Embarrassingly parallel database calls with Python

Niels Zeilemaker / Giovanni Lanzani
Big Data Hacker / Data Whisperer

@gglanzani
{nielszeilemaker,giovannilanzani}@godatadriven.com

GoDataDriven

PROUDLY PART OF THE XEBIA GROUP

Who are we

Background:

PhD Computer Science / Theoretical Physics

Now

GoDataDriven

Why embarrassingly parallel?

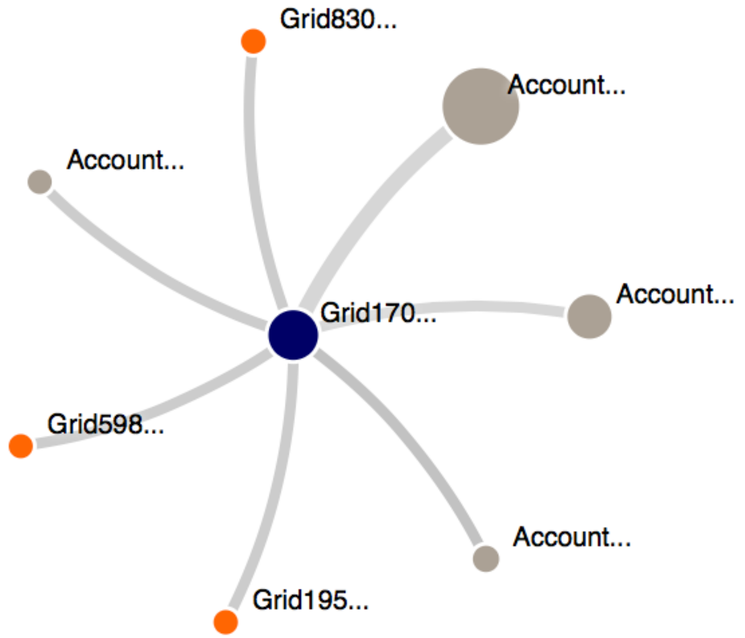
- No store and retrieve;
- Store, {transform, enrich, analyse} and then *retrieve*;
- Real-time: *retrieve* is not a batch process.

Retrieve network of businesses

Show their structure

Network

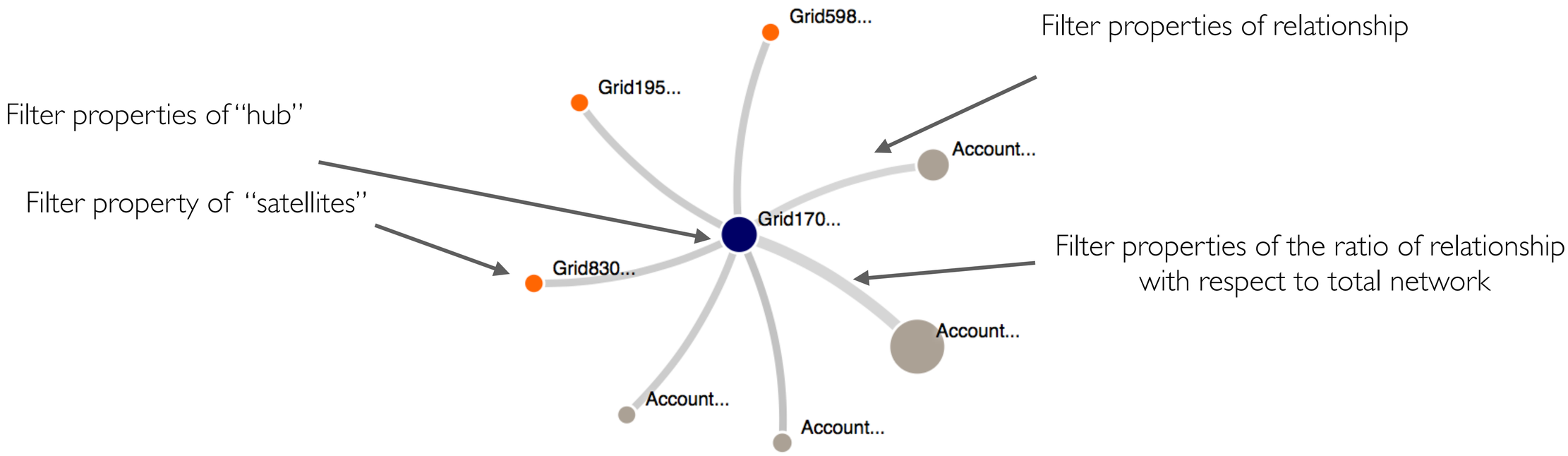
- Buyer
- Known Supplier
- Unknown Supplier



