggers two joins in the evaluation, because we have to join the edge collection twice with the vertex collection.

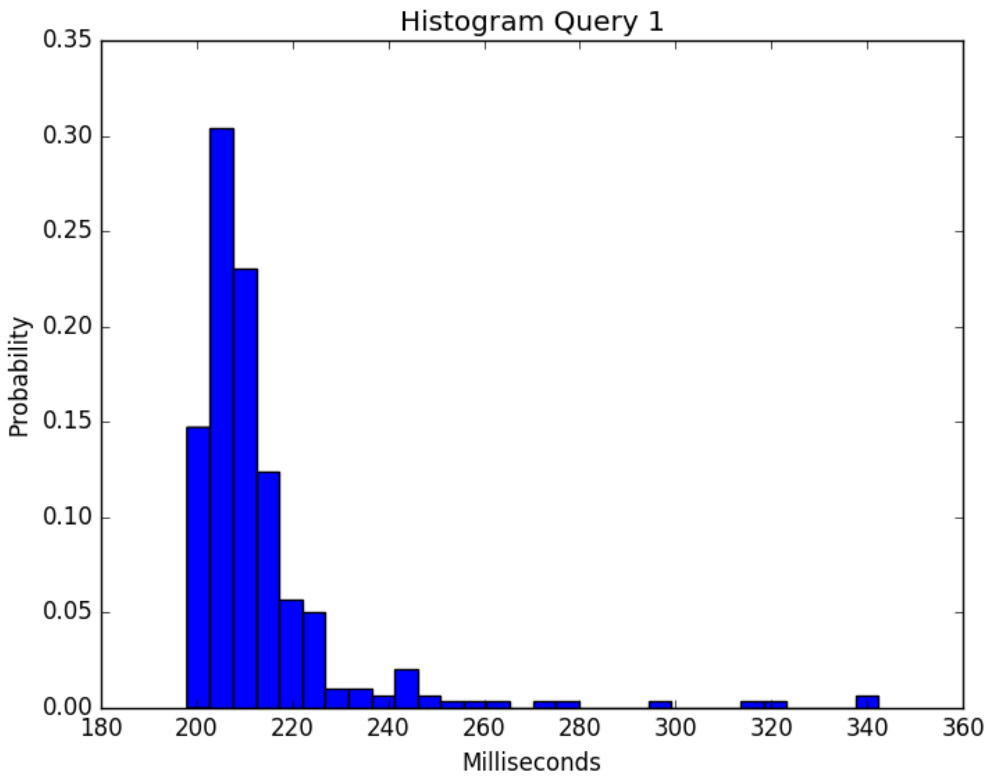
4. These are the results of the large fattree, with nearly 60'000 vertices and 50'000 bidirectional edge (ergo 100'000 in the collection). The results look very similar for the smaller fattree topology, just a bit smaller in magnitude.

Overview



Query 1

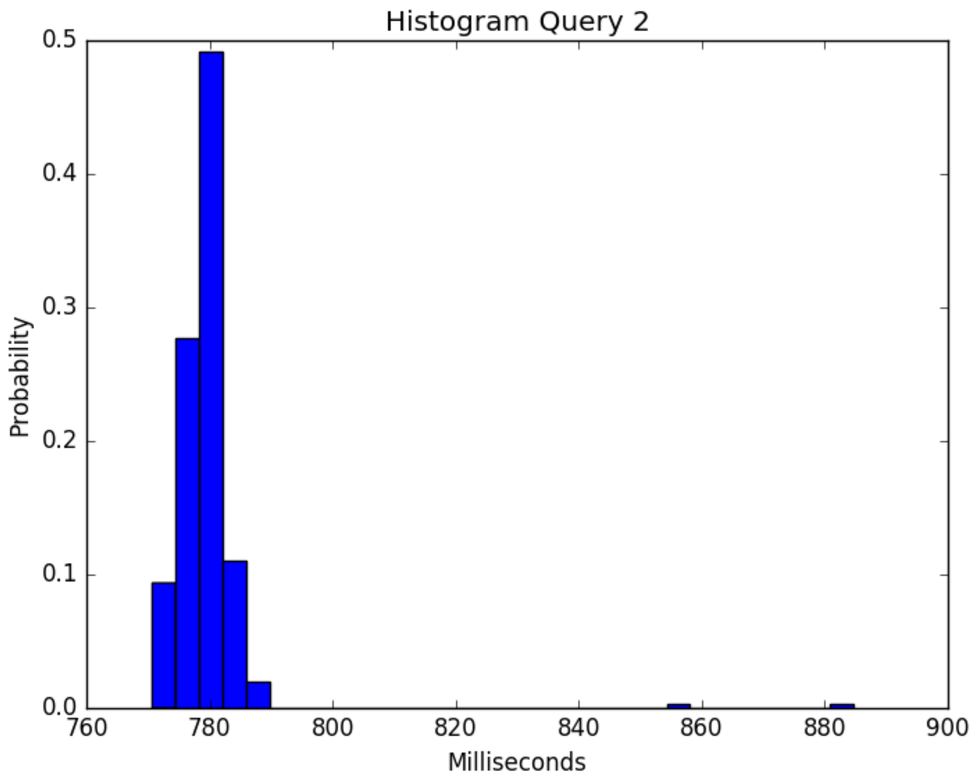
```
SELECT * WHERE u.label() = 'switch', u.position = 'access'
```



