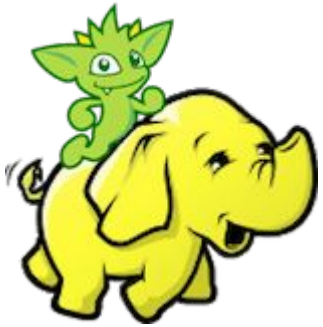
Neo4J Datalog Wrapper

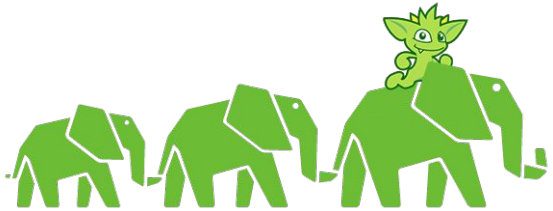
Plan

- Introduction
- Project Structure
- Tools
- Development Process
- Execution Plans
- Issues/Improvements
- Lessons/Future Outlook

1 Introduction

Need for virtualized architecture





Team Members

Julius Remigio

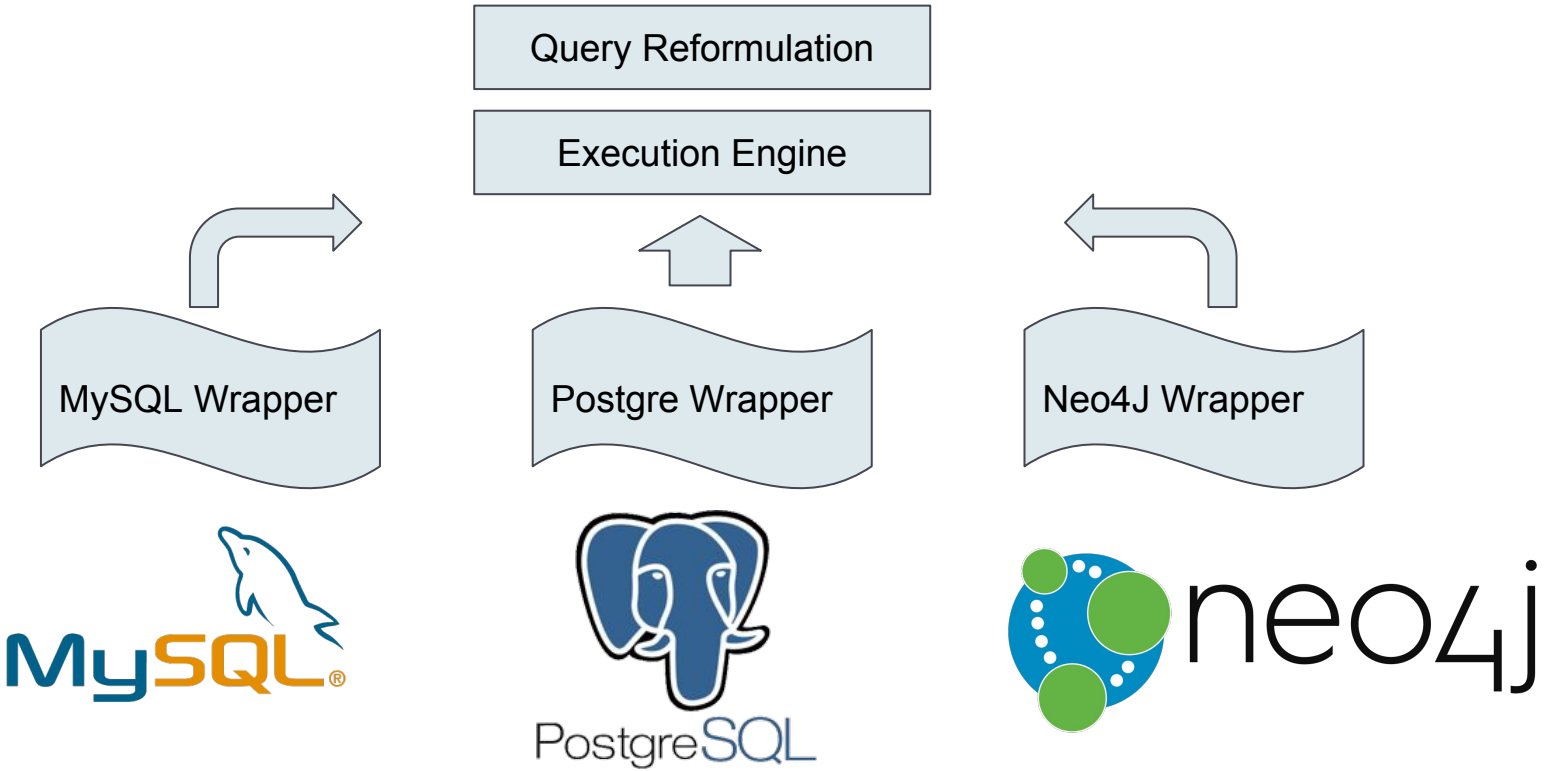
Ryan Riopelle

