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 Dimitrova
Moritz Hoffmann

October 2016 - April 2017

inf | 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 and attract new customers. Thus, graph databases have been increasing in popularity in comparison to traditional relational databases. This tendency has given birth to the need of powerful graph datawarehouses and query evaluators. In this thesis, we present Qlidaf, an Interface of the Property Graph Query Language for Diffrerential Dataflow.

Contents

List of Figures

1 Introduction

1.1 Motivation

In our modern digital age, the Internet is the largest centre of data and knownledge. The websites of large companies are accessed thousands of times a day. In order to cope with this huge amount of client requests, large sever farms were built. In these farms, hundreds of servers and switches are connected and request have to travel across multiple machines until they are answered. Thus, the need for a system that directs and balances the huge amount of request arises. The first step to deal with this challenge is to model this complicated network of computers as a property graph. In a second step, an algorithm determines how the request should best be directed in order to not overload any part of the network. This thesis focuses on the second step and proposes a program that can evaluate queries on a property graph, in order to determine weak links and find the optimal distribution of the work load.

1.2 Structure of the Thesis

The Thesis has the following structure:
First, we give an overview of the technologies used in the Thesis in Section 2.
We continue by giving a detailed explanation of the query parser in Chapter 3. In Section 4, we explain how the program sets up the dataflows according to the query. Next, we describe the experiments that were run in order to determine the performance of the program in Chapter 5. Finally, we draw our conclusion, provide an outlook and show potential for 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[**?**]. PGQL enables developers to write intuitive path queries over a property graph. The syntax of PGQL greatly resembles the one of SQL. 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 following, we will give a short overview of the four clauses. We will go into more detail in section 3, in which we explain precisely how each clause has to be parsed.

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

```
PATH connects_to := (:Generator) --> (:Connector WITH status = 'OPERATIVE')
```

In this example, we define connects_to as an edge between a generator and an operating connector.

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.

Example:

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

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.

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, w.name != 'Bob', v -> w
```

Finally, the result set can be modified in the **Solution Modifier Clause**. This Clause can be broken down into four subsections, which are all extremly similiar to their SQL counterpart:

Group by In the group by the developer specifies the attributes that shall be used for the aggregation in the select clause.

Order by A set of attributes that determine the order of the results, either ascending or descending.

Limit Provides the maximum cardinality of the result set.

Offset Denotes how many tuples of the result set shall be skipped.

The official PGQL Specifications can be accessed online at [**?**]

2.2 Dataflow Programming

Traditionally, a program is a list of instructions to be executed by the computer. However, there are certain programming paradigms that do not conform to this idea. Dataflow oriented programming is one of them. In dataflow programming paradigm, a program is modeled as a directed graph of data flowing between operations. Rather than explicitly defining the order of execution as most other programming paradigms do, dataflow programming specifies a network of nodes, between which the data is exchanged. In program execution, this means until a node's data buffer is non-empty, it remains idle.

Only after the node receives an input, it begins to execute the operations that are defined in the program. Once the computation is completed, the resulting data is forwarded to the next node. Each node can be viewed as a black box, since the programmer does not have to worry what happens inside. His responsibilities lie only in the task of connecting the nodes in the correct manner.

2.3 Differential Dataflow

Developed by Frank McSherry at Microsoft, Differential dataflow is a data-parallel programming framework designed to efficiently process large volumes of data and to quickly respond to arbitrary changes in input collections.[**?**]
Differential Dataflow is an extension of Timely Dataflow, which was first introduced by Murray. [**?**] 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.

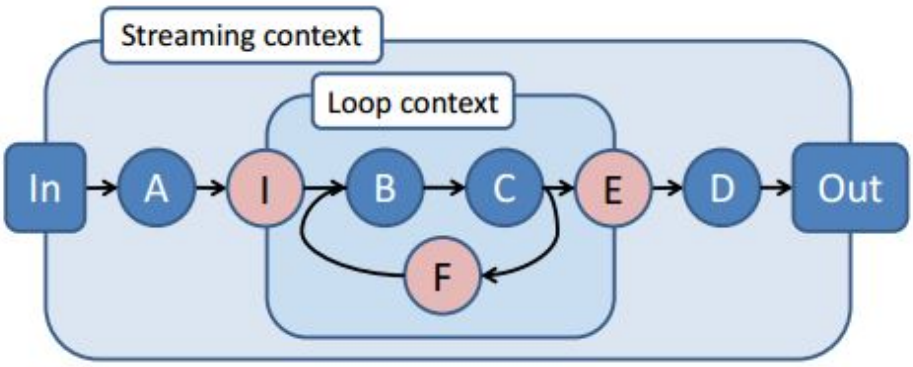


Figure 1: An example of a timely dataflow, Source:[**?**]

Differential Dataflow is open-source and available for download at [**?**].

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 year 2009, Mozilla offically began sponsering the project. It took another 6 years until the first stable version of the compiler was released. Nowadays most of the development is done by volunteers from the rust community. 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, hosted by the Stack Overflow Community.

Rust has been praised as it combines guaranteed memory safety with minimal runtime. Other prominent features are "zero-cost abstractions, move semantics, threads without data races, trait-based generics, pattern matching, type inference and efficient C bindings"[**?**].

One of Rust's most prominent feature is memory safety. That means it should never be possible to dereference a null pointer or a pointer to an object that has already been deallocated. However, it is still possible to produce out-of-bonds array accesses, since these checks have to be done at runtime.
In this thesis, we will often encounter recursive objects, such as trees for example. Since Rust does not have a null type, the children have to be wrapped in an object of type *Box*, in order to avoid infinite recursion.

```
1 struct BinaryTree {
2         leftChild: Box<BinaryTree>,
3         rightChild Box<BinaryTree>,
4 }
```

Rust is completly opensource, and a big part of the code was written by members of the community. It is currently licensed unter the Apache[1] and MIT license[2]

2.5 Nom

Our Query Parser is built using nom, [**?**]. 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. As of today, there exist dozends of parser written with nom, for example for gif, csv and tar files.

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.

[1] https://www.apache.org/licenses/LICENSE-2.0
[2] https://opensource.org/licenses/MIT

- many1!: Applies the parser 1 or more times.

Nom is open-source and available for download at [**?**].

3 Parser

In the following, we describe how our parser is built. As mentioned in Section 2.1, the Parser has to recognize the following 4 clauses: Path definitions (line 3), select (line 5), where (line 7), solution modifier (line 9). Finally, it returns a Query object (line 10) The main parse function therefore looks as follows:

```
1  named!(pgql_query<Query>,
2      do_parse!(
3          paths: opt!(path_clause) >>
4          space >>
5          select: select_clause >>
6          space >>
7          vvhere: where_clause >>
8          space >>
9          solmod: opt!(solution_modifier_clause) >>
10         (Query { select: select, vvhere: vvhere, paths: paths, solmod: solmod})
11     )
12 );
```

3.1 Path Clause

The Path Clause is a list of definitions. Each Path definition starts with a name (line 16), followed by ':=' (line 18)and then a path description (line 20). The definition is closed by the begining of a new line (line 6). The paths defined in this clause can then be reused in the Where Clause.

```
named!(path_clause<Vec<(String, Connection)> >,
    many0!(
        chain!(
            path: path ~
            opt!(space) ~
            tag!("\n"),
            || path
        )
    )
);

named!(path<(String, Connection)>,
    chain!(
        tag_no_case_s!("path") ~
        space ~
        name: char_only ~
        opt!(space) ~
        tag!(":=") ~
        opt!(space) ~
        pattern: path_pattern,
        || (String::from(name),pattern)
    )
);
```

3.2 Select Clause

The Select Clause of PGQL is very similar to the SQL one. Started by the keyword 'Select' (line 3), a list of attributes is provided (line 5). If the user wishes to retrieve everything, he is to use the asterix sign (line 18). Aggregate functions like 'Sum', 'Avg' etc. may also be accessed (line 12). Attributes may be renamed with the keyword 'as', but this function is currently not supported.

