Neo4J Datalog Wrapper

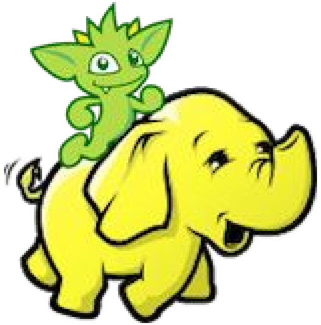
Plan

- Introduction
- General Requirements
- Project Structure
- Specific Team Requirements
- Tools
- Development Process
- Execution Plans
- Issues/Improvements
- Lessons/Future Outlook

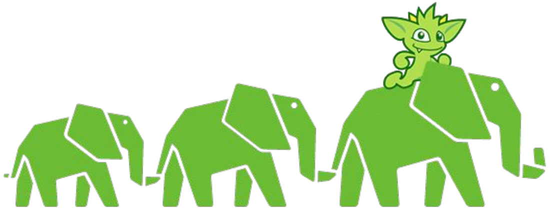
1

Introduction

Need for virtualized architecture



Team Members

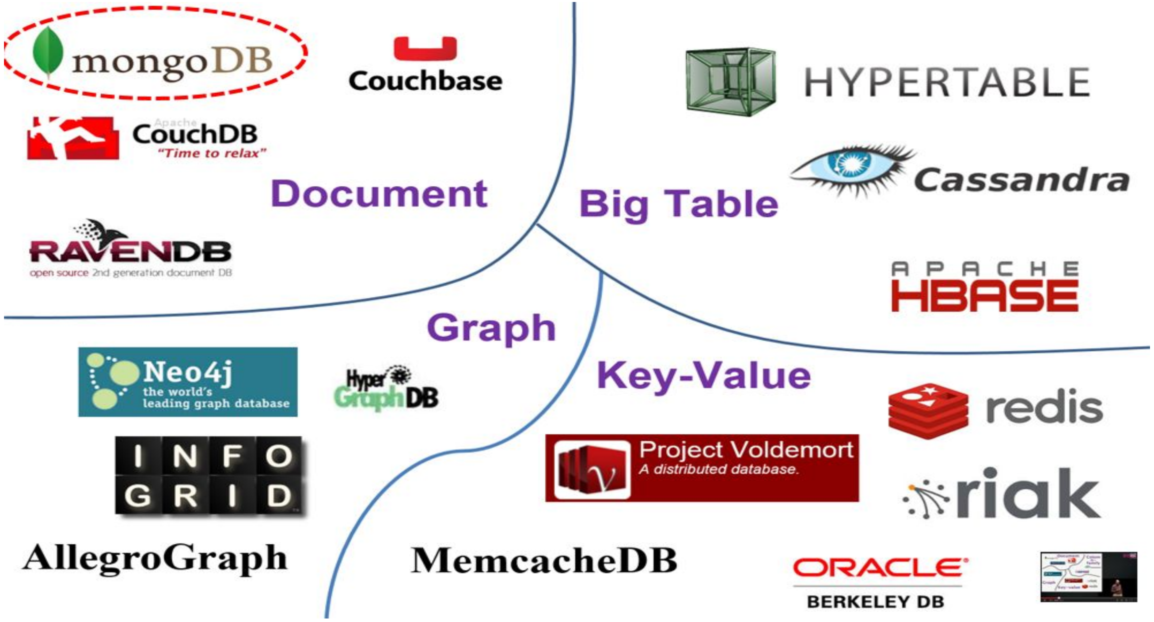


Julius Remigio

Ryan Riopelle

Michael Galarnyk

DSE 203 – Importance of Data Integration

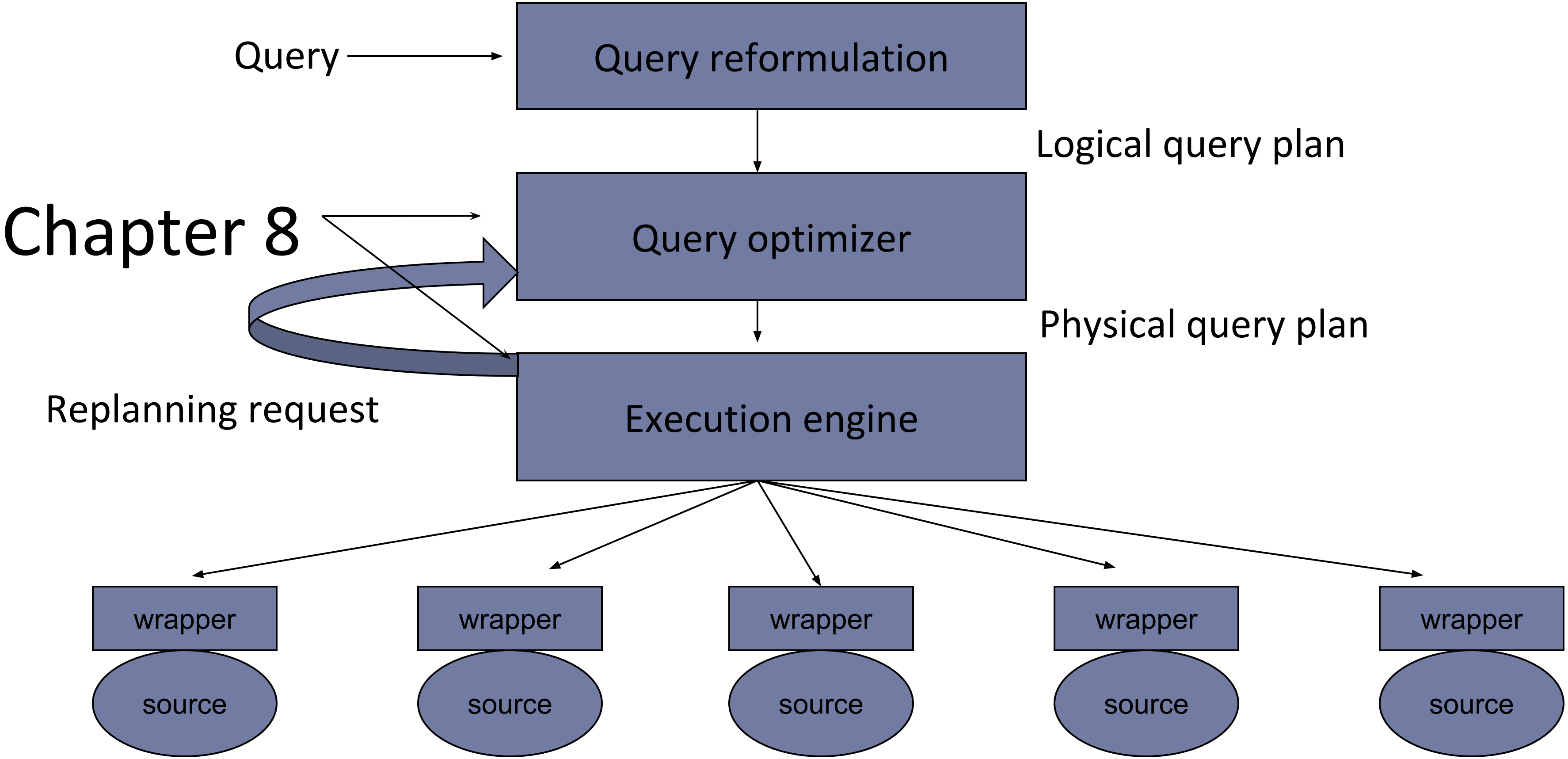

mongoDB
Couchbase
Apache CouchDB "Time to relax"
Document
RAVENDB open source 2nd generation document DB
HYPERTABLE
Cassandra
Big Table
APACHE HBASE
Graph
Neo4j the world's leading graph database
HyperGraphDB
INFO GRID
AllegroGraph
Key-Value
redis
Project Voldemort A distributed database.
riak
MemcacheDB
ORACLE BERKELEY DB