Circle size: incoming amount relative to other nodes
Line transparency: number of transfers
Line width: amount transferred between accounts

Challenges

- Relational data model seems to be the best… but:



Challenges

- That is up to 13 filters:
  - 11 JOIN's (with tables ranging from 300k to 15M records);
  - 9 WHERE's and 3 HAVING's;
  - 1 windowing function;
  - 4 CASE's;
  - (And 1 store procedure generating materialized views).

Example query

Data structure

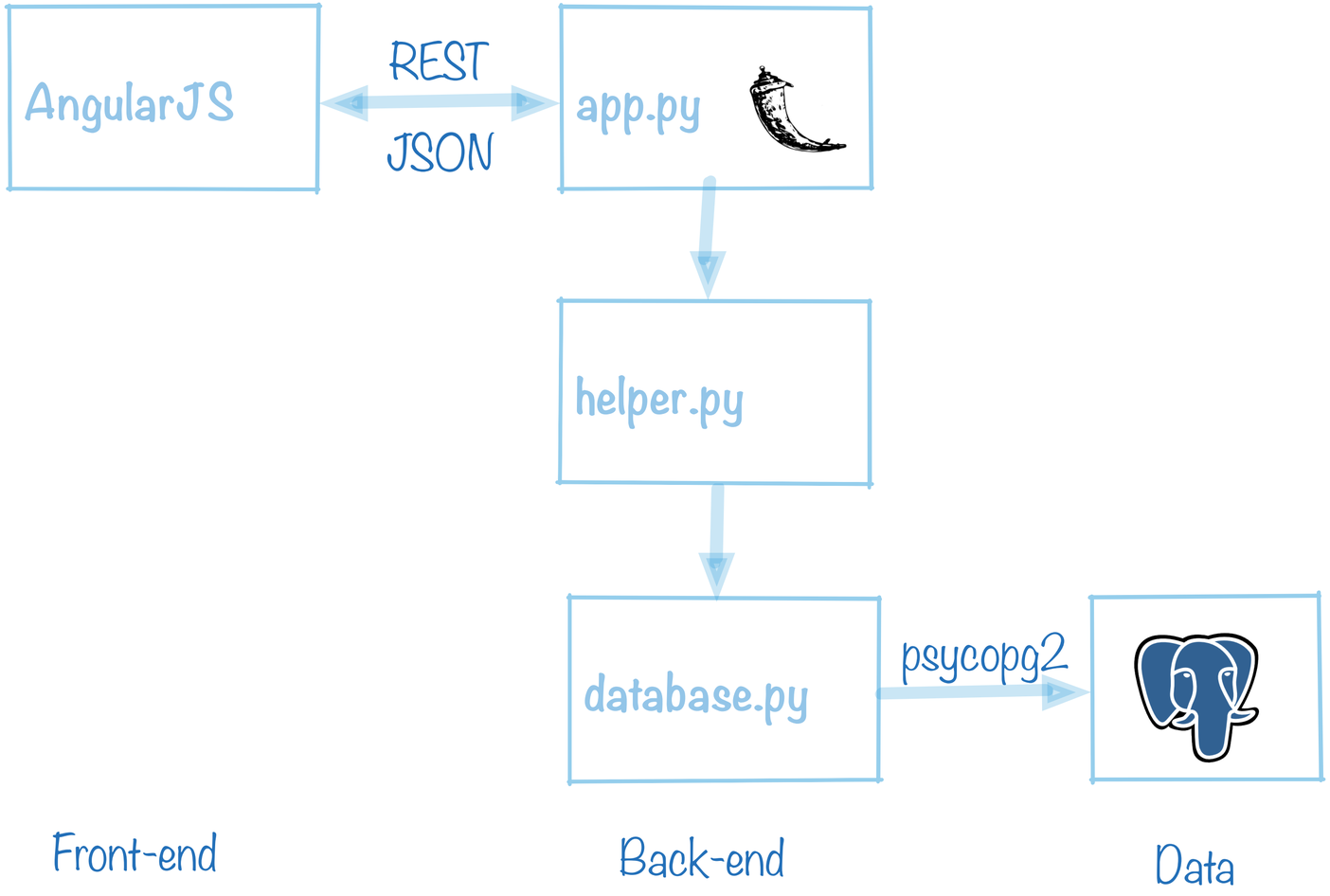
- Single large table contains all interactions between companies