```
named!(select_clause<Vec<SelectElem> >,
    chain!(
        tag_no_case_s!("select") ~
        space ~
        select_elems: many1!(select_elem),
        || select_elems
    )
);

named!(select_elem<SelectElem>,
        alt_complete!(
                aggregate => {|aggregation| SelectElem::Aggregation(aggregation)} |
                variable_name => {
                        |v| {let (name, field) = v;
                        let a = Attribute{name: name, field: field};
                        SelectElem::Attribute(a)}
                } |
                tag_s!("*") => {|_| SelectElem::Star}
        )
);
```

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 u.

Vertices are absoultly essential for the path constraints. A Vertex is delimited by brackets () (line 3 & 29) and its properties are specified inside the brackets (or at a later point in time as a seperate constraint). There are three possible properties, all of them are completly optional:

- The name of the vertex. If no name is provided, the vertex is anonymous (line 5).
- A set label constraints, recognized by the : sign (line 7). Label constraints differ from value constraints since the labels are defined in a very particular way, unlike the attributes. See section 4.1 for more details.

- A set of inlined value constraints, recognized by the keyword WITH (line 9). This is basically just syntatic sugar, for example, the two formulation (v WITH age >10) and (v), v.age >10 are completly equal.

Code sample

```
1  named!(query_vertex<Vertex>,
2      delimited!(
3          char!('('),
4          do_parse!(
5              vertex_name: opt!(char_only) >>
6              opt!(space) >>
7              label: opt!(label_constraint) >>
8              opt!(space) >>
9              inlined: opt!(inlined_constraints) >>
10             ({
11                 let mut constraints: Vec<Expr> = match inlined {
12                     Some(value) => value,
13                     None => Vec::new()
14                 };
15
16                 match label {
17                     Some(value) => {constraints.push(value);},
18                     None => {}
19                 }
20
21                 match vertex_name {
22                     Some(name) => {Vertex{name: String::from(name),
23                                   anonymous: false, constraints: constraints }},
24                     None => Vertex{name: String::from(""),
25                                   anonymous: true, constraints: constraints }
26                 }
27             })
28         ),
29         char!(')')
30   )
31 );
```

The connection between two vertices can be a very simple or a very sophisticated edge with many attributes. An edge is defined by an arrow sign ->and its properties are specified inside the brackets. If there are no properties to be defined, the brackets are omitted (line 3 & 5). There are three parameteres, all of them are completly optional:

- The name of the edge. If no name is provided, the edge is anonymous (line 10).
- One label constraint, recognized by the : sign (line 12).

- A set of inlined value constraints, recognized by the keyword WITH (line 14).

Code sample

```
1  named!(query_edge<Edge>,
2         alt_complete!(
3             tag!("-->") => { |_| Edge {name: String::from(""),
4                                        inverted: false, constraints: Vec::new() } } |
5             tag!("->") => { |_| Edge {name: String::from(""),
6                                        inverted: false, constraints: Vec::new() } } |
7             delimited!(
8                 tag!("-["),
9                 do_parse!(
10                    edge_name: opt!(char_only) >>
11                    opt!(space) >>
12                    label: opt!(label_constraint) >>
13                    opt!(space) >>
14                    inlined: opt!(inlined_constraints) >>
15                    ({
16
17                        let mut constraints: Vec<Expr> = match inlined {
18                            Some(value) => value,
19                            None => Vec::new()
20                        };
21
22                        match label {
23                            Some(value) => {constraints.push(value);},
24                            None => {}
25                        }
26
27                        match edge_name {
28                            Some(name) => {Edge{name: String::from(name),
29                                         inverted: false, constraints: constraints }},
30                            None => Edge{name: String::from(""),
31                                        inverted: false, constraints: constraints }
32                        }
33                    })
34                ),
35                tag!("]->")
36 );
```

The PGQ Language allows to nest an unlimited amount of (vertex,edge,vertex)-tuples in one single constraint. I.e. after parsing an initial vertex, infinitely many vertices proceeded by an edge may follow. To process this chain of tuples, a fold function (line 13) has to be used. While iterating through this vector, we clone the previous vertex,

then create an new connection and push it into another vector(line 21). This vector is then returned at the end of the function (line 24).

```
1  named!(path_pattern<Vec<Connection> >,
2      chain!(
3          initial: query_vertex ~
4          remainder: many1!(
5              chain!(
6                  space ~
7                  edge: query_edge ~
8                  space ~
9                  target: query_vertex,
10                 || (edge, target)
11             )
12         ), ||
13     fold_connection(initial, remainder)
14     )
15 );
16
17 fn fold_connection(initial: Vertex, remainder: Vec<(Edge, Vertex)>) -> Vec<Connection> {
18     let mut result = Vec::new();
19     let mut previous = initial;
20     for (edge, vertex) in remainder{
21         result.push(Connection{source: previous, target: vertex.clone(), edge: edge});
22         previous = vertex;
23     }
24     return result;
25 }
```

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' (line 2)
- Floats e.g. 40 (line 3)
- Booleans, which are either true or false (line 4)

```
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)) }

    )
);
```

Possible operateros are:

- arithmetric: addition, subtraction, multiplication, division, modulo
- comperative: equals, not equals, greater, greater equals, smaller, smaller equals
- boolean: and, or, not
- string: like. The like is denoted as = and tests if a string is contained within another string. For example 'lice' = 'Alice' returns true.

Expressions play a vital part in the parsing of a query. For simplicity, only the addition and subtraction expression are shown in the following example.

```
named!(expression< Expr >, chain!(
    initial: operand ~
    remainder: many0!(
            alt!(
                chain!(opt!(space) ~ tag!("+")  ~ opt!(space) ~ op: operand,   || (Oper::Add, op))       |
                chain!(opt!(space) ~ tag!("-")  ~ opt!(space) ~ op: operand,   || (Oper::Sub, op))
             )
        ), ||
    fold_exprs(initial, remainder)
    )
);

fn fold_exprs(initial: Expr, remainder: Vec<(Oper, Expr)>) -> Expr {
    remainder.into_iter().fold(initial, |acc, pair| {
        let (oper, expr) = pair;
        match oper {
            Oper::Add       => Expr::Add(Box::new(acc), Box::new(expr)),
         Oper::Sub       => Expr::Sub(Box::new(acc), Box::new(expr)),
        }
    })
}
```

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.

```
named!(solution_modifier<(GroupBy, OrderBy, (i32, i32))>,
    chain!(
        group: group_by ~
        space ~
        order: order_by ~
        space ~
        limit: limit_offset,
        || (group, order, limit)
    )
);
```

4 Query Evaluation

In this chapter we will discuss how we construct the dataflow from our queries.

Figure 2: An overview of 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 differential 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.

Small sample graph

```
1 * { 'Server' }          name:'center'  ip:'192.168.0.0' ram:4
2 * { 'VM' }              name:'node1'   ip:'192.168.0.1' ram:8
3 * { 'Firewall' 'VM' }   name:'node2'   ip:'192.168.0.2' ram:16.5
4 * { 'Host' 'Server' 'VM' } name:'node3' ip:'192.168.0.3' ram:32.5
5 * { 'Switch' }          name:'node4'   ip:'192.168.0.4' ram:64.5
1 2 'connectss' bandwidth:1.5 utilization:0.14
2 3 'connects' bandwidth:2.1 utilization:0.22
2 4 'connects' bandwidth:1.1 utilization:0.38
4 5 'connects' bandwidth:1.3 utilization:0.58
```

The graph parser looks as folllows:

```
named!(graph_parser<Graph>,
    chain!(
        nodes: many1!(node) ~
        edges: many1!(edge),
        || Graph {nodes: nodes, edges: edges}
    )
);
```

Each vertex begins with an integer, representing the ID of the vertex, follow by an asterix. Subsequently the lables of the vertex are presented, delimited by curly brackets. At least one label has to be provided whereas an upper limit does not exist. Finally, all the attributes of the vertex are presented in a JSON-like fashion.

```
named!(node<Node>,
    chain!(
        id: unsigned_int ~
        space ~
        char!('*') ~
        space ~
        labels: labels ~
        space ~
        attr: attributes ~
        line_end,
        || {
            let mut map = HashMap::with_capacity(attr.len());
            for elem in attr{
                let (name, value) = elem;
                map.insert(String::from(name), value);
            }
            Node {id: id, label: labels, attribute_values: map }
        }
    )

);
```

```
named!(labels<Vec <String> >,
    chain!(
        char!('{') ~
        opt!(space) ~
        labels: many1!(
            chain!(
                s: string ~
                opt!(space),
                || String::from(s)
            ) ~
        char!('}'),
        || labels
    )
);
```

```
named!(attributes< Vec< (&str, Literal) > >,
    many0!(
        chain!(
            name: char_only ~
            opt!(space) ~
            char!(':') ~
            opt!(space) ~
            value: literal ~
            opt!(space),
            || (name, value)
        )
    )
);
```

The edges of the graph are parsed in a very similiar way. There are two main differences: instad of an asterix, a second integer is given denoting the ID of the target vertex, while the first integer determines the ID of the source vertex. Additionally, edges must have exactly one single label. The rest remains unchanged in comparison to the vertex parser.

