

# 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

[illegible]

Item	Category	Rating	Number of Ratings	# Suppliers	Min Price	Max Price	Total Spend
Item 1	Category 1	5.0	10	5	\$100	\$200	\$1,000
Item 2	Category 2	4.5	20	10	\$150	\$300	\$2,000
Item 3	Category 3	4.0	30	15	\$200	\$400	\$3,000
Item 4	Category 4	3.5	40	20	\$250	\$500	\$4,000
Item 5	Category 5	3.0	50	25	\$300	\$600	\$5,000
Total			150	75			\$15,000

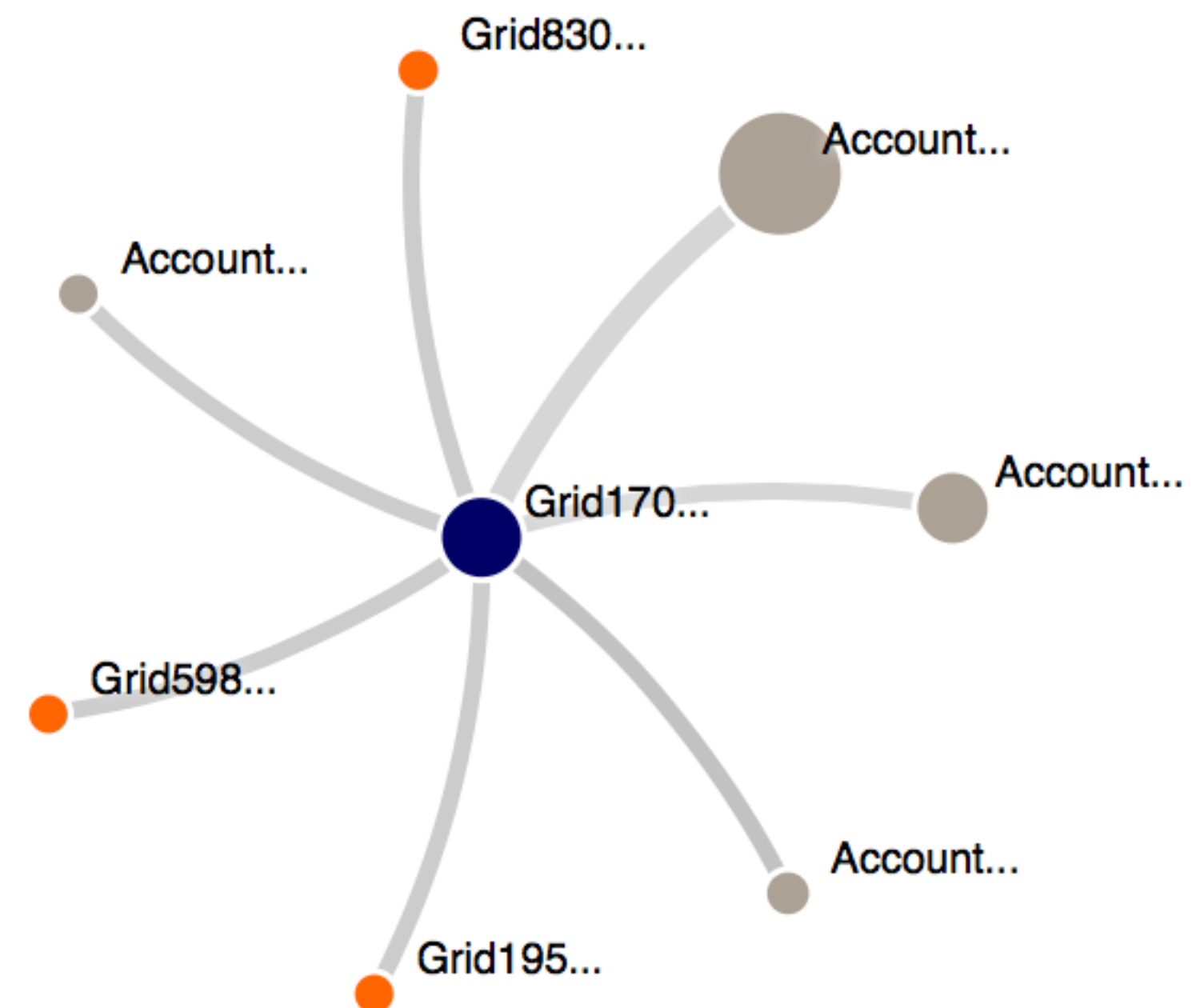
Page 1 of 1



# Show their structure

## Network

- Buyer
- Known Supplier
- Unknown Supplier



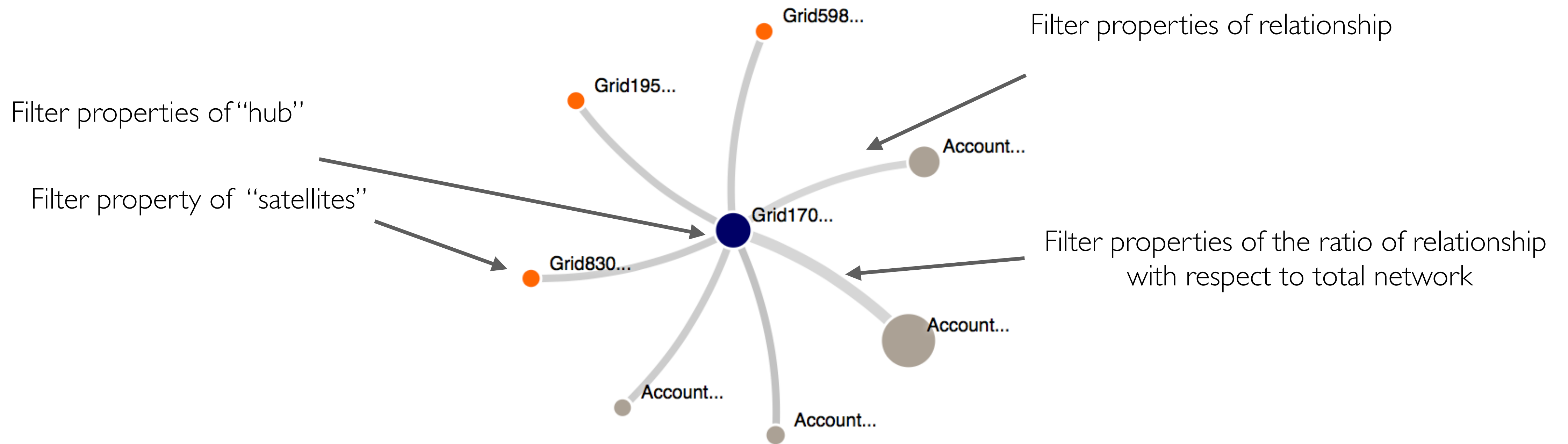
Circle size: incoming amount relative to other nodes  
Line transparency: number of transfers  
Line width: amount transferred between accounts





# Challenges

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



...and filter properties of the total network!