Michael Galarnyk

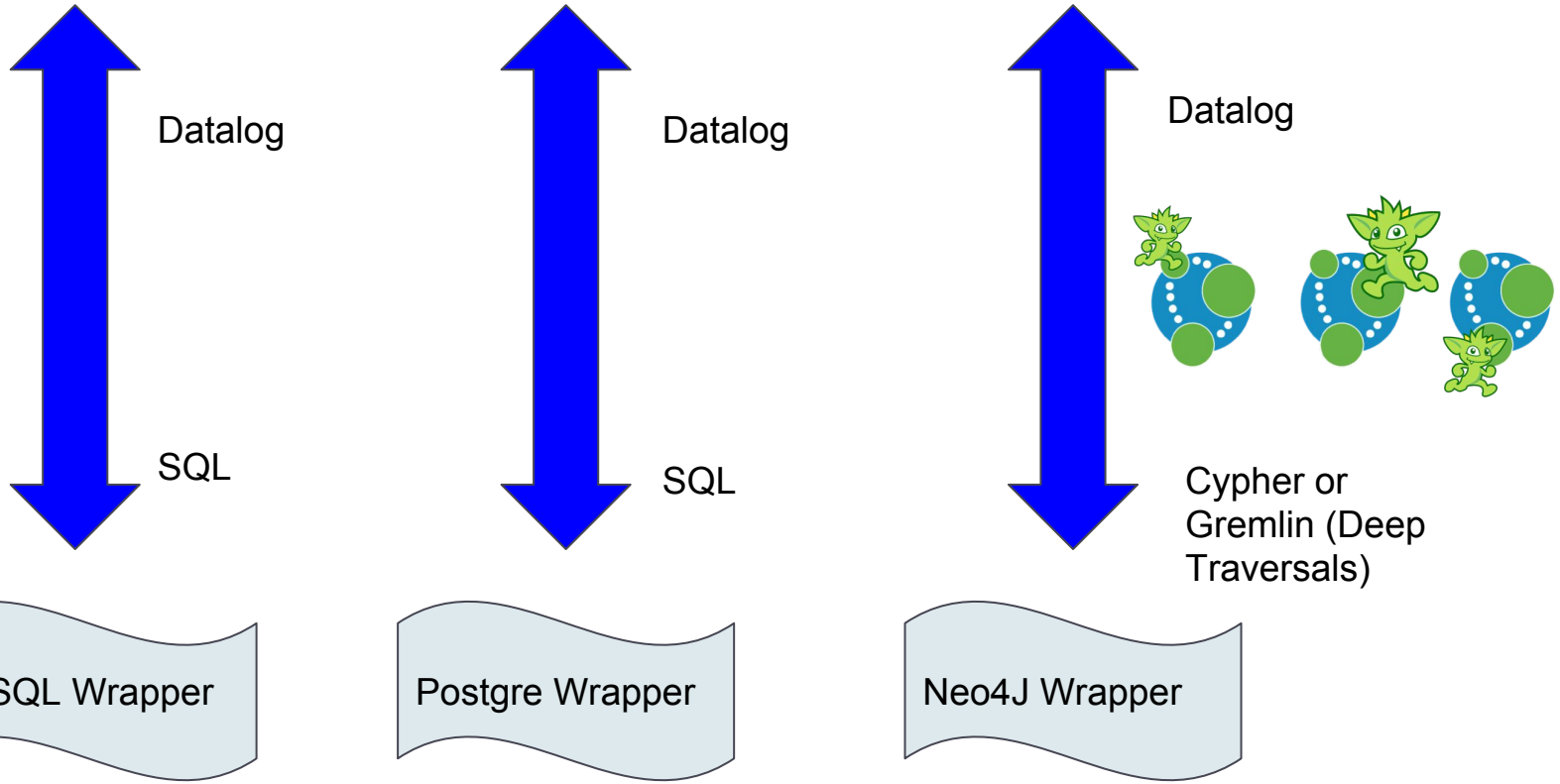
2 Project Structure

Need for virtualized architecture

Virtual Data Integration Architecture



Query Languages

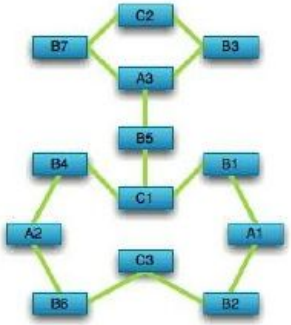


Graph Databases

Compared to Relational Databases

Optimized for aggregation

Optimized for connections



Neo4J Wrapper Process Flow


```