2

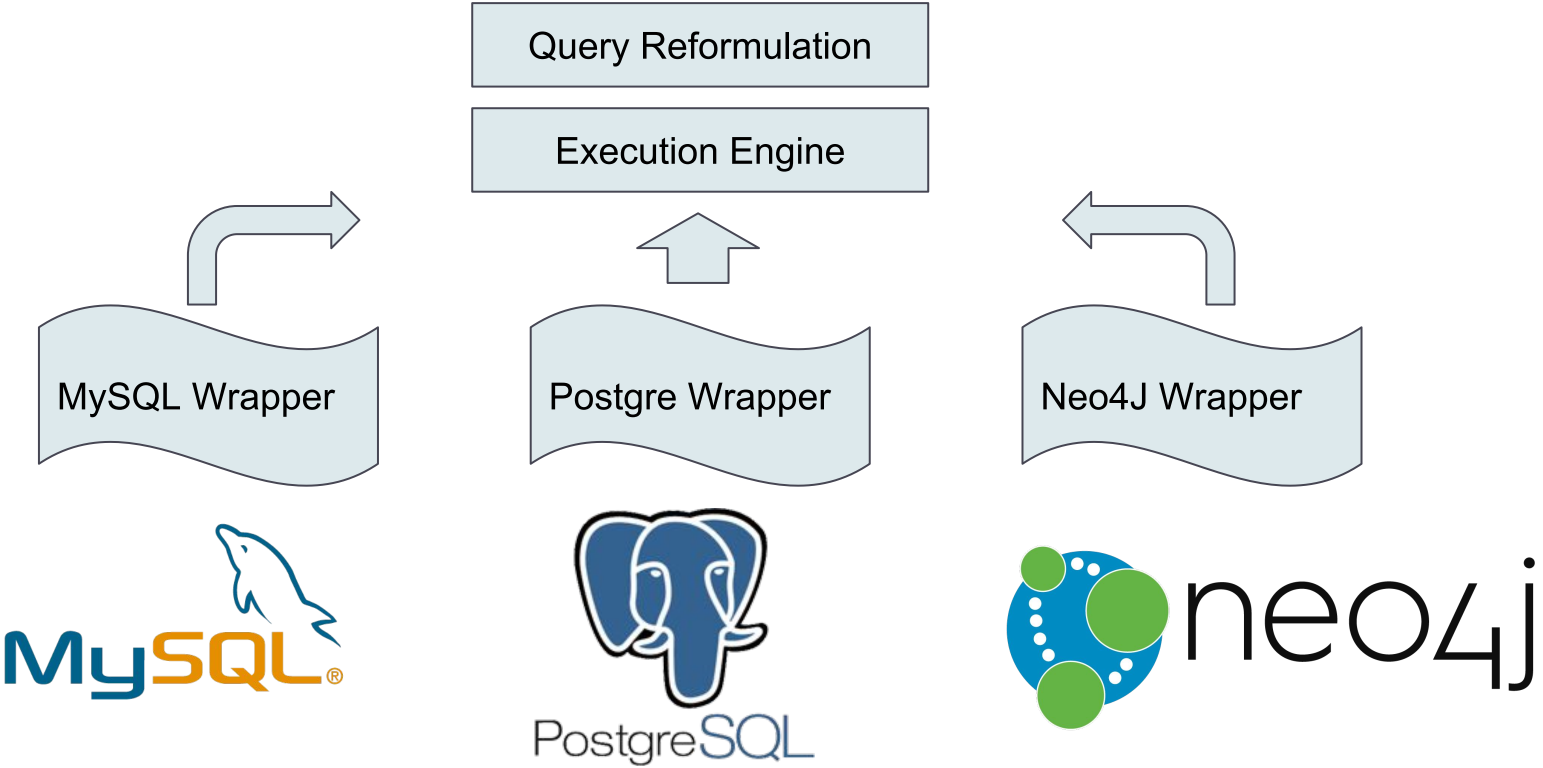
Project Structure

Need for virtualized architecture

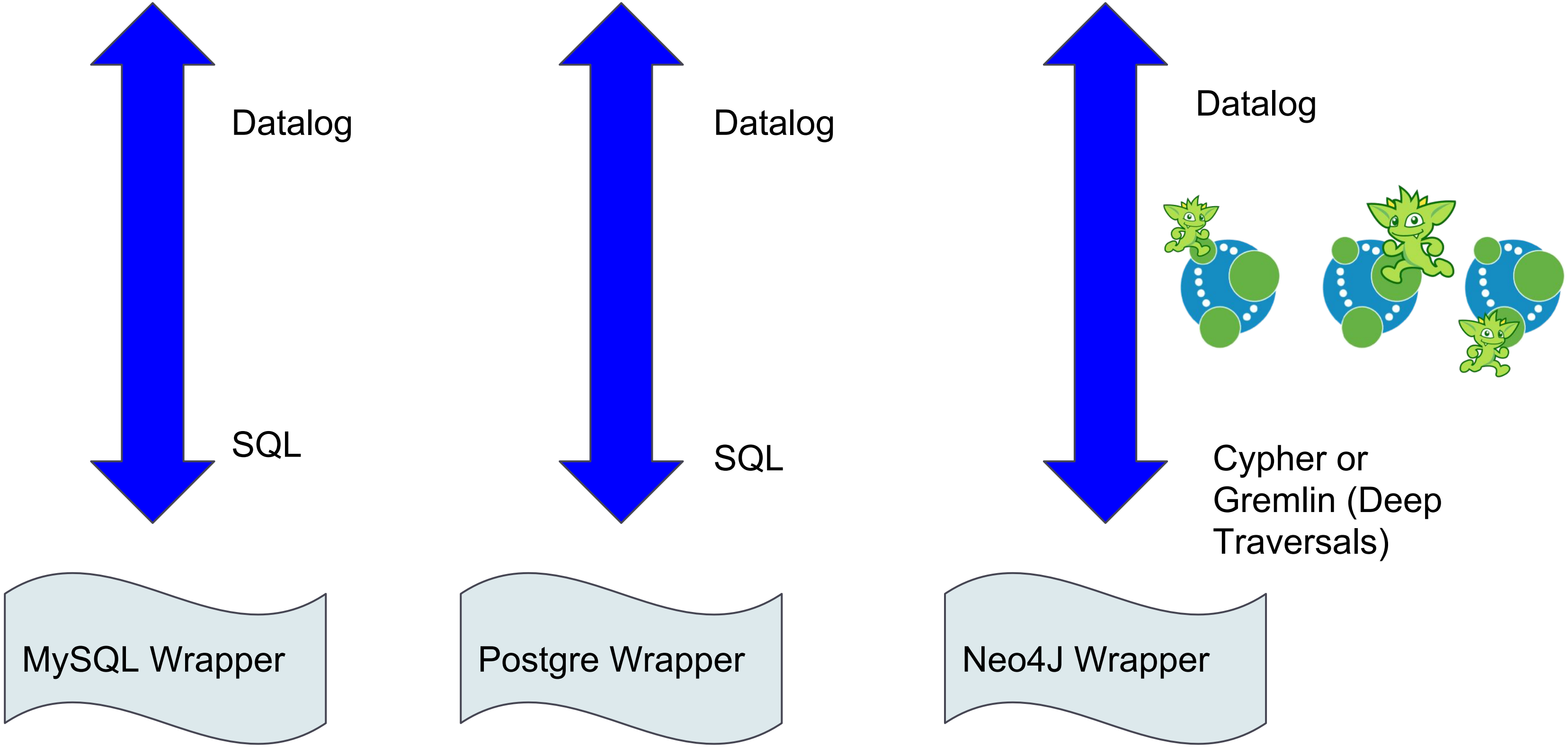
Query Processing



Virtual Data Integration Architecture