Date	Payer	Beneficiary	Amount	#Transactions
2015-01	GDD	PyData	100	1

First question: which database?

- Postgresql
  - Window function, WITH, functional/partial indexes, open source;
- With the right indexes: 3s per query.

Architecture



JS- I

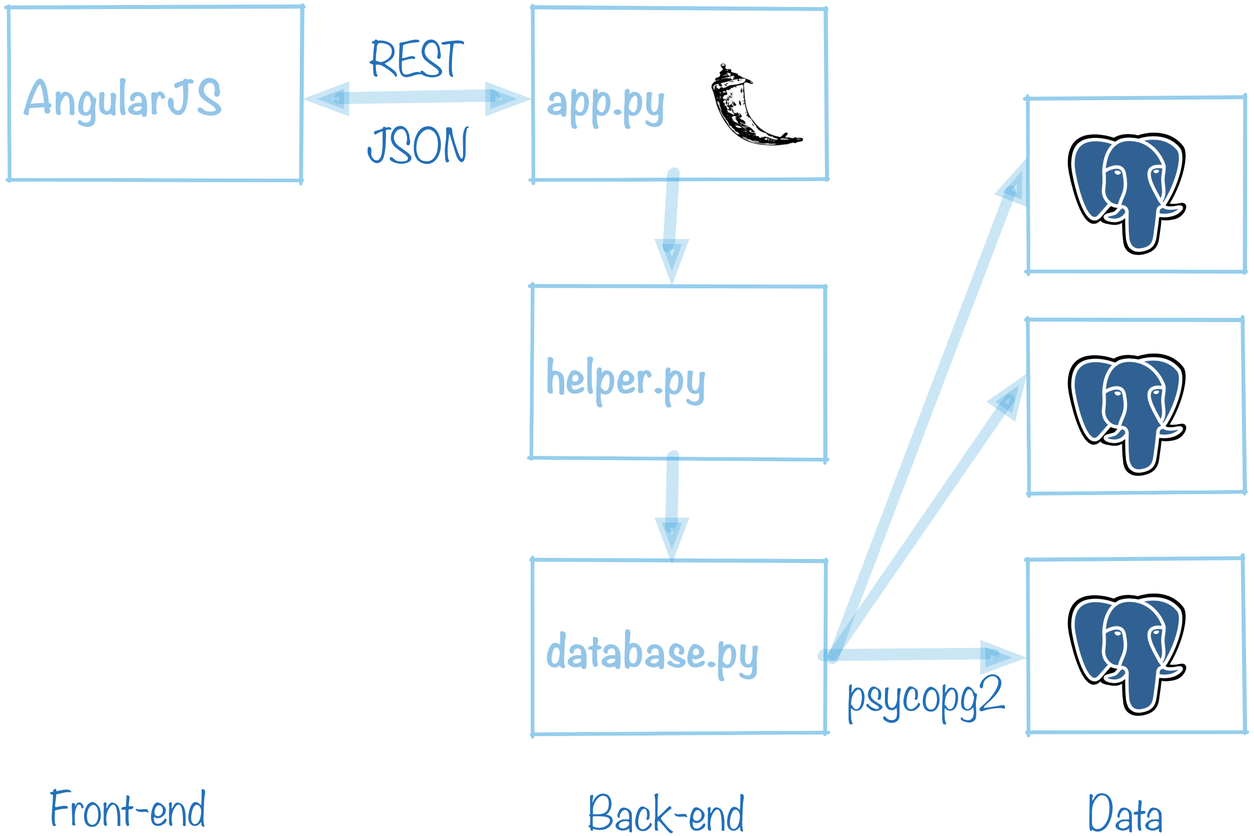
JS-2

Scaling issues: app is not realtime

- Load balacing does not reduce (single) query runtime
- Sharding makes queries faster if the shard key is in a where clause
- Our use case requires us to query all data from either
  - the payer
  - the beneficiary
- Traditional sharding will not cut it