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

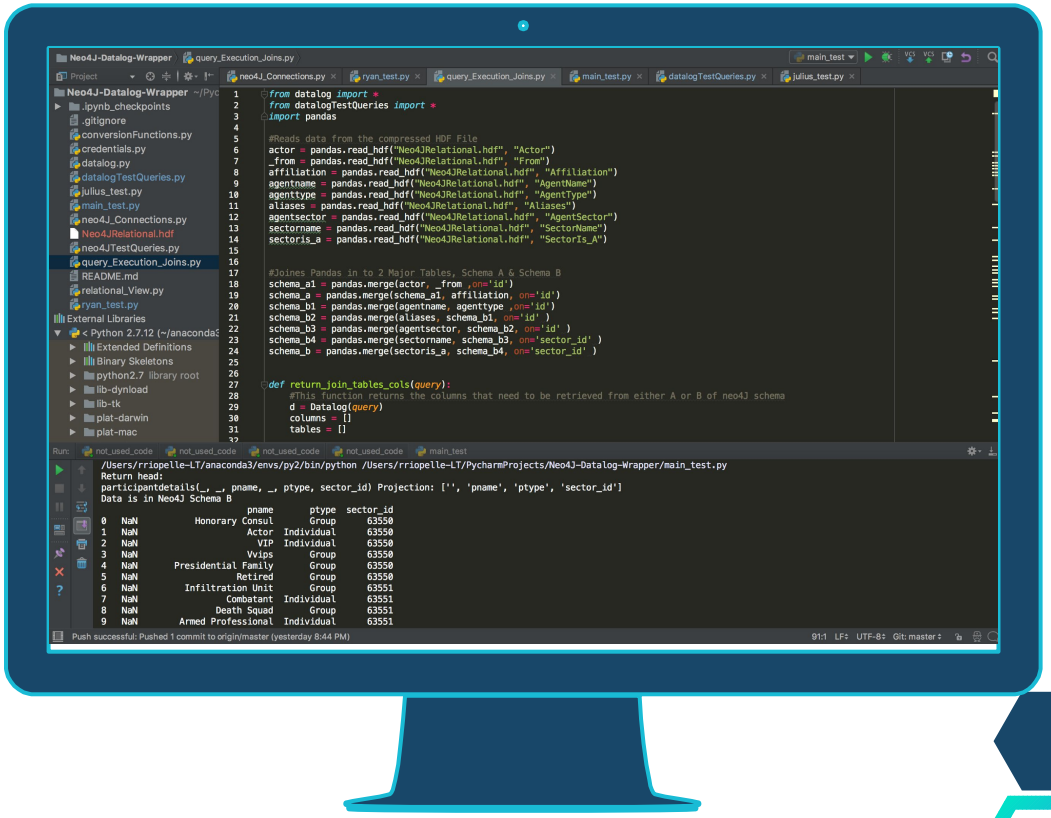
3 Tools

NEED 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.



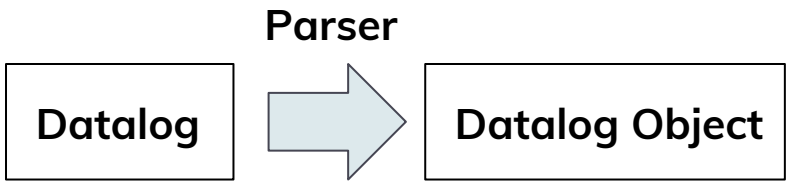
4

Parser

Parse incoming datalog query

Datalog Parser Overview

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



Datalog Components

Datalog Example:

head(a, b, c, d, e) = a(a, b, c), b(c, ‘foo’, e), e > 10