```
named!(edge<GraphEdge>,
    chain!(
        source: unsigned_int ~
        space ~
        target: unsigned_int ~
        space ~
        label: string ~
        space ~
        attr: attributes ~
        line_end,
        || {
            let mut map = HashMap::with_capacity(attr.len());
            for elem in attr{
                let (name, value) = elem;
                map.insert(String::from(name), value);
            }
            GraphEdge {source: source, target: target,
                label:String::from(label), attribute_values: map }
        }
    )
);
```

4.2 Construction of the Dataflow

The biggest challenge in the thesis was translating the PGQL query into a dataflow. As described in chapter 3, a query contains a vector of projections and a vector of constraints. We chose to apply the projections at the vey latest point in time. Thus, our evaluation begins with the second vector.

4.3 Constraint sorting

First, we iterate through the vector and sort the constraints by type:

- If the constraint is a path pattern, it is put into a seperate vector.
- If the constraint is an expression, we determine the name of the involved vertex or edge by recursively exploring the expression. Once the variable name is known, the expression is then pushed into a Hashmap of Vectors. For each variable there exists a seperate vector with expressions. This grouping helps tremendously when filtering the vertex set later on.

Code sample

```
for constraint in &vvhere{
                match constraint {
                    &Constraint::PathPattern(ref pattern) => connections.push(pattern),
                    &Constraint::Expr(ref expr) => {
                        let name = explore_expr((*expr).clone());
                        let mut new = false;
                        match selections.get_mut(&name) {
                            Some(vec) => vec.push((*expr).clone()),
                            None => new = true,
                        }
                        if new {
                            selections.insert(name, vec![(*expr).clone()]);
                        }
                    },
                }
            }
```

4.4 Constructing an execution plan

Once we have iterated over the entire vector, we begin the construction of an execution plan. We iterate through all the path patterns (we have stored them previously in

a seperated vector) and make a decision how to construct the plan. There are three possibilities:

- Both the source and the target of the edge have not appeared before. In this case, we pull the source and the target set and join them with the edge set.
- The source of the edge has already appeared before. In this case, we pull the target set, map the result we carried onto the correct id and then join them with the edge set.
- Both the source and the target of the edge have appeared before. In this case, we are dealing with a loop in the query. Thus, we do not need to join any sets, but we can just enforce an additional constraint on the ids.

Code sample

```
for connection in connections {
                if !used_fields.contains(&connection.source.name) {
                    used_fields.insert(&connection.source.name);
                    used_fields.insert(&connection.target.name);
                    names.insert(&connection.source.name, ids);
                    ids = ids + 1;
                    names.insert(&connection.target.name, ids);
                    ids = ids + 1;

                    let mut new = false;
                    match selections.get_mut(&connection.source.name) {
                            Some(mut vec) => vec.append(& mut connection.source.constraints.clone()),
                            None => new = true,
                    }
                    if new {
                            selections.insert(connection.source.name.clone(), connection.source.constraints
                    }
                    new = false;
                    match selections.get_mut(&connection.target.name) {
                            Some(mut vec) => vec.append(& mut connection.target.constraints.clone()),
                            None => new = true,
                    }
                    if new {
                            selections.insert(connection.target.name.clone(), connection.target.constraints
                    }

                    execution_plan.push(
                        PhyPlan{
                            name: vec![connection.source.name.clone(), connection.target.name.clone()],
                            left: connection.source.name.clone(),
                            right: connection.target.name.clone(),
                            join_id: 0,
                            join: true,
                            filter_id: 100,
                            constraints: connection.edge.constraints.clone(),
                        }
                    );
                }
                else if !used_fields.contains(&connection.target.name) {
                    used_fields.insert(&connection.target.name);
                    names.insert(&connection.target.name, ids);
                    ids = ids + 1;

                    let mut new = false;
                    match selections.get_mut(&connection.target.name) {
                            Some(mut vec) => vec.append(& mut connection.target.constraints.clone()),
                            None => new = true,
                    }
                    if new {
                            selections.insert(connection.target.name.clone(), connection.target.constraints
                    }
                    execution_plan.push(
                        PhyPlan{
                            name: vec![connection.source.name.clone(), connection.target.name.clone()],
                            left: connection.source.name.clone(),
                            right: connection.target.name.clone(),
                            join_id: *names.get(&connection.source.name).unwrap(),
```

4.5 Selection on the vertex set

The next step is to apply all the selections we have sorted beforehand on the vertex set. For each non-anonymous vertex, we filter the entire set and save the result in a Hashmap. The key in this Hashmap is the name of the vertex. **Code sample**

```
let mut plans = HashMap::new();
for (name, filter) in selections {
    let result = vertices.filter(move |x| {
        check_node(&x, &filter)
    });
    plans.insert(name, result);
}
```

4.6 Executing the plan

After constructing a plan, we iterate over through all the steps we assemble in the chapter 4.4. In the beginng, our result, which is a Collection of Vectors of Vertices, is empty. We begin by pulling the source's vertex collection from the Hasmap we defined in section 4.5. The Hashmap's key is the name of the vertex. Once we have the correct vertex collections, we join it with the edge set, and map the result, in order to keep track of the vertex ID of the target. We then repeat this step for the target vertex. We end up with a vector of vertices, which we store as the result.

If there are more elements in the execution plan, we repeat the previous steps until we are done. The only difference is that we reuse our previous result, instead of pulling a new collection from our vertex Hashmap. **Code sample**

```
let mut result = None;
for step in execution_plan {
    if step.join && step.join_id == 0 {
        let sources = match plans.get(&step.left){
            None => vertices,
            Some(list) => list,
        };
        let targets = match plans.get(&step.right){
            Some(list) => list,
            None => vertices,
        };
        result =    Some(sources.map(|x| (x.id, x))
                    .join(&edges.filter(move |x| check_edge(&x, &step.constraints))
                    .map(|x| (x.source,x.target)))
                    .map(|(_,v1,v2)| (v2,v1))
                    .join(&targets.map(|x| (x.id, x)))
                    .map(|(_,v1,v2)| vec![v1,v2]));
    }
    else if !step.join {
        let int = step.join_id;
        let int2 = step.filter_id;
        result = Some(result.unwrap().map(move |x| (x[int].id, x))
                .join(&edges.filter(move |x| check_edge(&x, &step.constraints))
                .map(|x| (x.source,x.target)))
                .filter(move |x| {let &(ref key, ref vec, ref id) = x; vec[int2].id == *id})
                .map(|(_,v1,_)| v1));
    }
    else {
        let targets = match plans.get(&step.right){
            Some(list) => list,
            None => vertices,
        };
        result = Some(result.unwrap().map(move |vec| (vec[step.join_id].id, vec))
                .join(&edges.filter(move |x| check_edge(&x, &step.constraints))
                .map(|x| (x.source,x.target)))
                .map(|(_,v1,v2)| (v2,v1))
                .join(&targets.map(|x| (x.id, x)))
                .map(|(_, mut v1,v2)| {v1.push(v2);v1}));
    }
}
```

4.7 Applying the projection and aggregation

The final step is to apply the projections and the aggregation that were passed in a vector. In order to do get the desired result, we iterate over this vector and take the appropriate action depending what attribute is asked for by the user. In case of an aggregation, we first group the records, and then count, sum etc. the values. **Code sample**

```
for projection in &select{
    match projection {
        &SelectElem::Star => {
            list.inspect(|&(ref x,_)| println!("{:?}", x));
        },
        &SelectElem::Attribute(ref attr) => {
            let field = string_to_static_str(attr.field.clone());
            let id = *(names.get(&attr.name).unwrap());
            list.inspect(move |&(ref x,_)| println!("{:?}", x[id].get(&field.into()).unwrap()));
        },
        &SelectElem::Aggregation(ref aggr) => {
            match aggr {
                &Aggregation::Count(ref attr) => {
                        list.map(move |x| (1, x)).group(move |key, vals, output| {
                                let mut count = 0;
                                for _ in vals {
                                    count = count + 1;
                                }
                                output.push(((),count));
                            }).inspect(|&(_, ref x)| println!("{:?}", x));
                }
            }
        }
    }
}
```

4.8 Evaluating an expression

It is of great importance to evaluate the expressions formulated in the query in the dataflow.