New architecture

- Instead, let's run the queries in parallel across sharded instances and merge the result in python



New Data structure

Date	Sharded-Payer	Beneficiary	Amount	#Transactions
2015-01	GDD	PyData	100	1
2015-01	GDD	Xebia	15	3

Date	Sharded-Payer	Beneficiary	Amount	#Transactions
2015-01	PyData	GDD	100	1
2015-03	PyData	Xebia	20	2

Date	Sharded-Payer	Beneficiary	Amount	#Transactions
2015-02	Xebia	GDD	100	1
2015-03	Xebia	PyData	20	2

Old code (single database)

```
pool = ThreadedConnectionPool(1, 20, dsn=d)
connection = pool.getconn()
cursor = connection.cursor()
cursor.execute(my_query)
cursor.fetchall()
```

New code (multiple databases)

```
pools = [ThreadedConnectionPool(1, 20, dsn=d) for d in dsns]
connections = [pool.getconn() for pool in pools]
parallel_connection = ParallelConnection(connections)
cursor = parallel_connection.cursor()
cursor.execute(my_query)
cursor.fetchall()
```

parallel_connection.py

```
from threading import Thread
class ParallelConnection(object):
    """
       This class manages multiple database connections, handles the parallel access to it, and
       hides the complexity this entails. The execution of queries is distributed by running it
       for each connection in parallel. The result (as retrieved by fetchall() and fetchone())
       is the union of the parallelized query results from each connection.
    """

    def __init__(self, connections):
        self.connections = connections
        self.cursors = None
```

parallel_connection.py

```
def execute(self, query, tuple_args=None, fetchnone=False):
    self._do_parallel(lambda i,c: c.execute(query, tuple_args))

def _do_parallel(self, target):
    threads = []
    for i, c in enumerate(self.cursors):
        t = Thread(target=lambda i=i, c=c: target(i,c))
        t.setDaemon(True)
        t.start()
        threads.append(t)

    for t in threads:
        t.join()
```

parallel_connection.py

```
def fetchone(self):
    results = [None] * len(self.cursors)
    def do_work(index, cursor):
        results[index] = cursor.fetchone()
    self._do_parallel(do_work)

    results_values = filter(is_not_none, results)
    if results_values:
        return list(chain(results_values))[0]

def fetchall(self):
    results = [None] * len(self.cursors)
    def do_work(index, cursor):
        results[index] = cursor.fetchall()
    self._do_parallel(do_work)

    return list(chain(*[rs for rs in results]))
```

Unsharded tables?

- They are present in every Postgres instance
- Space is not an issue nowadays

Results

- Queries on sharded tables execute in 1/N, where N is the number of Postgres instances;
- Plus some neglibigle thread overhead
- Our results, using 3 servers 1.04s instead of 3.0s

Update/Inserts

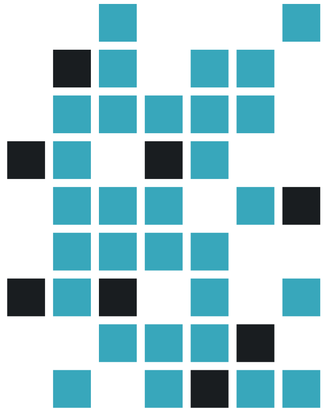
- Short anwser, not supported
- parallel_connection.py does not know of the existence of the shards
  - It simply executes a single query multiple times
- In order to support updates and inserts, a sharded insert/insert all needs to be implented
  - (PR are welcome)

Our insert approach

- Load data in batches, coordinated with ansible
- To determine the shard, we compute the hash + modulo in hive

Where can I get it

- https://github.com/godatadriven/parallel-connection



GoDataDriven

We're hiring / Questions? / Thank you!

Niels Zeilemaker / Giovanni Lanzani
Big Data Hacker / Data Whisperer

@gglanzani
{nielszeilemaker,giovannilanzani}@godatadriven.com