Components of Datalog Query:

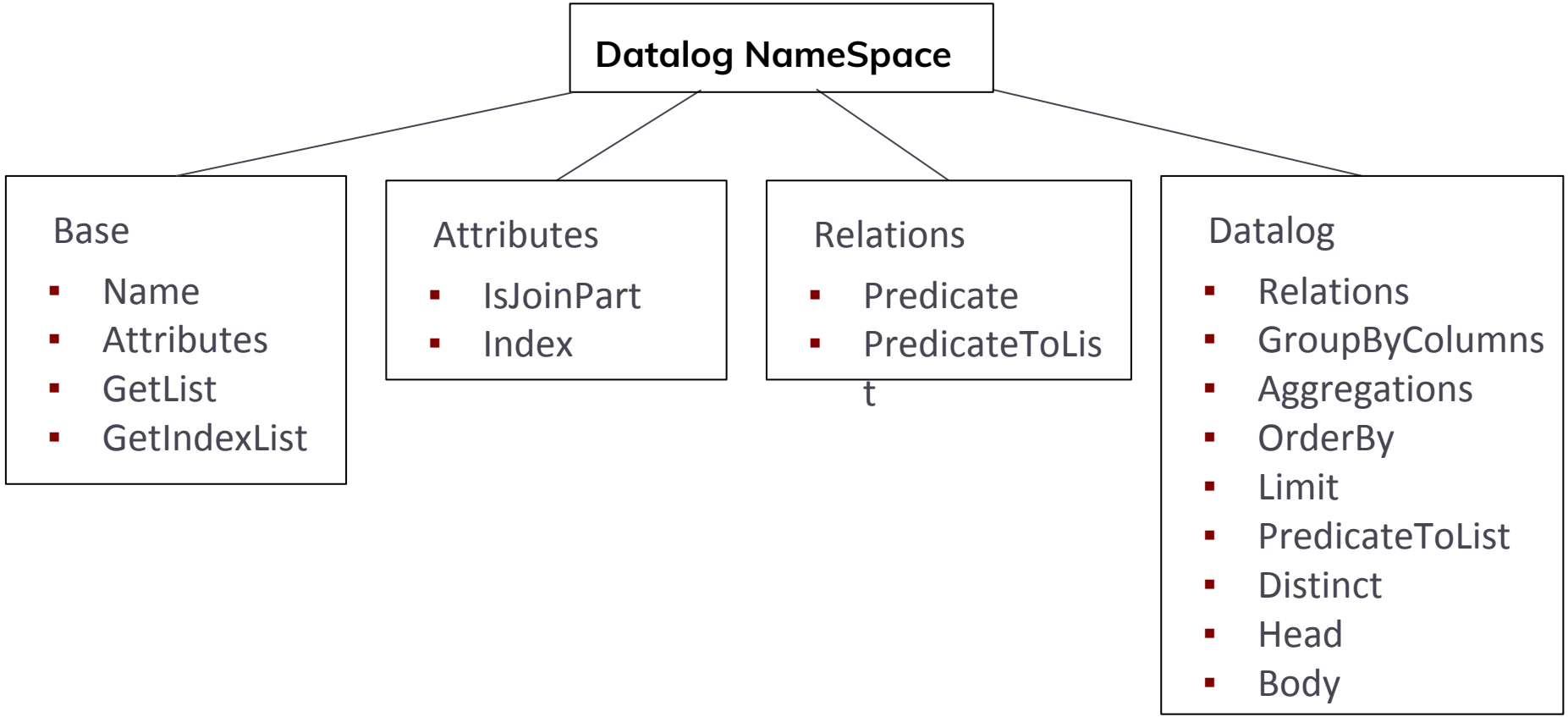
Head, body, relations, attributes, predicates, aggregations,etc.

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



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

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

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