```
fn evaluate_expr (constraint: Expr, node: &Node) -> Literal {
    match constraint{
        Expr::Equal(left, right)         => Literal::Boolean(evaluate_expr(*left, node) == evaluate_expr(*r
        Expr::NotEqual(left, right)      => Literal::Boolean(evaluate_expr(*left, node) != evaluate_expr(*r
        Expr::Smaller(left, right)       => evaluate_expr(*left, node).smaller(evaluate_expr(*right, node))
        Expr::SmallerEq(left, right)     => evaluate_expr(*left, node).smaller_eq(evaluate_expr(*right, nod
        Expr::Greater(left, right)       => evaluate_expr(*left, node).greater(evaluate_expr(*right, node))
        Expr::GreaterEq(left, right)     => evaluate_expr(*left, node).greater_eq(evaluate_expr(*right, nod
        Expr::Like(left, right)          => evaluate_expr(*left, node).contains(evaluate_expr(*right, node)
        Expr::And(left, right)           => evaluate_expr(*left, node).and(evaluate_expr(*right, node)),
        Expr::Or(left, right)            => evaluate_expr(*left, node).or(evaluate_expr(*right, node)),
        Expr::Not(value)                 => evaluate_expr(*value, node).not(),
        Expr::Label(label)               => Literal::Boolean(node.label.contains(&label)),
        Expr::Add(left, right)           => evaluate_expr(*left, node).add(evaluate_expr(*right, node)),
        Expr::Sub(left, right)           => evaluate_expr(*left, node).sub(evaluate_expr(*right, node)),
        Expr::Mul(left, right)           => evaluate_expr(*left, node).mul(evaluate_expr(*right, node)),
        Expr::Div(left, right)           => evaluate_expr(*left, node).div(evaluate_expr(*right, node)),
        Expr::Modulo(left, right)        => evaluate_expr(*left, node).modulo(evaluate_expr(*right, node)),
        Expr::Literal(value)             => value,
        Expr::Attribute(attribute)       => {
            match node.attribute_values.get(&attribute.field) {
                Some(literal) => (*literal).clone(),
                None => Literal::Boolean(false),
                //panic!("Field {:?} does not exist!", &attribute.field)
                }
            },
        Expr::BuiltIn(_, function) => {
            match function {
                BuiltIn::Label => Literal::Str(node.label[0].clone()),
                BuiltIn::Id => Literal::Float(node.id as f32),
                _ => panic!("Function {:?} not supported for nodes", function)
            }
        }
    }
}
```

5 Results and Discussion

5.1 Hardware

All the experiments were conducted on an ETH server (exact model?). This machine possesses 4 x 12 AMD Opteron 6174 processors, each one running at 2.2 GHz. The operating System was Debian 7.0 and the installed memory was 128 GB of RAM.