2 simple selections, 0 joins. One pass through the vertices collection is enough to produce the result. 200 milliseconds evaluation time.

Query2

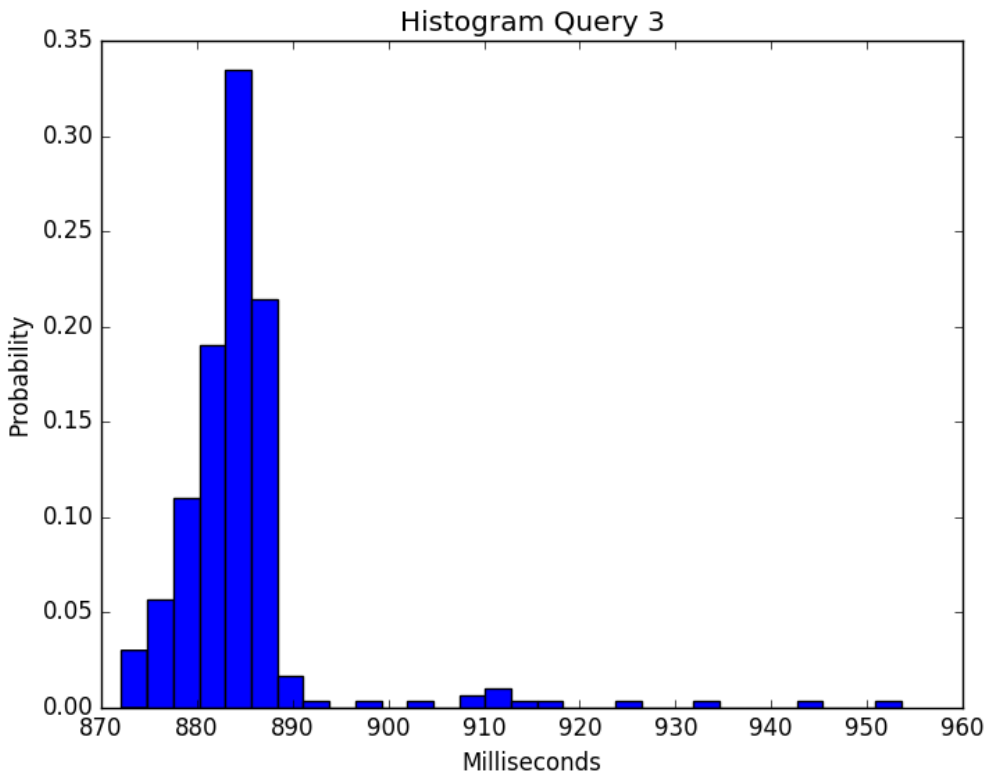
```
SELECT * WHERE (n) -[e with weight < 4]-> (m)
```



2 Joins and 1 Selection. The joins are not very expensive since we only use about 30% of the edges. 700 milliseconds evaluation time.