```
predicate: ["g = CONTAINS('foo')"]
   h    2    False
   g    1    False    CONTAINS('foo')
   f    0    True
group by: ['GROUP_BY([a, b], d =
   COUNT(c))']
grouping columns: ['a', 'b']
aggregations: ['COUNT(c) AS d']
having clause: [d < 100]
order by: ORDER BY b DESC
limit: 25
distinct: True
```

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

Issues

- 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

Issue - Nesting 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))
```

Solution - Example

- Standardized syntax makes things easier
  - All functions represented in CAPS
  - All relations represented in lowercase

UPPER() = function *foo() = relation*

- Limiting use of nesting to a few known cases
  - Write special code to identify these cases and handle appropriately

GROUP_BY([x,y], z= COUNT(x))

Solution

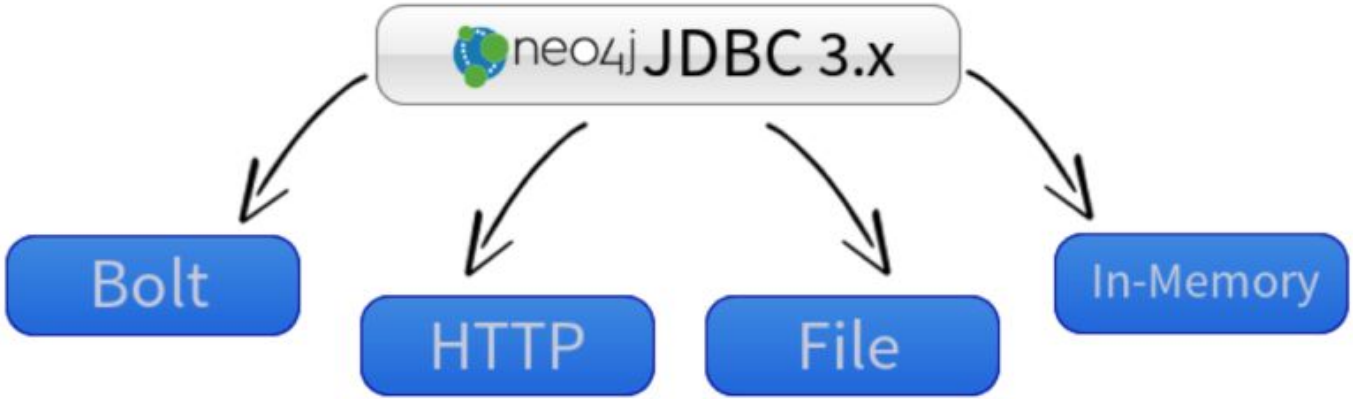
```
Group By Example:
GROUP_BY Pattern = '(GROUP_BY[(].+[)][)])'
Relation Pattern = '([a-z09]+[(].+[)])'
Function Pattern = '([A-Z]+[(].+[)])'
Datalog = 'a(x, y, a, b) :- b(x, y, z),
                GROUP_BY([x, y], a = COUNT(z), b = SUM(Z))'
Matches in body:
> relations: b(x, y, z)
> functions: COUNT(z), SUM(Z)
> group_by:  GROUP_BY([x, y], a = COUNT(z), b = SUM(Z))
>
```

A better approach...