Query Languages



Datalog

SQL

MySQL Wrapper

Datalog

SQL

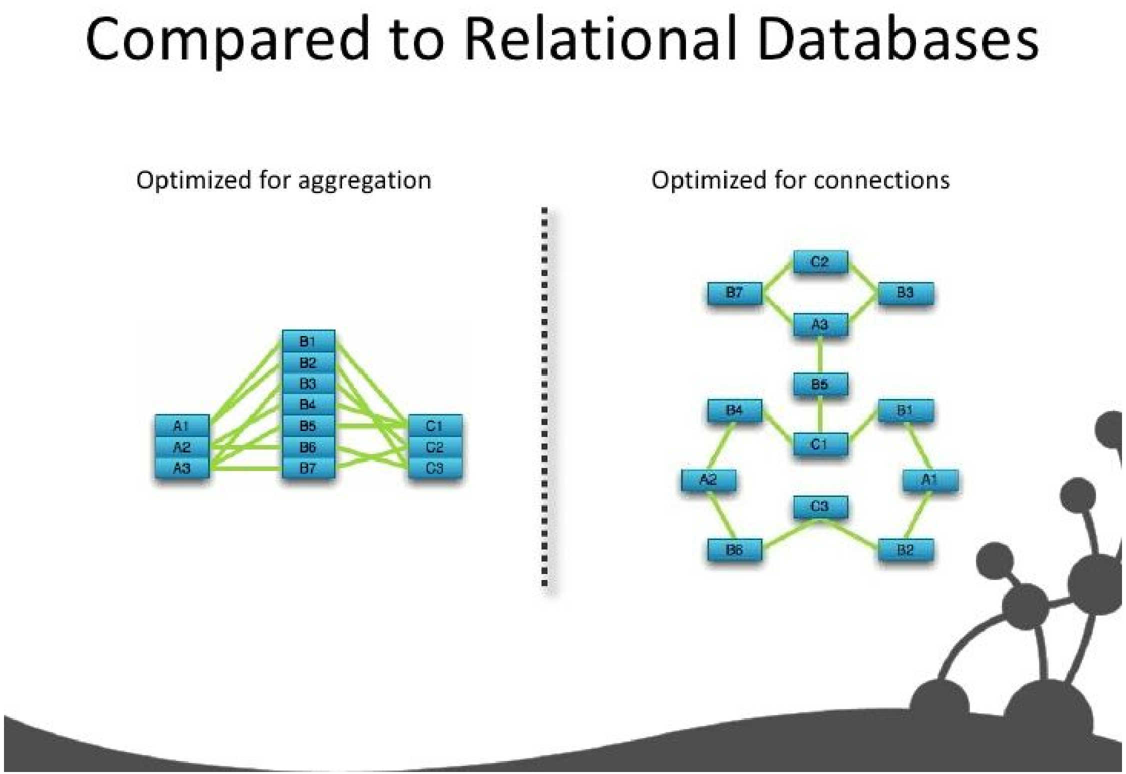
Postgre Wrapper

Datalog

Cypher or Gremlin (Deep Traversals)