Query3

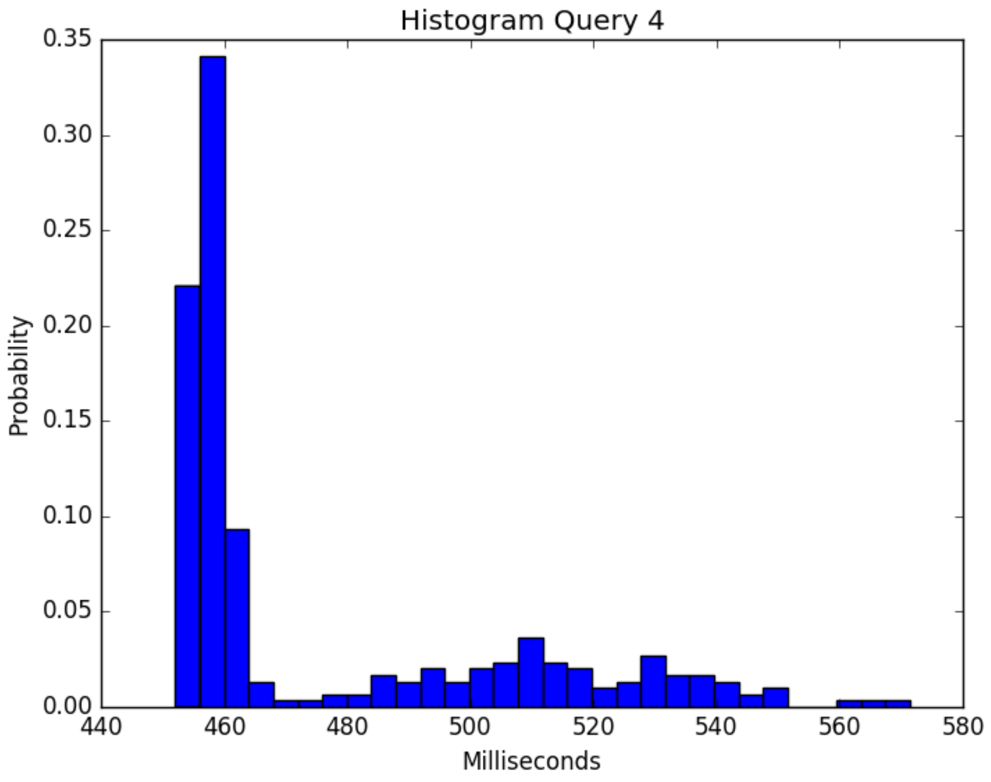
```
SELECT * WHERE (n:switch) -> (m with position = 'distribution')
```



2 Joins and 2 Selections. I expected this query to be a little bit slower since we are using all the edges in the joins, but it is only a tiny bit slower than Query 2.

Query4

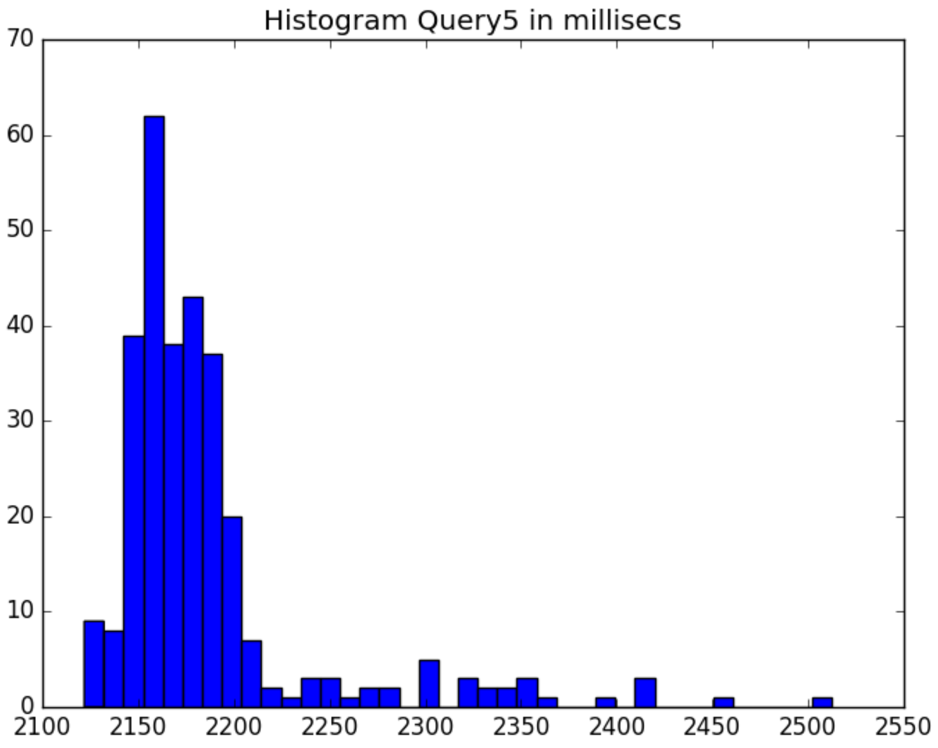
```
SELECT * WHERE (n with position = 'distribution')
-[e with weight > 8]-> (m with position = 'access')
```



2 Joins and 3 Selections. Even though this query has more constraints than Query 2 and 3, it is faster since we only use 10% of the edges in the joins.