PyParsing - a grammar based approach

5 Execution Plans

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/")
```

Data Retrieval Plans

a) Plan 1
   i) Pull and Save To Relational
   ii) Parser
   iii) Joins Tables
   iv) Implement Predicates Over Joined Tables
   v) Return Dataframe
b) Plan 2
   i) Chain cypher queries
   ii) Predicate pushdown directly Neo4J
   iii) Output any given Neo4J query based on the datalog input
   iv) Output any given number of tables based on input

Retrieving Data In Relational Format

Relational Tables For Schema A

Actor = DataFrame(graph.data("Match (a:Actor) Return distinct ID(a) as id, a.Type as ptype, a.Name as pname, a.AliasList as AliasList"))

From = DataFrame(graph.data("MATCH (a:Actor)-[:From]-(c:Country) RETURN ID(a) as id,c.Name as country"))

Affiliation = DataFrame(graph.data("MATCH (a:Actor)-[r:Affiliation]-(o:Organization) RETURN ID(a) as id,o.Name as org, r.beginDate as start, r.endDate as end"))

Relational Tables For Schema B

AgentName = DataFrame(graph.data("Match (a:AgentName) Return ID(

a) as id, a.Name as pname"))

AgentType = DataFrame(graph.data("Match (a_n:AgentName)-[:AgentType]-(a_t:AgentType) Return ID(a_n) as id, a_t.Name as ptype"))

Aliases = DataFrame(graph.data("Match (a_n:AgentName)-[:AgentAlias]-(a:Aliases) Return ID(a_n) as id, a.AliasList as alias"))

AgentSector = DataFrame(graph.data("Match (a:AgentName)-[:Sector]-(s:Sector) Return ID(a) as id, ID(s) as sector_id"))

SectorName = DataFrame(graph.data("Match (s:Sector) Return ID(s) as sector_id, s.Name as name"))