Neo4J Wrapper

Graph Databases

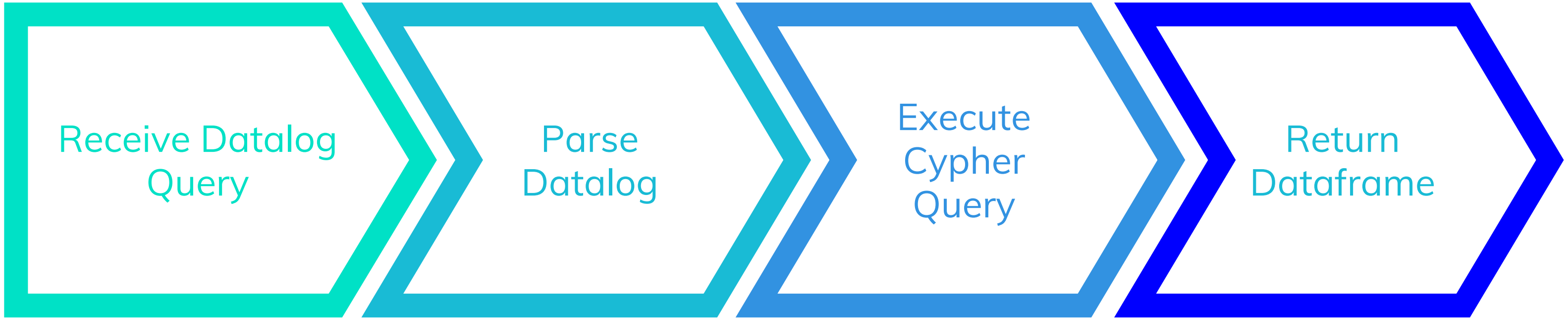


Compared to Relational Databases

Optimized for aggregation

Optimized for connections

Neo4J Wrapper Process Flow



Receive Datalog Query

Parse Datalog

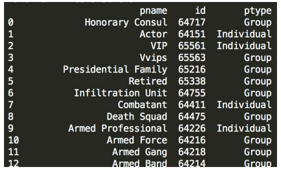
Execute Cypher Query

Return Dataframe

q(organization) :-
actor(id, _, pname, _),
affiliation(id, organization, _, _),
pname = 'Ariel Sharon'

Tables: ['affiliation', 'actor']
Columns: ['organization', 'pname', 'id']
Required Projection: ['ization']
Data is in Neo4J Schema A

Match (a: Actor {Name: 'Ariel Sharon'})-[aff:Affiliation]->(r) return r.Name as organization



```
                   pname     id       ptype
0        Honorary Consul  64717       Group
1                  Actor  64151  Individual
2                    VIP  65561  Individual
3                  Vvips  65563       Group
4    Presidential Family  65216       Group
5                Retired  65338       Group
6      Infiltration Unit  64755       Group
7              Combatant  64411  Individual
8            Death Squad  64475       Group
9     Armed Professional  64226  Individual
10           Armed Force  64216       Group
11            Armed Gang  64218       Group
12            Armed Band  64214       Group
```

3

Tools

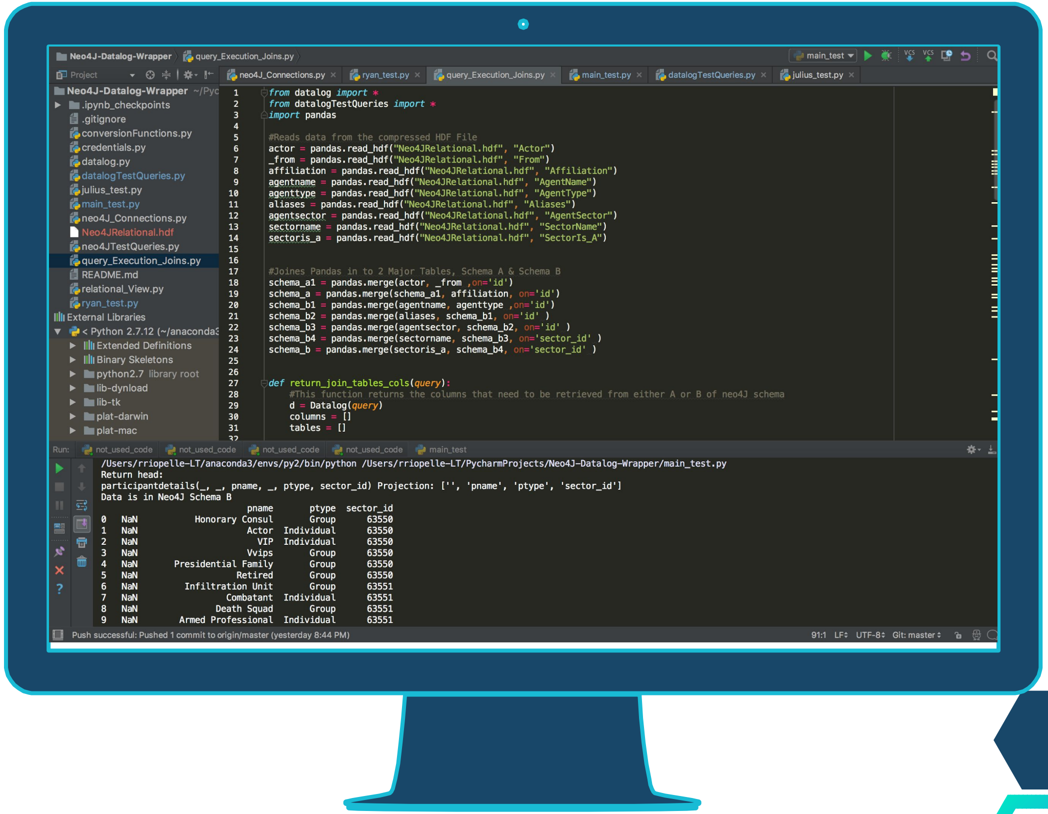
Need for virtualized architecture

NEED TO USE GITHUB!