5.2 Topologies

We select two distinctly different topologies: fattree [**?**] and jellyfish [**?**]. fattree is a modification of the leaf-spine structure, commonly found in datacenters, and it is thus a practically deployable solution. jellyfish is inspired by work on random graphs showcasing that random structures can deliver shorter average paths but is more of an ideal-case reference rather than a realistic topology. 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 small fattree topology has the same vertices to edge ratio as the large fattree, but it is overall much smaller. It contains 17'665 vertices, of which 1'280 are switches. Furthermore the graph possesses a total of 16' 400 bidirectional edges. (needs better formulation)

Large Jellyfish The large jellyfish topology matches the large fattree in terms of vertices, but contains far less edges. It contains 56'160 vertices, of which 864 are switches. Furthermore the graph possesses a total of 6'910 bidirectional edges. (needs better formulation)

Small Jellyfish The small jellyfish topology is the smallest one of the four topologies. It contains 16'770 vertices, of which 390 are switches. Furthermore the graph possesses a total of 2'145 bidirectional edges. (needs better formulation)

5.3 Expectations

In all topologies, weight is an attribute of the edges and uniformly distributed from 1 to 9. Consequently, the selection 'weight <2' will therefore remove around 89% of all edges. A lesser amount of edges immensely shortens the duration of any subsequent join process.
Since joining the entire vertex and edge set is a very costly operation, it is possible that

queries with multiple, but restricted joins are faster than such, which contain fewer joins, but do not include any restrictions on the edge set.

Furthermore, it is important to consider the fact that the attributes of vertices and edges in a Differential Dataflow Collection are stored in a Vector of (String, Literal) tuples. We chose this suboptimal implementation since Differential Dataflow does not aprreciate HashMaps at all. However, in order to do a selection on either set, this Vector has to be transformed back into a HashMap for each edge respectively vertex. This process can be quite timeconsuming, therefore the more selections a query contains, the slower it expected to be.

It should also be noted that one single edge in the query triggers two joins in the evaluation. Reason being that we have to join the edge collection twice with the vertex collection, in order to determine all adjacent vertices of a given vertex.