SectorIs_A = DataFrame(graph.data("MATCH p=(s1:Sector)-[:`is-a`]->(s2:Sector) RETURN ID(s1) as sector_id, ID(s2) as sector_id2"))

Cypher Converted to Relational View

Cypher Relational View

Schema A

- Actor: [u'AliasList' u'id' u'pname' u'ptype']
- From: [u'country' u'id']
- Affiliation: [u'end' u'id' u'org' u'start']

Schema B

- 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']

Return Format

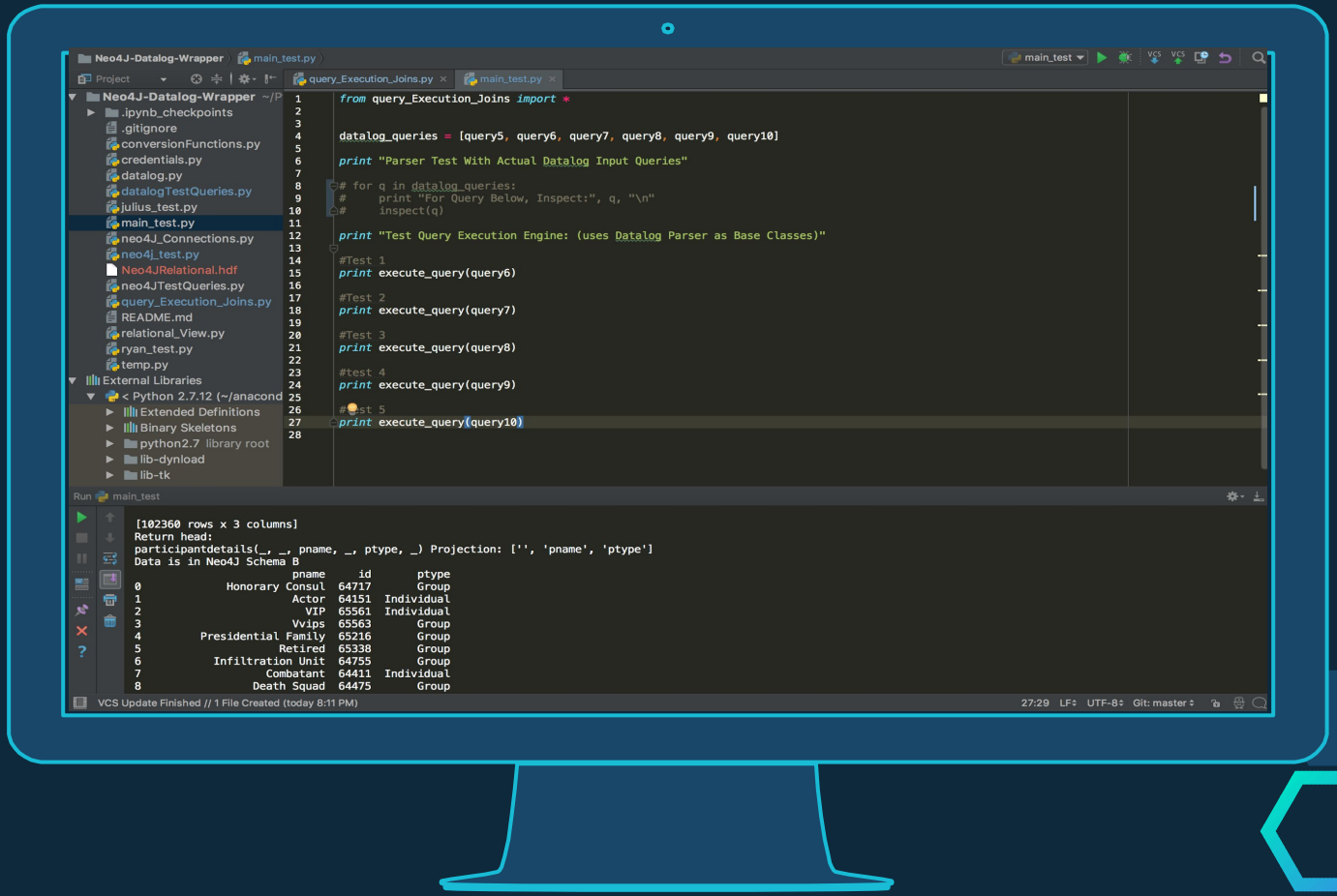
- ParticipantsGlobalSchema = ['_0', 'id', '_2', 'ptype', 'pname', 'sector_id']
- ParticipantDetailGlobalSchema = ['_0', 'Category', 'org', 'country', 'name', '_5']
- Events = ['Date']

Query Execution Classes

Execution consists of a set of classes

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

Show Working Code



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(_, name)

Map: <relation>⟶<node/edge>

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

agentname(id, pname)⟶**ID(<n>AgentName)**, <n>Organization.Name

Cypher:

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

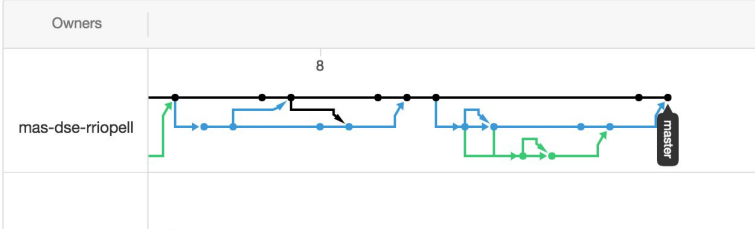
6

Lessons/Future Considerations

Takeaways

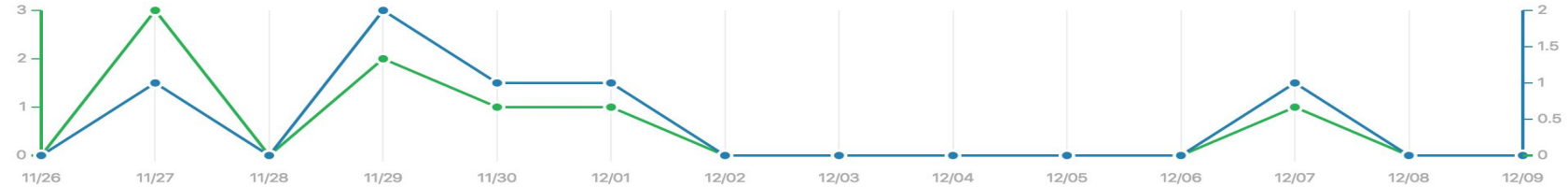
- KISS
- Finding the right approach early is critical
- Get source control working early
- Don’t underestimate complexity and effort
- Communication is important
- Integration is hard

Github Stats



Contributors | **Traffic** | Commits | Code frequency | Punch card | Network

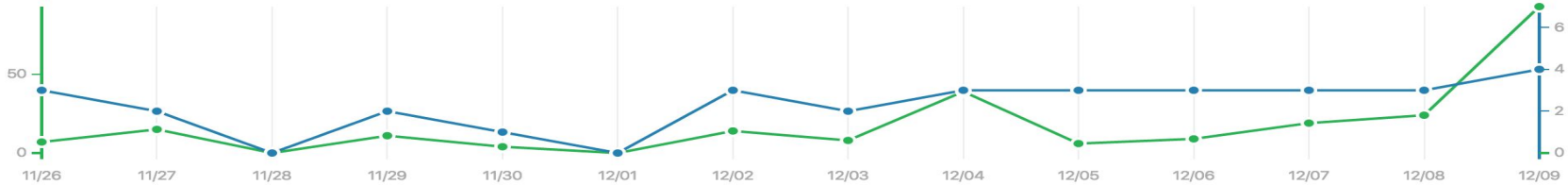
Git clones



8
Clones

4
Unique cloners

Visitors



249
Views

10
Unique visitors

Thanks!

Any questions?