Show and explain your web, app or software projects using these gadget templates.

PyCharm Really Helps!

Show and explain your web, app or software projects using these gadget templates.



```python
from datalog import *
from datalogTestQueries import *
import pandas

#Reads data from the compressed HDF File
actor = pandas.read_hdf("Neo4JRelational.hdf", "Actor")
_from = pandas.read_hdf("Neo4JRelational.hdf", "From")
affiliation = pandas.read_hdf("Neo4JRelational.hdf", "Affiliation")
agentname = pandas.read_hdf("Neo4JRelational.hdf", "AgentName")
agenttype = pandas.read_hdf("Neo4JRelational.hdf", "AgentType")
aliases = pandas.read_hdf("Neo4JRelational.hdf", "Aliases")
agentsector = pandas.read_hdf("Neo4JRelational.hdf", "AgentSector")
sectorname = pandas.read_hdf("Neo4JRelational.hdf", "SectorName")
sectoris_a = pandas.read_hdf("Neo4JRelational.hdf", "SectorIs_A")


#Joines Pandas in to 2 Major Tables, Schema A & Schema B
schema_a1 = pandas.merge(actor, _from ,on='id')
schema_a = pandas.merge(schema_a1, affiliation, on='id')
schema_b1 = pandas.merge(agentname, agenttype ,on='id')
schema_b2 = pandas.merge(aliases, schema_b1, on='id' )
schema_b3 = pandas.merge(agentsector, schema_b2, on='id' )
schema_b4 = pandas.merge(sectorname, schema_b3, on='sector_id' )
schema_b = pandas.merge(sectoris_a, schema_b4, on='sector_id' )


def return_join_tables_cols(query):
    #This function returns the columns that need to be retrieved from either A or B of neo4J schema
    d = Datalog(query)
    columns = []
    tables = []
```

```
/Users/rriopelle-LT/anaconda3/envs/py2/bin/python /Users/rriopelle-LT/PycharmProjects/Neo4J-Datalog-Wrapper/main_test.py
Return head:
participantdetails(_, _, pname, _, ptype, sector_id) Projection: ['', 'pname', 'ptype', 'sector_id']
Data is in Neo4J Schema B
                    pname       ptype  sector_id
0   NaN       Honorary Consul       Group      63550
1   NaN                 Actor  Individual      63550
2   NaN                   VIP  Individual      63550
3   NaN                 Vvips       Group      63550
4   NaN    Presidential Family       Group      63550
5   NaN               Retired       Group      63550
6   NaN      Infiltration Unit       Group      63551
7   NaN             Combatant  Individual      63551
8   NaN           Death Squad       Group      63551
9   NaN     Armed Professional  Individual      63551
```

Push successful: Pushed 1 commit to origin/master (yesterday 8:44 PM)

4

Execution Plans

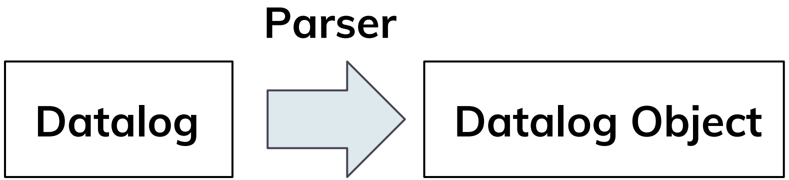
Need for virtualized architecture

What is a wrapper?

- Wrappers are components of DI systems that communicate with the data sources
  - sending queries from higher levels in the system to the sources
  - converting replies to a format that can be manipulated by query processor
- Complexity of wrapper depends on nature of data source
  - e.g., source is RDBMS, wrapper's task is to interact with JDBC driver
  - in many cases, wrapper must parse semi-structured data such as HTML pages and transform it into a set of tuples
  - we focus on this latter case

Datalog Parser Overview

- Class Datalog (datalog.py)
- Input: Standard Datalog
- Strategy: regular expressions
- Output: Datalog Object