Query5

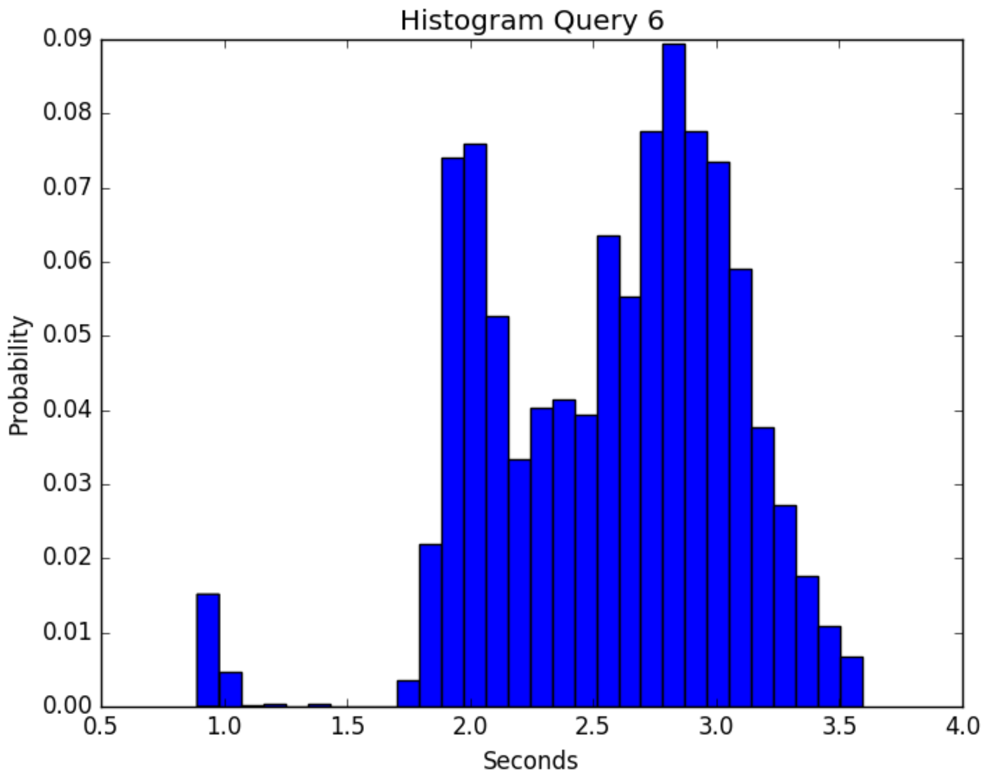
```
SELECT * WHERE (u WITH position = ’access’) -> (v WITH position = ’distribution’)
 -> (w WITH position = ’core’)
```



4 Joins and 3 Selections. The last 3 queries all contain multiple query edges and are much slower. Since there is no selection on the edge, this particular query is quite slow even though it has ”only” 4 joins.