In total, we ran seven different queries, with each subsequent query being more complex than the preceding one. For each query the resulting dataflow is drawn on the following pages. The numbers on the edges indicate the size of (intermediate) result set on the large fattree.

5.4 Overview

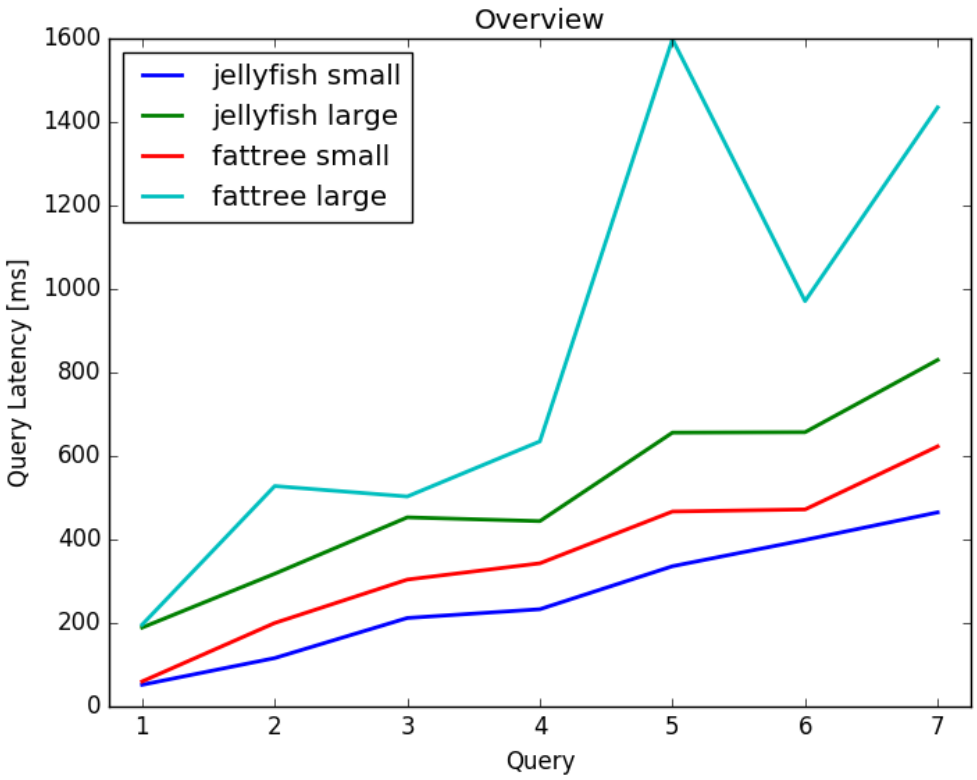


Figure 3: An overview of the seven queries, run with 32 workers

Throughout all seven queries, the following pattern is noticeable:
The influence of the number of workers behaves unexpectedly, especially for queries 1,4 and 6. In a perfect world, the query latency would be inversely proportional to the number of workers. However, we only observe this behaviour for query 5, and only for the fattree topologies.
More often than not, an increase of worker threads leads to a higher latency, especially when the latency is already quite low, around the 50 to 200 millisecond range. We reason that this happens because at that level, the communication overhead dominates the time savings achieved by parallelism. This claim is additionally supported by the fact, that query 5 has one of the longest latencies, ranging up to 10 seconds. When evaluating queries with lower latency, the contention between the workers increases and we frequently observe that the best performance is achieved when running the query with just a single worker.

Furthermore graphs with smaller sizes show larger variance than their counterpart when

run with more than four workers. This behaviour is due to the fact that the time spent coordination the different threads plays a bigger role in small graphs. The time savings achieved by parallelization are outweighed by the communication overhead.

The following pages contain the dataflows executing the query and the evaluation times distribution for each of the seven queries. While describing the dataflows, we use the following notation:

Projections are denoted by the symbol π. They always occur at the very end of a query.

Selections are denoted by the symbol σ..

Selections are denoted by the symbol $\bowtie$. .

5.5 Query 1

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

Peak Memory: 377'840 kb
Result size: 1'152
2 simple selections, 0 joins. One pass through the vertices collection is enough to produce the result.

Dataflow

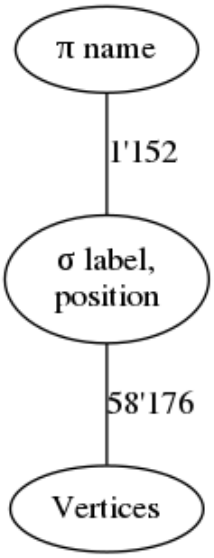
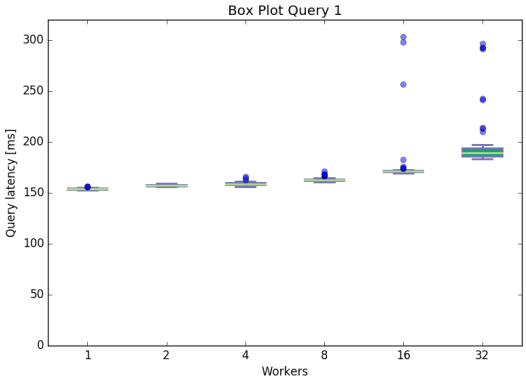
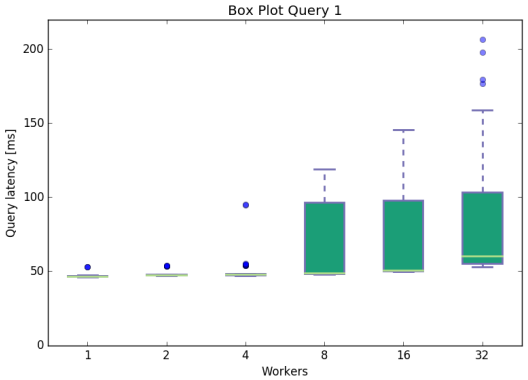


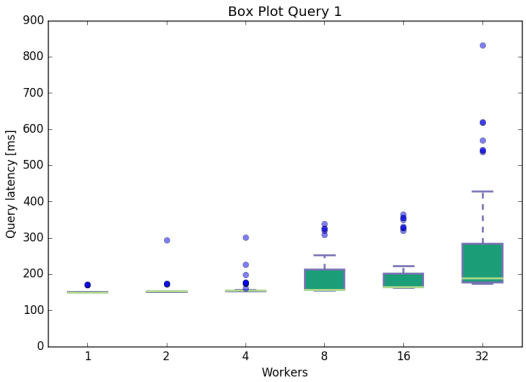
Figure 4: Dataflow Query1



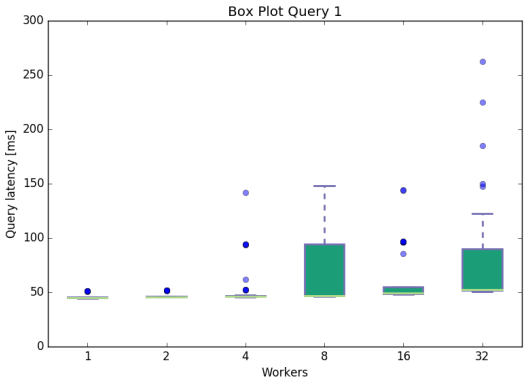
(a) Large Fat Tree



(b) Small Fat Tree



(c) Large Jellyfish



(d) Small Jellyfish

Figure 5: Boxplots for query 1

5.6 Query2

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