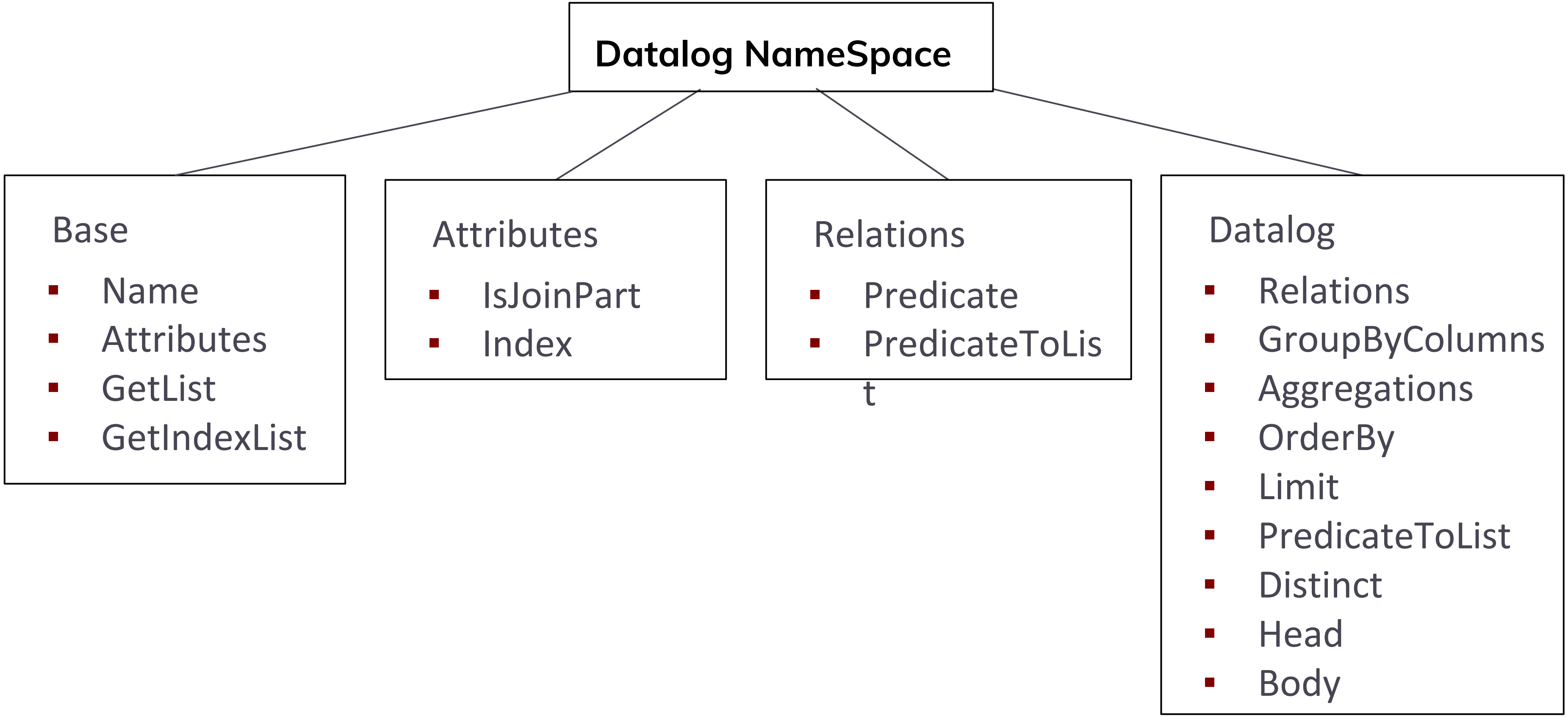


Datalog Classes

Parsers consists of a set of classes and a super class

- DatalogBase(*object*)
- DatalogAttribute(*DatalogBase*)
- DatalogRelation(*DatalogBase*)
- Datalog(*Base*)

Datalog NameSpace and Members



Datalog NameSpace

Base

- Name
- Attributes
- GetList
- GetIndexList

Attributes

- IsJoinPart
- Index

Relations

- Predicate
- PredicateToList

Datalog

- Relations
- GroupByColumns
- Aggregations
- OrderBy
- Limit
- PredicateToList
- Distinct
- Head
- Body

Inspecting Datalog Object Schema

query: A(a, b , e) :- mytable(a, b, c, d, _), other(f, g, h), CONTAINS(g,'foo'),a > 1, GROUP_BY([a, b], d = COUNT(c)), d < 100, SORT_BY(b, 'DESC'), f = FUN(e), LIMIT(25), DISTINCT

```
def inspect(query):
  # use to inspect object properties
  d = Datalog(query)
  print 'query:', query
  print 'head:',d.head
  print 'name:',d.name
  print 'projection:', d.getList
  print 'Join keys:',d.joinKeys
  for x in d.relations:
     print x.name
     print 'predicate:', x.predicateToList
     for a in x.attributes:
        print '\t', x.attributes[a].name, '\t', \
           x.attributes[a].index, '\t',\
           x.attributes[a].isJoinPart, '\t',\
           x.predicate[x.attributes[a].index] if x.attributes[a].index in x.predicate else
   ''
  print 'group by:', d.groupBy
  print 'grouping columns:', d.groupByColumns
  print 'aggregations:', d.aggregations
  print 'having clause:', d.predicateToList
  print 'order by:', d.orderBy
  print 'limit:', d.limit
  print 'distinct:', d.distinct
```

Sample Output

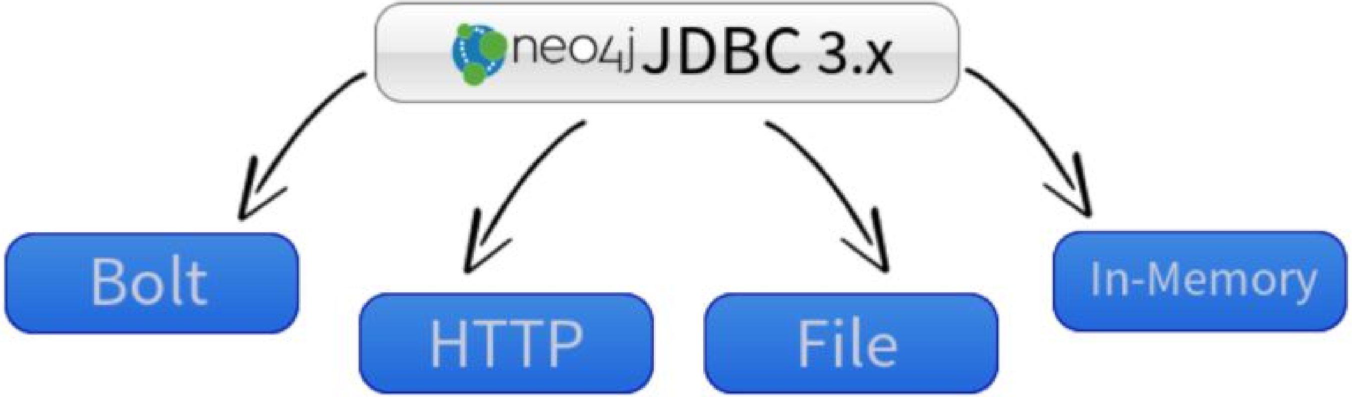
```
 head: A(a, b , e)
name: A
projection: ['a', 'b', 'e']
Join keys: []
mytable
predicate: []
   a    0    False    a > 1
   c    2    False
   b    1    False
   d    3    False    d < 100
other
```

```
predicate: ["g = CONTAINS('foo')"]
   h    2    False
   g    1    False    CONTAINS('foo')
   f    0    False    f = FUN(e)
group by: ['GROUP_BY([a, b], d =
   COUNT(c))']
grouping columns: ['a', 'b']
aggregations: ['COUNT(c) AS d']
having clause: []
order by: ORDER BY b DESC
limit: 25
distinct: True
```

Solution

- Standardized syntax makes things easier
  - All functions represented in CAPS
  - All relations represented in lowercase
- Limiting use of nesting to a few cases
  - Write special code to identify these cases and handle appropriately