Query6

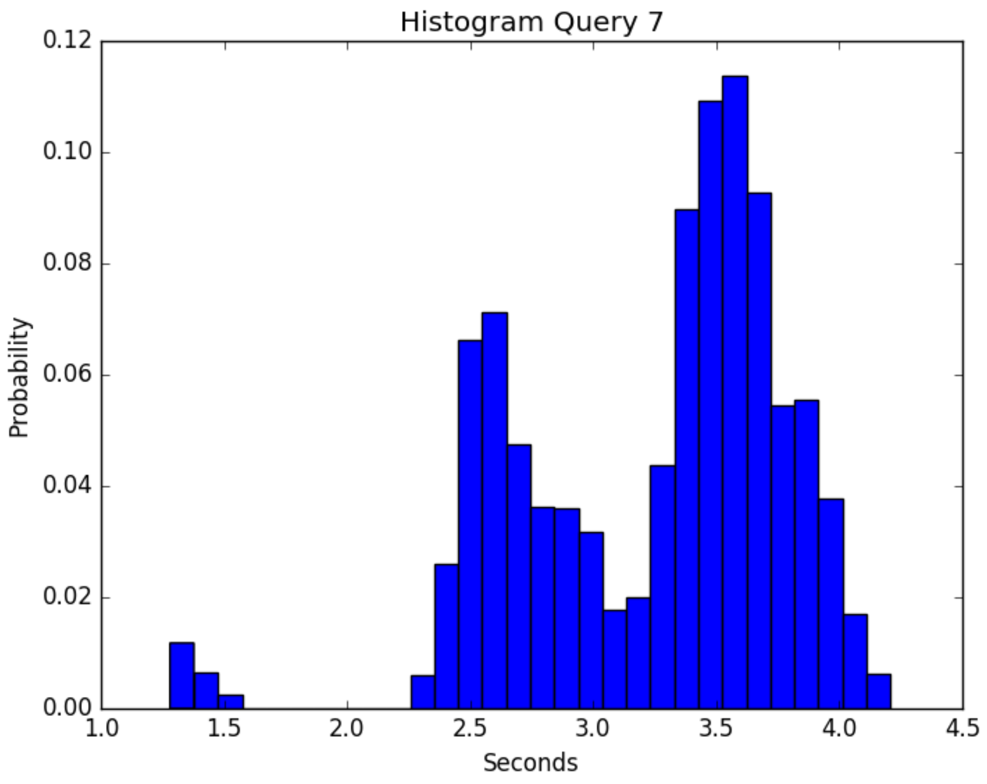
```
SELECT * WHERE (u WITH position = 'access')
 -[e with weight  < 2]-> (v WITH position = 'distribution')
 -[f with weight > 8]-> (w WITH position = 'core')
```



4 Joins and 5 Selections. Like query 4, the evaluation time goes down as we add more constraints on the edges. Since we have a lot less tuples in the 3rd and 4th join, this query is faster than previous one.

Query7

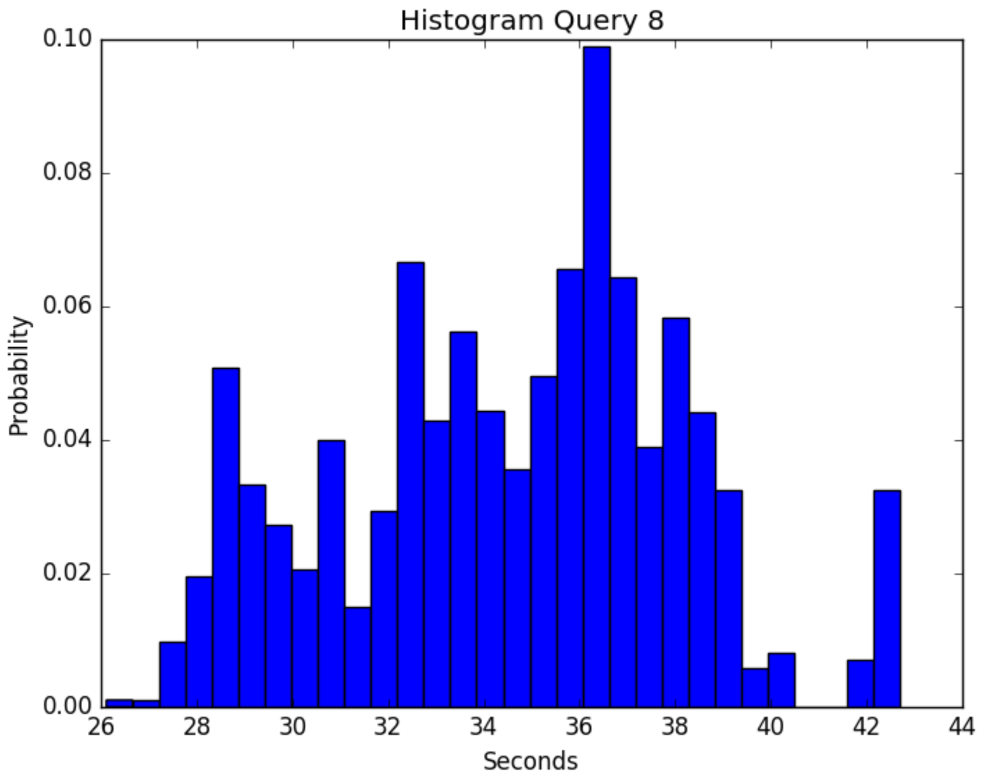
```
SELECT * WHERE (u WITH position = 'access')
 -[e with weight  < 2]-> (v WITH position = 'distribution')
 -[f with weight  < 2]-> (w WITH position = 'core')
 -[g with weight  < 2]-> (x WITH position = 'distribution')
```



6 Joins and 7 Selections. This is probably the most interesting result. We add two more joins to the query, but we restrict the number of edges used in all the joins dramatically. This consequently leads to a surprisingly low evaluation time, almost on the level as query 5. This demonstrates how big the impact of the joins is for the evaluation time.