Peak Memory: 3'597'452 kb
Result size: 36'542
2 Joins and 1 Selection. The joins are not very expensive since we only use about 30% of the edges.

Dataflow

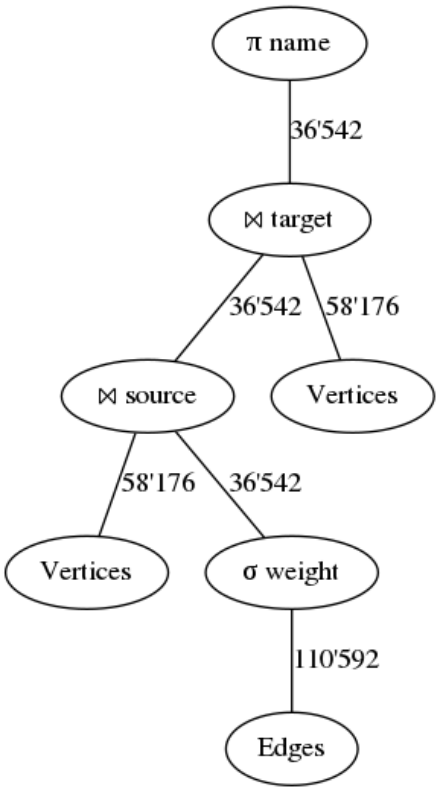
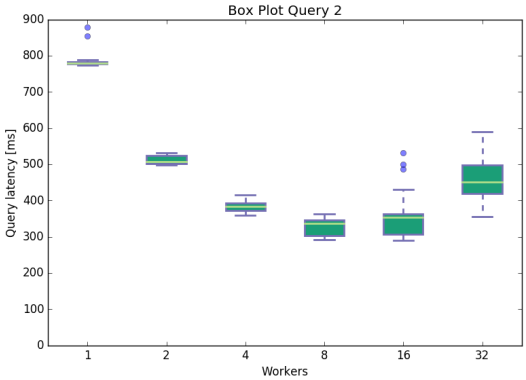
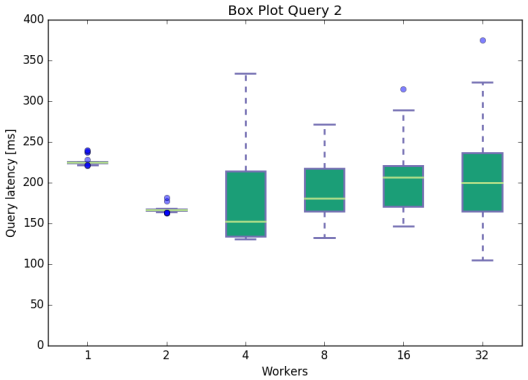


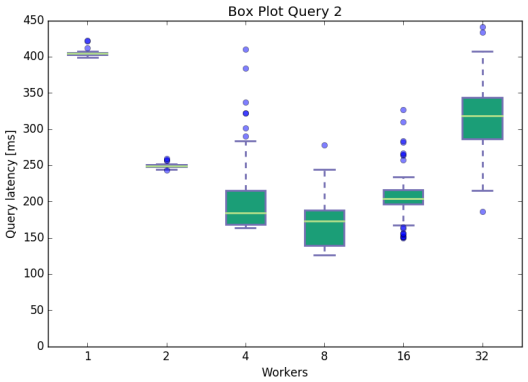
Figure 6: Dataflow Query2



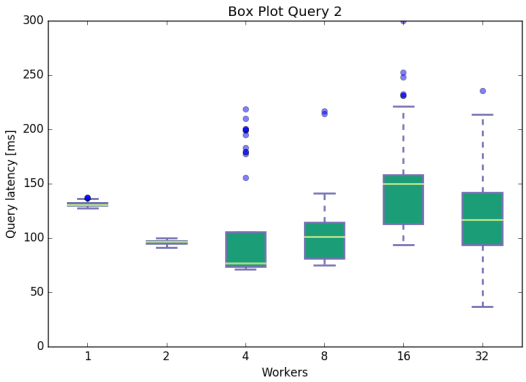
(a) Large Fat Tree



(b) Small Fat Tree



(c) Large Jellyfish



(d) Small Jellyfish

Figure 7: Boxplots for query 2

5.7 Query3

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

Peak Memory: 1'433'376 kb
Result size: 55'296
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.

Dataflow

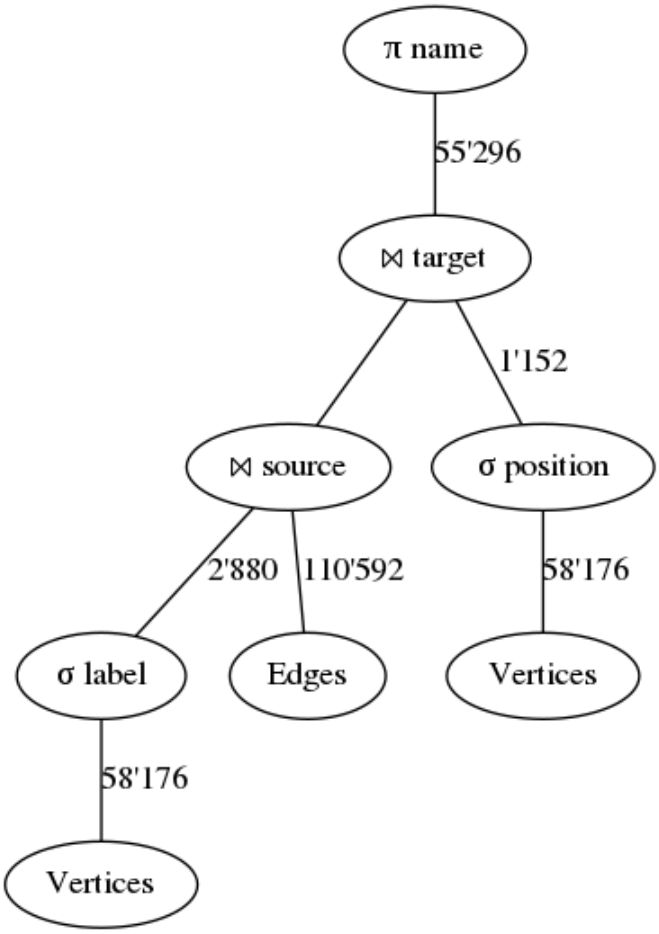
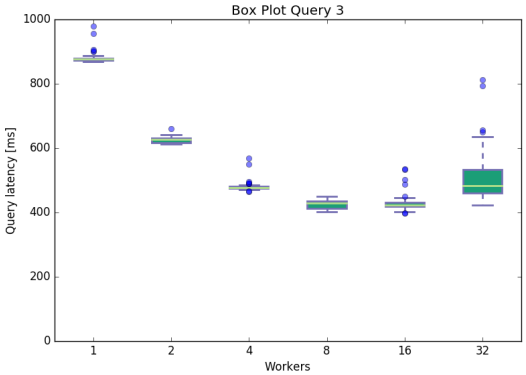
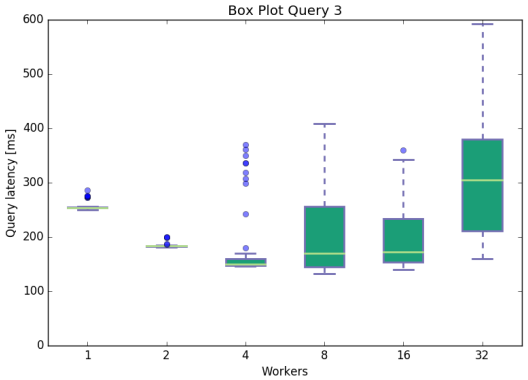


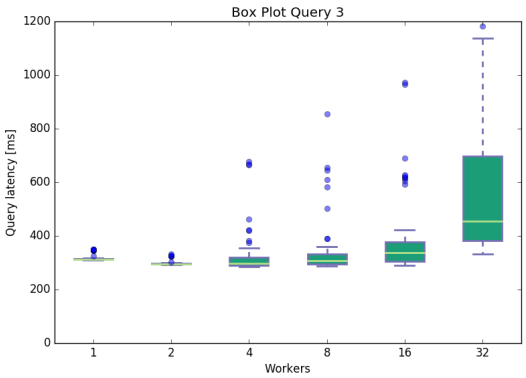
Figure 8: Dataflow Query3



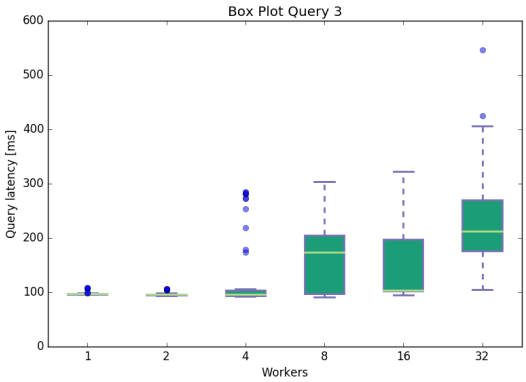
(a) Large Fat Tree



(b) Small Fat Tree



(c) Large Jellyfish



(d) Small Jellyfish

Figure 9: Boxplots for query 3

5.8 Query4

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

Peak Memory: 1'114'896 kb
Result size: 3'097
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.

Dataflow

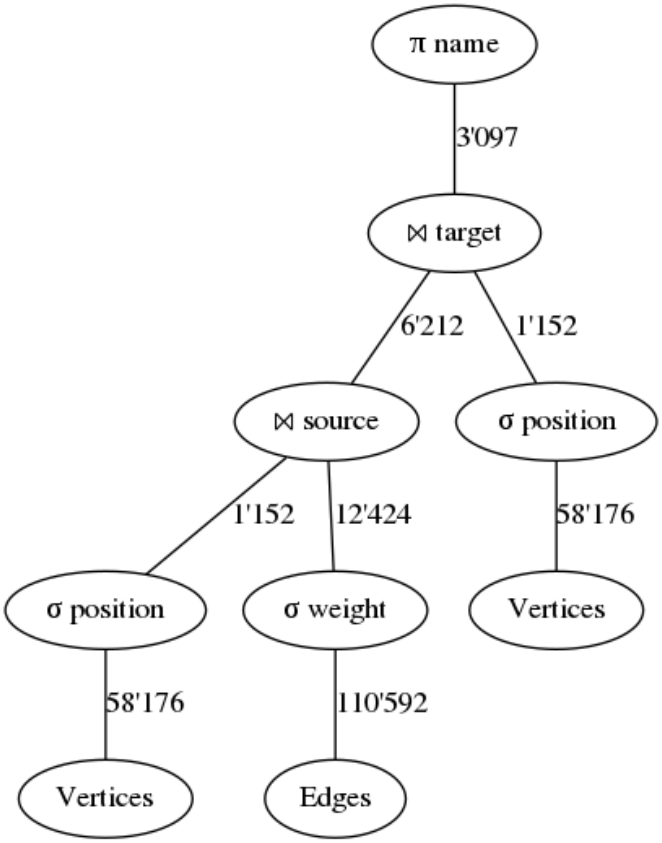
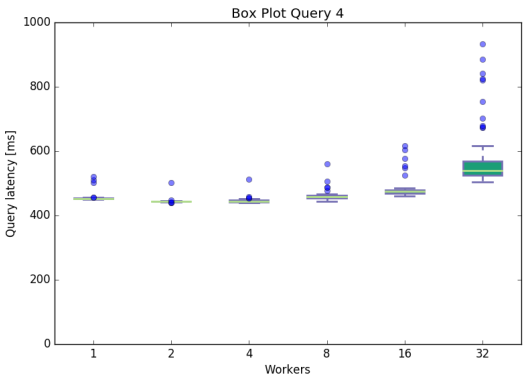
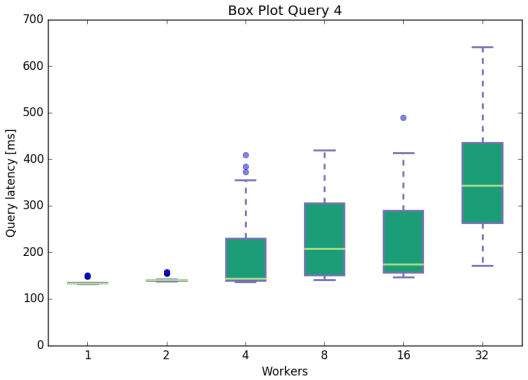


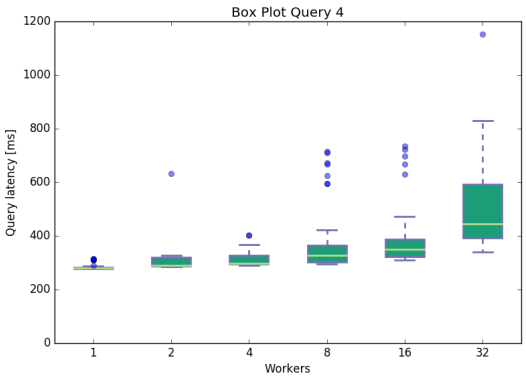
Figure 10: Dataflow Query4



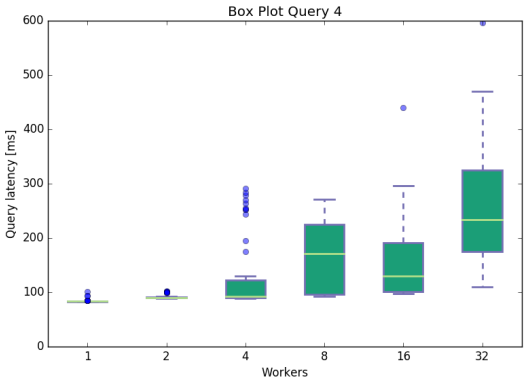
(a) Large Fat Tree



(b) Small Fat Tree



(c) Large Jellyfish



(d) Small Jellyfish

Figure 11: Boxplots for query 4

5.9 Query5

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

Peak Memory: 3'797'976 kb
Result size: 663'552
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.

Dataflow

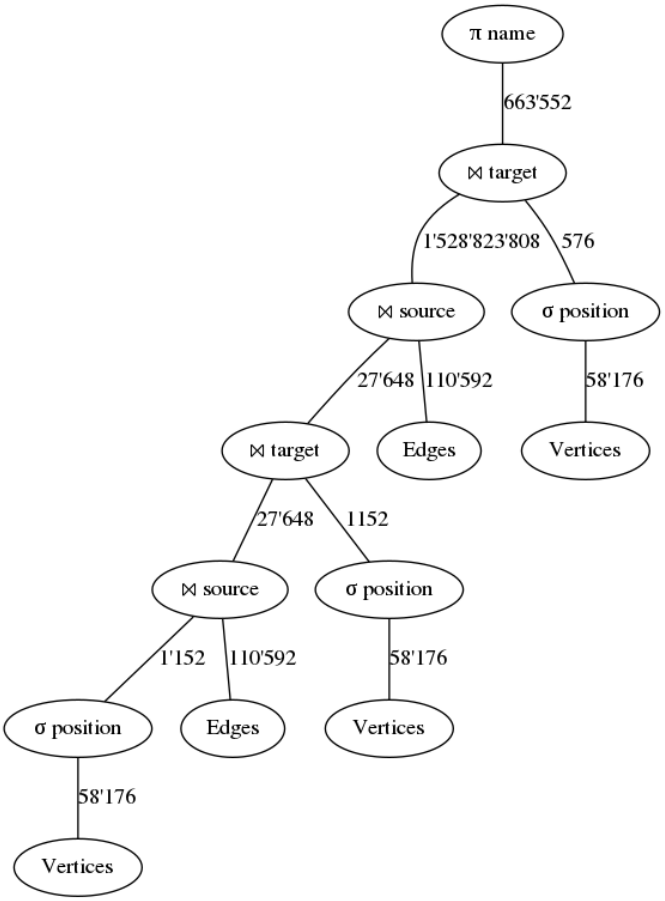
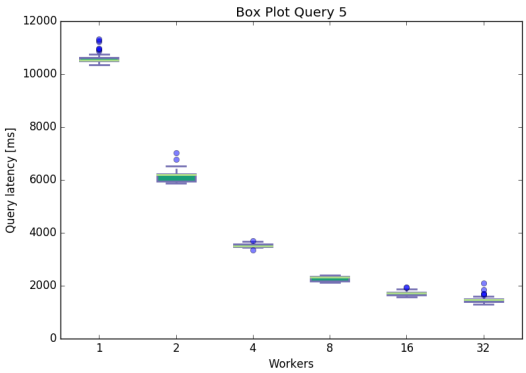
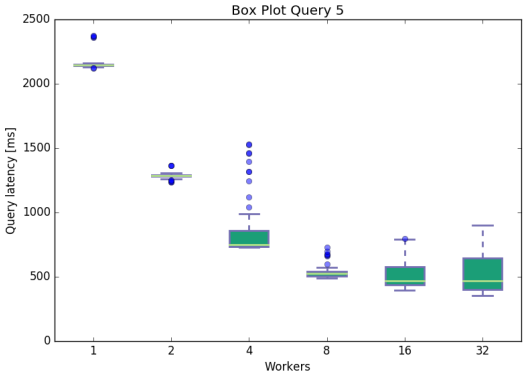


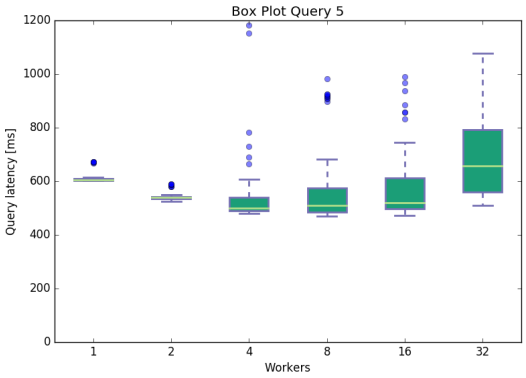
Figure 12: Dataflow Query5



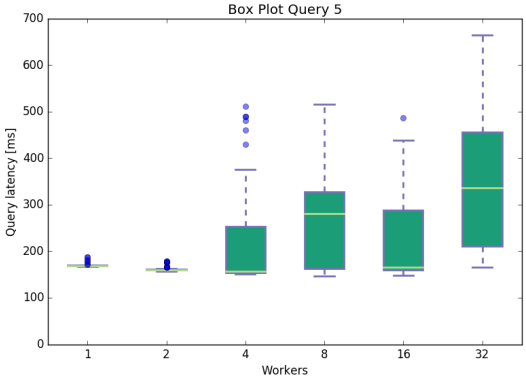
(a) Large Fat Tree



(b) Small Fat Tree



(c) Large Jellyfish



(d) Small Jellyfish

Figure 13: Boxplots for query 5

5.10 Query6

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

Peak Memory: 1'573'940 kb
Result size: 8'334
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.

Dataflow

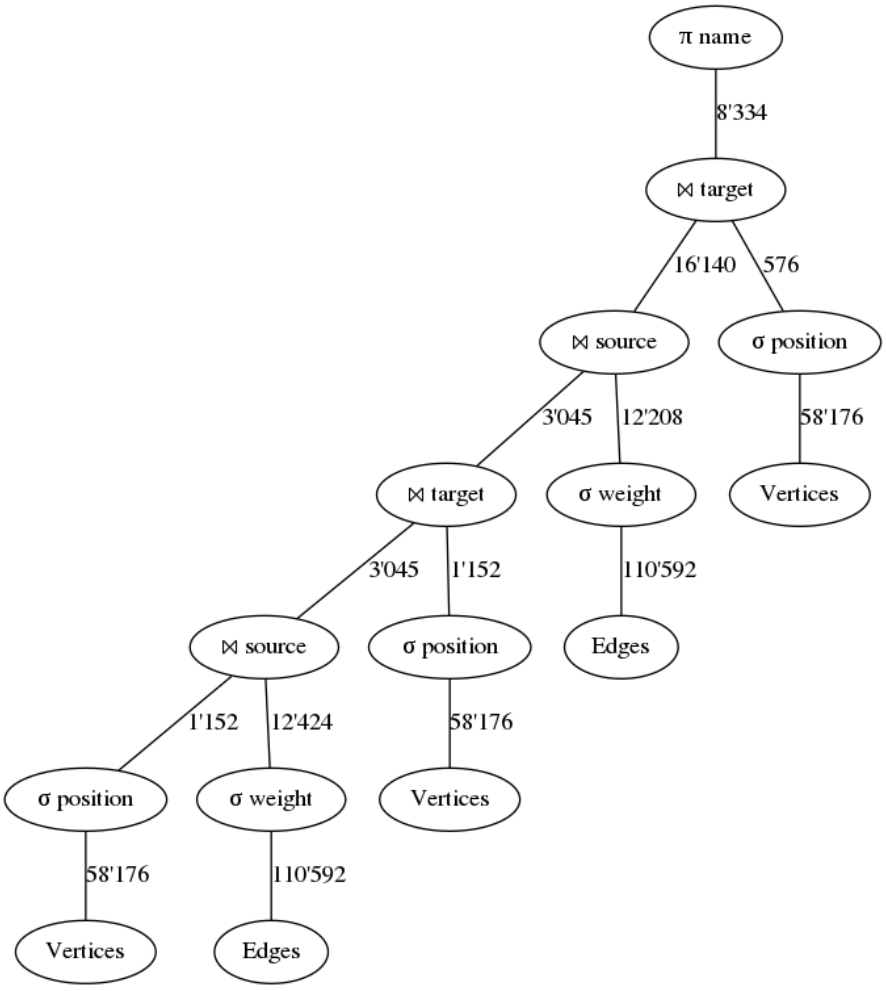
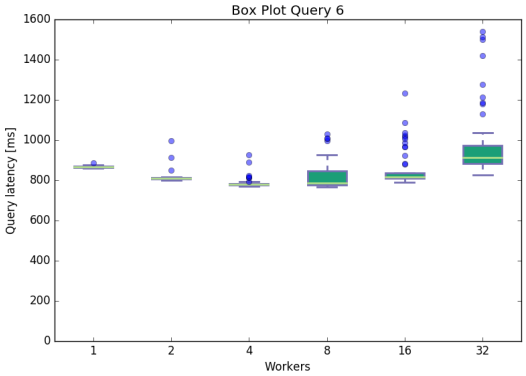
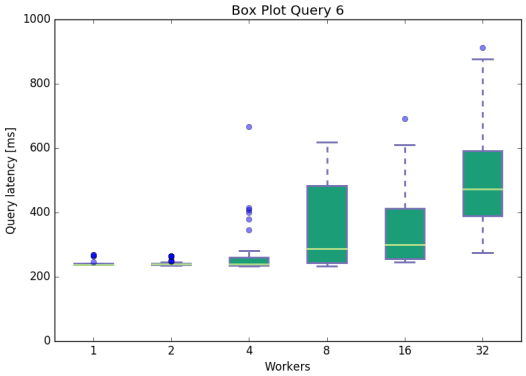


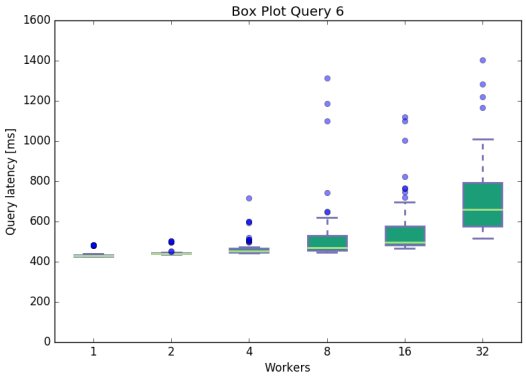
Figure 14: Dataflow Query6



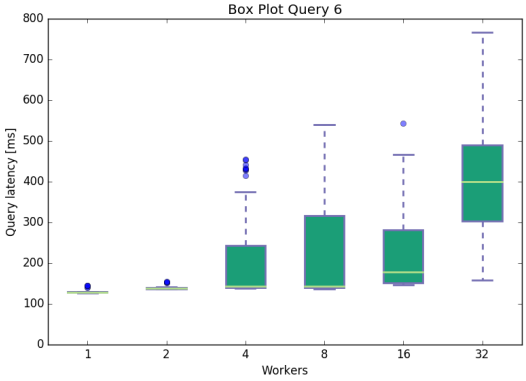
(a) Large Fat Tree



(b) Small Fat Tree



(c) Large Jellyfish



(d) Small Jellyfish

Figure 15: Boxplots for query 6

5.11 Query7