Connection Strings



Accessing Neo4J Using Python

- Neo4j Python Driver
- The Example Project
- Neo4j Community Drivers
- Py2neo
- Neo4jRestClient
- Bulbflow

```
from neo4j.v1 import GraphDatabase, basic_auth
driver = GraphDatabase.driver("bolt://54.85.112.231:7687",
auth=basic_auth("neo4j", "LEbKqX3q"))
session = driver.session()
```

```
authenticate("54.85.112.231:7474", "neo4j", "LEbKqX3q")
graph = Graph("bolt://54.85.112.231/db/data/")
```

Query Execution Classes

Parsers consists of a set of classes and a super class

- ReturnJoinTablesCols (*query*)
- ReturnSchemaA_or_B(*tables, columns*)
- ProjectedDataOutput(*dataframe*)
- ExecuteQuery(query)

Cypher Converted to Relational View

Cymer Relational View

- Actor: [u'AliasList' u'id' u'pname' u'ptype']
- From: [u'country' u'id']
- Affiliation: [u'end' u'id' u'org' u'start']
- AgentName: [u'id' u'pname']
- AgentType: [u'id' u'ptype']
- Aliases: [u'alias' u'id']
- AgentSector: [u'id' u'sector_id']
- SectorName: [u'name' u'sector_id']
- SectorIs_A: [u'sector_id' u'sector_id2']

Compared to MySQL

- Single GTD Table

Compared to Postgre

- Two GDELT Tables

Ideal State w/ ID

DLOG: q(name) :- actor(id, _, name, _), affiliation(id, 'Taliban', _, _)

Map: <relation>⟶<node/edge>

actor (id,name)⟶**ID(<n>Actor)**, <n>Actor.Name,

affiliation(id, ‘Taliban’)⟶**ID(<n>Actor)**, <n>Organization.Name

Cypher:

```
MATCH (o: Organization {Name: 'Taliban'})-[]-(p:Actor)
    RETURN p.Name as name
```

Ideal State Arbitrary Join

DLOG: q(name) :- actor(_, _, name, _), agentname(_, pname)

Map: <relation>⟶<node/edge>

actor (id,name)⟶**ID(<n>Actor)**, <n>Actor.Name,

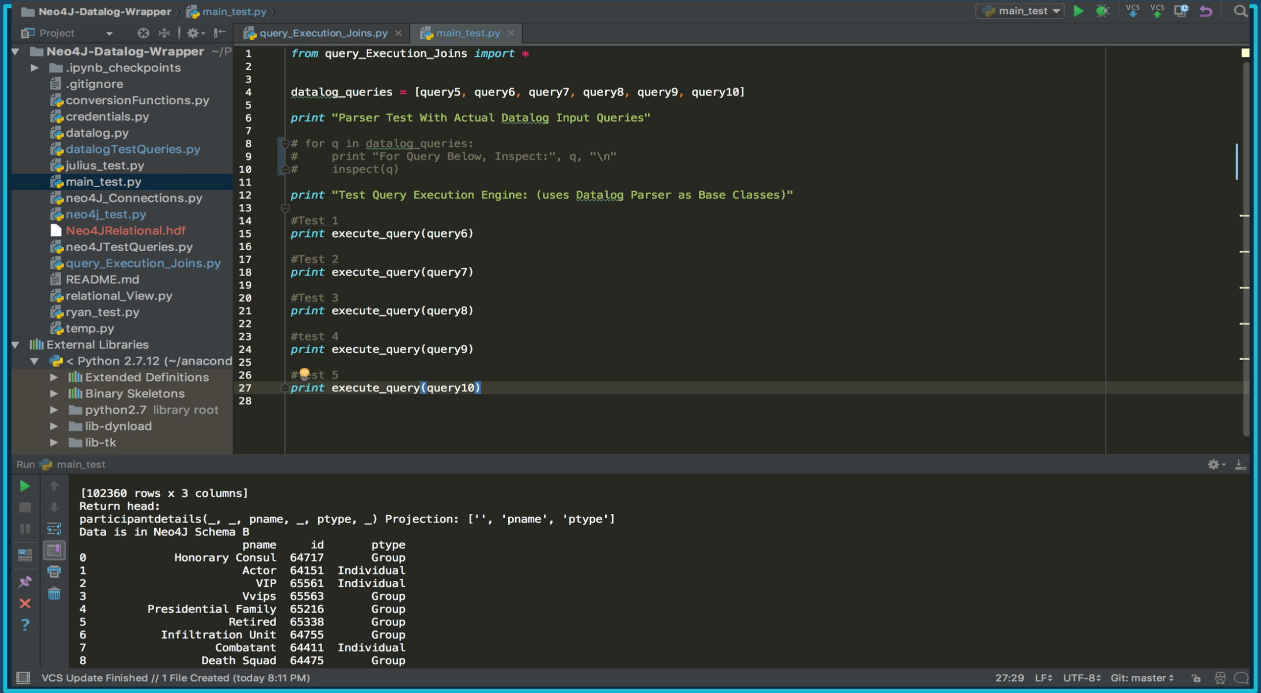
agentname(id, ‘Taliban’)⟶**ID(<n>AgentName)**, <n>Organization.Name

Cypher:

```
MATCH (o: AgentName)
MATCH (p: Actor) WHERE o.Name = p.Name
     RETURN p.Name as name
```

Example Code

Show and explain your web, app or software projects using these gadget templates.



```
from query_Execution_Joins import *


datalog_queries = [query5, query6, query7, query8, query9, query10]

print "Parser Test With Actual Datalog Input Queries"

# for q in datalog_queries:
#     print "For Query Below, Inspect:", q, "\n"
#     inspect(q)

print "Test Query Execution Engine: (uses Datalog Parser as Base Classes)"

#Test 1
print execute_query(query6)

#Test 2
print execute_query(query7)

#Test 3
print execute_query(query8)

#test 4
print execute_query(query9)

#Test 5
print execute_query(query10)

```

```
[102360 rows x 3 columns]
Return head:
participantdetails(_, _, pname, _, ptype, _) Projection: ['', 'pname', 'ptype']
Data is in Neo4J Schema B
                       pname     id       ptype
0            Honorary Consul  64717       Group
1                      Actor  64151  Individual
2                        VIP  65561  Individual
3                      Vvips  65563       Group
4         Presidential Family  65216       Group
5                    Retired  65338       Group
6           Infiltration Unit  64755       Group
7                  Combatant  64411  Individual
8                Death Squad  64475       Group
```