Query 8

```
SELECT * WHERE (u WITH position = 'access') -> (v WITH position = 'distribution')
 -> (w WITH position = 'core') -> (x WITH position = 'distribution')
```



6 Joins and 4 Selections. I have not included this query in the figure because the evaluation times are around 30 to 40 seconds when run with 48 workers. When using only a single worker I have measured evaluation times over 10 minutes. This is due to the fact that we join 6 times with the full edge collection.

7 Related Work

In this chapter we give an overview of other existing graph databases and query evluators.

7.1 PQL

A program query language, PQL for short, is a source language-independent notation to specify program queries and program views. PQL is used as an interface to Static Program Analyzers (SPA), interactive tools that enhance program understanding by answering queries about programs. Queris on global program design as well as searches for detail code patterns are both possible in PQL. Program queries and patterns supported by other notations described in literature and those supported by commercial tools can be written simply and naturally in PQL.[3]

7.2 Green-Marl

Green-Marl is a domain-specific language (DSL) with high level language construct that allow developers to describe their graph analysis algorithms intuitively, but expose the data-level parallelism inherent in the algorithms. Green-Marl comes with its own compiler which translates high-level algorithmic description written in Green-Marl into an efficient C++ implementation by exploiting this exposed datalevel parallelism. Furthermore, the Green-Marl compiler applies a set of optimizations that take advantage of the high-level semantic knowledge encoded in the Green-Marl DSL. Most graph analysis algorithms can be written very intuitively with Green-Marl and experimental results show that the compiler-generated implementation out of such descriptions performs just as well as or better than highly-tuned handcoded implementations.[2]

7.3 Gremlin

Developed by the Apache Software Foundation, Gremlin is a query language as well as a graph traversal machine. The graph traversal machine Gremlin consists of three parts that continously interact with each other: first the graph, second the traversal and finally the set of traversers. The traversers move about the graph according to the instructions specified in the traversal, where the result of the computation is the ultimate locations of all halted traversers. A Gremlin machine can be executed over any supporting graph computing system such as an OLTP graph database and/or an OLAP graph processor. The language Gremlin is a functional language implemented in the user's native programming language. Gremlin supports both imperative and declarative querying. [6]

7.4 SQLGraph

SQLGraph is a Graph Store that combines existing relational optimizers with a novel schema, in an attempt to give better performance for property graph storage and retrieval than popular noSQL graph stores. The schema combines relational storage for adjacency information with JSON storage for vertex and edge attributes. This particular schema design has benefits compared to a purely relational or purely JSON solution. The query translation mechanism translates Gremlin queries with no side effects into SQL queries so that one can leverage relational query optimizers. [7]

7.5 GraphiQL

GRAPHiQL is an intuitive query language for graph analytics, which allows developers to reason in terms of nodes and edges rather than the tables and joins which are used in relational databases. GRAPHiQL provides key graph constructs such as looping, recursion, and neighborhood operations. At runtime, GRAPHiQL compiles graph programs into efficient SQL queries that can run on any relational database. [4]

8 Summary

8.1 Conclusion

8.2 Acknowledgements

I would like to thank my supervisors John Liagouris, Desislava Dimitrova and Moritz Hoffmann.

References

[1] G. Couprie. Nom, a byte oriented, streaming, zero copy, parser combinators library in rust. In *2015 IEEE Security and Privacy Workshops*, pages 142–148, May 2015.

[2] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: A dsl for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362, New York, NY, USA, 2012. ACM.

[3] Stan Jarzabek. *PQL: A language for specifying abstract program views*, pages 324–342. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[4] A. Jindal and S. Madden. Graphiql: A graph intuitive query language for relational databases. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 441–450, Oct 2014.

[5] Frank McSherry, Derek Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Proceedings of CIDR 2013*, January 2013.

[6] Marko A. Rodriguez. The gremlin graph traversal machine and language. *CoRR*, abs/1508.03843, 2015.

[7] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1887–1901, New York, NY, USA, 2015. ACM.

[8] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. Pgql: A property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, pages 7:1–7:6, New York, NY, USA, 2016. ACM.