```
SELECT v.name 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')
```

Peak Memory: 2'210'824 kb
Result size: 49'956
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.

Dataflow

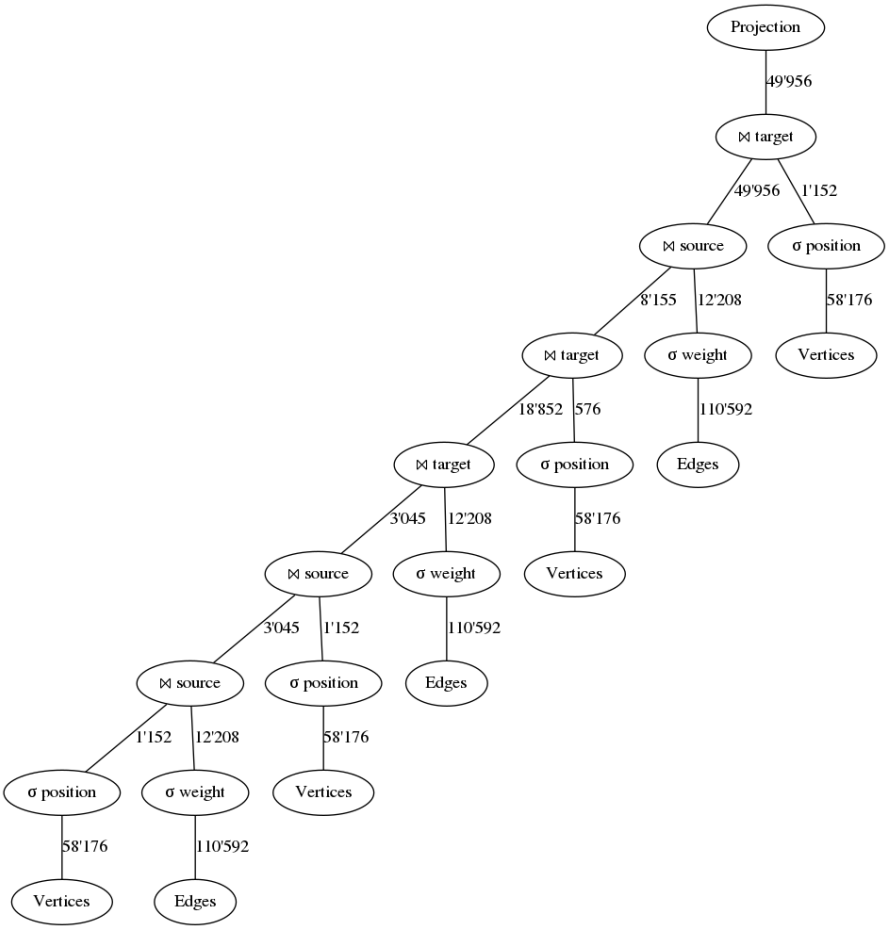
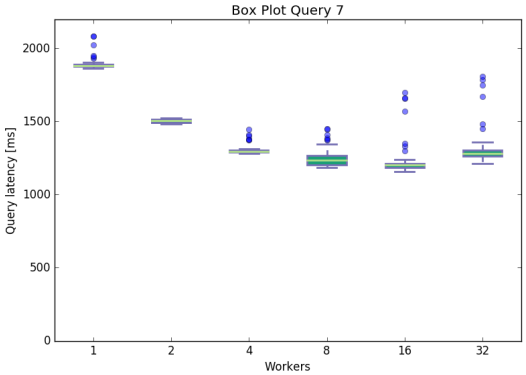
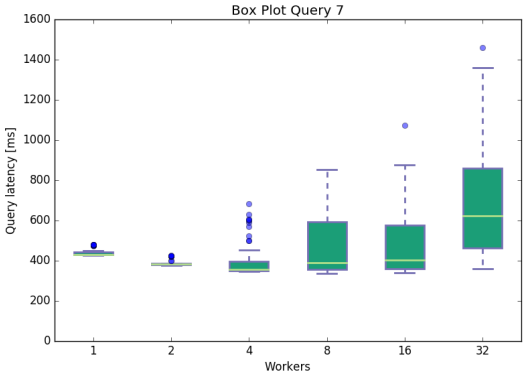


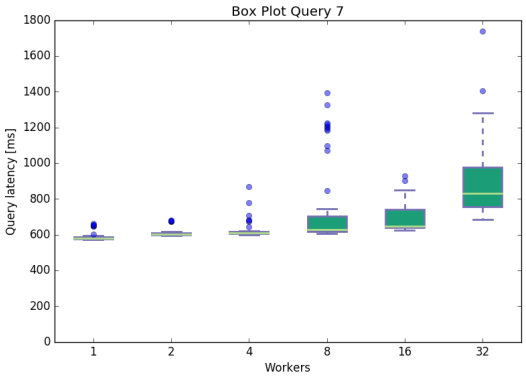
Figure 16: Dataflow Query7



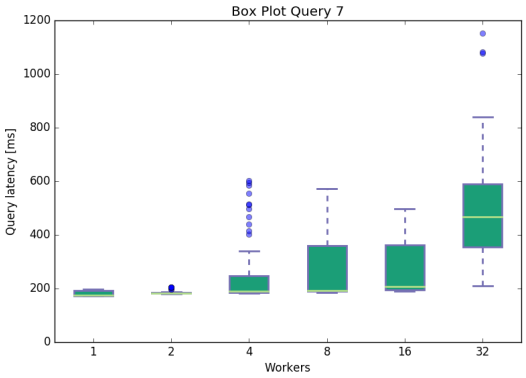
(a) Large Fat Tree



(b) Small Fat Tree



(c) Large Jellyfish



(d) Small Jellyfish

Figure 17: Boxplots for query 7

6 Related Work

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

6.1 SPARQL

RDF is a directed, labeled graph data format for representing information in the Web. This specification defines the syntax and semantics of the SPARQL query language for RDF. SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions. SPARQL also supports extensible value testing and constraining queries by source RDF graph. The results of SPARQL queries can be results sets or RDF graphs.[**?**]

6.2 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.[**?**]

6.3 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.[**?**]

6.4 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

	PGQL	Gremlin	GraphiQL	SPARQL	PQL	Green-Marl	SQLGraph
Language							
Operating System							
Open Source							

Table 1: My caption

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. [**?**]

6.5 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. [**?**]

6.6 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. [**?**]

6.7 Overview

7 Summary

7.1 Conclusion

In this thesis we presented qlidaf, a program to evaluate PGQL queries in a dataflow environment. We showed how our program parsers queries and graphs, and then evaluates said query on the graph.

7.2 Future Work

Shortest Path Extend the evaluator to find the shortest path.
Additional Joins Enable joins in the value constraint, e.g. v.age = u.age.
Query Optimizer Implement a query optimizer.

7.3 Acknowledgements

I would like to thank my supervisors John Liagouris, Desislava Dimitrova and Moritz Hoffmann for their continuous support and help. They provided many meaningful suggestions and proposals. Their expertise and experience was invaluable.

I would also like to thank Professor Roscoe for taking the time to supervise my thesis.

References

[1] Github Frank McSherry Differential Dataflow. `https://github.com/frankmcsherry/differential-dataflow`. Accessed: 2017-03-30.

[2] Github Frank McSherry Timely Dataflow. `https://github.com/frankmcsherry/timely-dataflow`. Accessed: 2017-03-30.

[3] Github Geoffroy Couprie Nom. `https://github.com/geal/nom`. Accessed: 2017-03-30.

[4] Official Rust Website. `https://www.rust-lang.org/en-US/`. Accessed: 2017-03-30.

[5] Oracle Edge-List Format. `https://docs.oracle.com/cd/E56133_01/2.3.1/reference/loader/file-system/plain-text-formats.html`. Accessed: 2017-03-30.

[6] Oracle Official PGQL Specifications. `http://pgql-lang.org/spec/1.0/`. Accessed: 2017-03-30.

[7] Sparql Query Language. `https://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`. Accessed: 2017-03-30.

[8] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[9] 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.

[10] 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.

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

[12] 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.

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

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

[15] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 225–238, San Jose, CA, 2012. USENIX.

[16] 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.

[17] 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.