VCS Update Finished // 1 File Created (today 8:11 PM)

5

Issues/Improvements

Need for virtualized architecture

Issues

Nested expressions in datalog

- difficult to parse using regex

Identifying Functions vs Relations

- Syntactically both look the same
- Not sure if data should have been modeled in two separate databases

Issue - Example

```
Group By Example:
Pattern = ‘(\w+[(].+[)])’
Datalog = ‘a(x, y, a, b) :- b(x, y, z),
                 GROUP_BY([x, y], a = COUNT(z), b = SUM(Z))’
Some Possible Matches in body:
> b(x, y, z)
> b(x, y, z), GROUP_BY([x, y], a = COUNT(z), b = SUM(Z))’
> GROUP_BY([x, y], a = COUNT(z)
> COUNT(z)
> GROUP_BY([x, y], a = COUNT(z), b = SUM(Z))
```

Issues

i) Data should not be modeled in two separate schemas

ii) Cannot use surrogate key IDs for joining or tracking of data

iii) Data variables in global schema do not equal those in Neo4j or Return statements. I.e. Proposed global projection variables does not make sense (not tied directly to our portion, but didn’t seem correct)

iv) Match statements take too long to run or don't run at all

   (1) Prevents correct predicate pushdown

   (2) Provides a reason to use HDF files

v) Actually creating one single joined Neo4J does not work

6

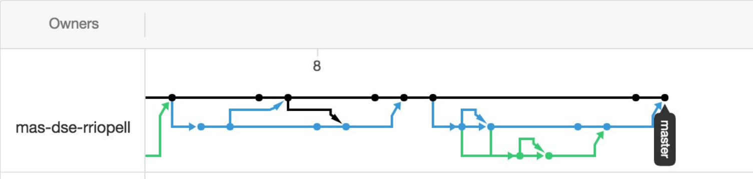
Lessons/Future Considerations

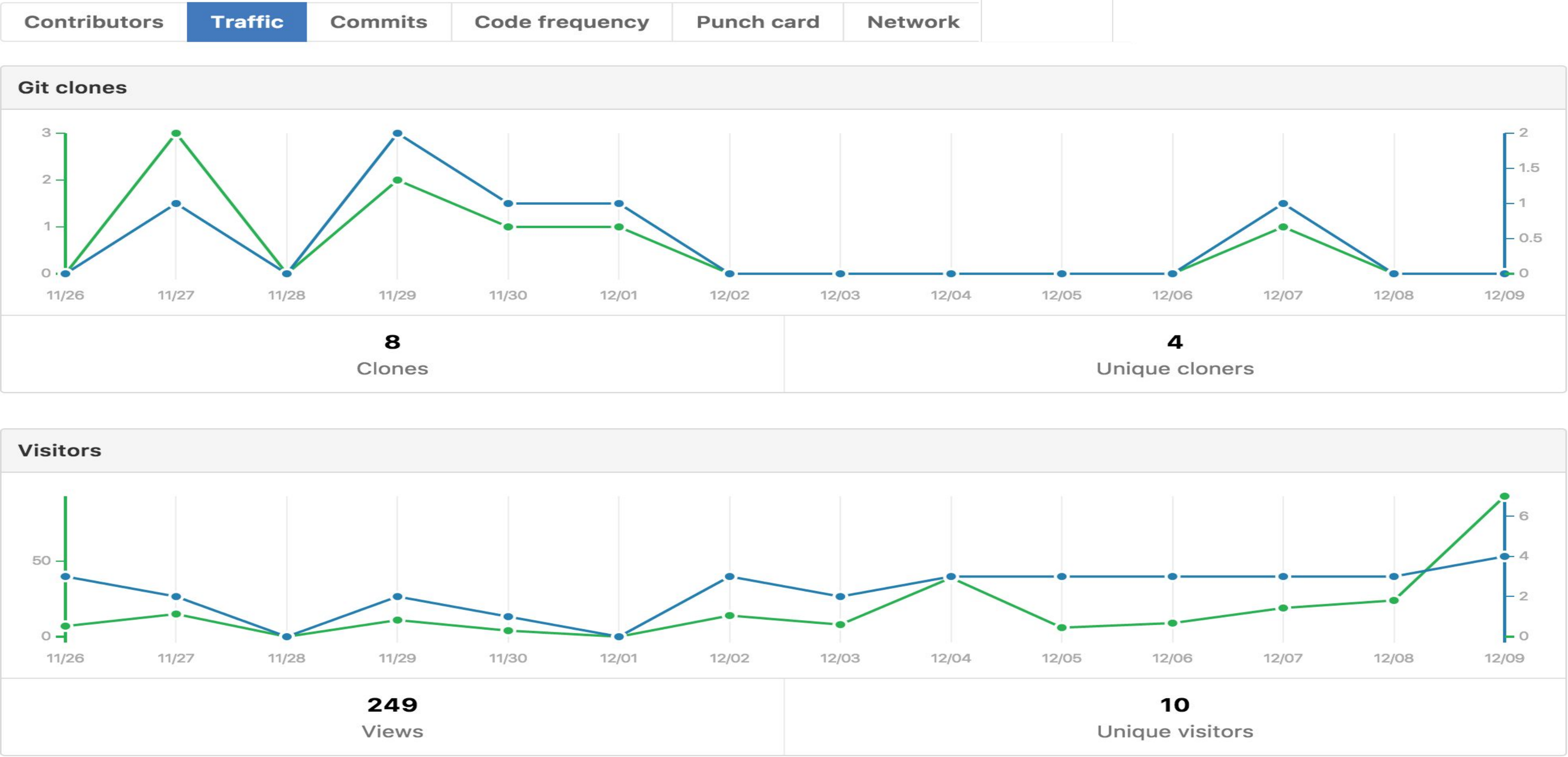
Need for virtualized architecture

DI Lessons Learned

- Regex is good for only the simple cases
    - Lexical parsers like PyParsing handle nesting better and allow for user defined grammar
    - Understanding the graph database schema is necessary to translate
- It’s not easy!

Github Stats





Git clones

8
Clones

4
Unique cloners

Visitors

249
Views

10
Unique visitors

Thanks!

Any questions?