# 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

[illegible][illegible]

```

440  cat <<<<<<<<<<
4410  (ls -la /usr/bin | grep -E '^d' | wc -l)
4420  1
4430  1
4440  1
4450  1
4460  1
4470  1
4480  1
4490  1
4500  1
4510  1
4520  1
4530  1
4540  1
4550  1
4560  1
4570  1
4580  1
4590  1
4600  1
4610  1
4620  1
4630  1
4640  1
4650  1
4660  1
4670  1
4680  1
4690  1
4700  1
4710  1
4720  1
4730  1
4740  1
4750  1
4760  1
4770  1
4780  1
4790  1
4800  1
4810  1
4820  1
4830  1
4840  1
4850  1
4860  1
4870  1
4880  1
4890  1
4900  1
4910  1
4920  1
4930  1
4940  1
4950  1
4960  1
4970  1
4980  1
4990  1
5000  1
5010  1
5020  1
5030  1
5040  1
5050  1
5060  1
5070  1
5080  1
5090  1
5100  1
5110  1
5120  1
5130  1
5140  1
5150  1
5160  1
5170  1
5180  1
5190  1
5200  1
5210  1
5220  1
5230  1
5240  1
5250  1
5260  1
5270  1
5280  1
5290  1
5300  1
5310  1
5320  1
5330  1
5340  1
5350  1
5360  1
5370  1
5380  1
5390  1
5400  1
5410  1
5420  1
5430  1
5440  1
5450  1
5460  1
5470  1
5480  1
5490  1
5500  1
5510  1
5520  1
5530  1
5540  1
5550  1
5560  1
5570  1
5580  1
5590  1
5600  1
5610  1
5620  1
5630  1
5640  1
5650  1
5660  1
5670  1
5680  1
5690  1
5700  1
5710  1
5720  1
5730  1
5740  1
5750  1
5760  1
5770  1
5780  1
5790  1
5800  1
5810  1
5820  1
5830  1
5840  1
5850  1
5860  1
5870  1
5880  1
5890  1
5900  1
5910  1
5920  1
5930  1
5940  1
5950  1
5960  1
5970  1
5980  1
5990  1
6000  1
6010  1
6020  1
6030  1
6040  1
6050  1
6060  1
6070  1
6080  1
6090  1
6100  1
6110  1
6120  1
6130  1
6140  1
6150  1
6160  1
6170  1
6180  1
6190  1
6200  1
6210  1
6220  1
6230  1
6240  1
6250  1
6260  1
6270  1
6280  1
6290  1
6300  1
6310  1
6320  1
6330  1
6340  1
6350  1
6360  1
6370  1
6380  1
6390  1
6400  1
6410  1
6420  1
6430  1
6440  1
6450  1
6460  1
6470  1
6480  1
6490  1
6500  1
6510  1
6520  1
6530  1
6540  1
6550  1
6560  1
6570  1
6580  1
6590  1
6600  1
6610  1
6620  1
6630  1
6640  1
6650  1
6660  1
6670  1
6680  1
6690  1
6700  1
6710  1
6720  1
6730  1
6740  1
6750  1
6760  1
6770  1
6780  1
6790  1
6800  1
6810  1
6820  1
6830  1
6840  1
6850  1
6860  1
6870  1
6880  1
6890  1
6900  1
6910  1
6920  1
6930  1
6940  1
6950  1
6960  1
6970  1
6980  1
6990  1
7000  1
7010  1
7020  1
7030  1
7040  1
7050  1
7060  1
7070  1
7080  1
7090  1
7100  1
7110  1
7120  1
7130  1
7140  1
7150  1
7160  1
7170  1
7180  1
7190  1
7200  1
7210  1
7220  1
7230  1
7240  1
7250  1
7260  1
7270  1
7280  1
7290  1
7300  1
7310  1
7320  1
7330  1
7340  1
7350  1
7360  1
7370  1
7380  1
7390  1
7400  1
7410  1
7420  1
7430  1
7440  1
7450  1
7460  1
7470  1
7480  1
7490  1
7500  1
7510  1
7520  1
7530  1
7540  1
7550  1
7560  1
7570  1
7580  1
7590  1
7600  1
7610  1
7620  1
7630  1
7640  1
7650  1
7660  1
7670  1
7680  1
7690  1
7700  1
7710  1
7720  1
7730  1
7740  1
7750  1
7760  1
7770  1
7780  1
7790  1
7800  1
7810  1
7820  1
7830  1
7840  1
7850  1
7860  1
7870  1
7880  1
7890  1
7900  1
7910  1
7920  1
7930  1
7940  1
7950  1
7960  1
7970  1
7980  1
7990  1
8000  1
8010  1
8020  1
8030  1
8040  1
8050  1
8060  1
8070  1
8080  1
8090  1
8100  1
8110  1
8120  1
8130  1
8140  1
8150  1
8160  1
8170  1
8180  1
8190  1
8200  1
8210  1
8220  1
8230  1
8240  1
8250  1
8260  1
8270  1
8280  1
8290  1
8300  1
8310  1
8320  1
8330  1
8340  1
8350  1
8360  1
8370  1
8380  1
8390  1
8400  1
8410  1
8420  1
8430  1
8440  1
8450  1
8460  1
8470  1
8480  1
8490  1
8500  1
8510  1
8520  1
8530  1
8540  1
8550  1
8560  1
8570  1
8580  1
8590  1
8600  1
8610  1
8620  1
8630  1
8640  1
8650  1
8660  1
8670  1
8680  1
8690  1
8700  1
8710  1
8720  1
8730  1
8740  1
8750  1
8760  1
8770  1
8780  1
8790  1
8800  1
8810  1
8820  1
8830  1
8840  1
8850  1
8860  1
8870  1
8880  1
8890  1
8900  1
8910  1
8920  1
8930  1
8940  1
8950  1
8960  1
8970  1
8980  1
8990  1
9000  1
9010  1
9020  1
9030  1
9040  1
9050  1
9060  1
9070  1
9080  1
9090  1
9100  1
9110  1
9120  1
9130  1
9140  1
9150  1
9160  1
9170  1
9180  1
9190  1
9200  1
9210  1
9220  1
9230  1
9240  1
9250  1
9260  1
9270  1
9280  1
9290  1
9300  1
9310  1
9320  1
9330  1
9340  1
9350  1
9360  1
9370  1
9380  1
9390  1
9400  1
9410  1
9420  1
9430  1
9440  1
9450  1
9460  1
9
```





# Data structure

- Single large table contains all interactions between companies

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

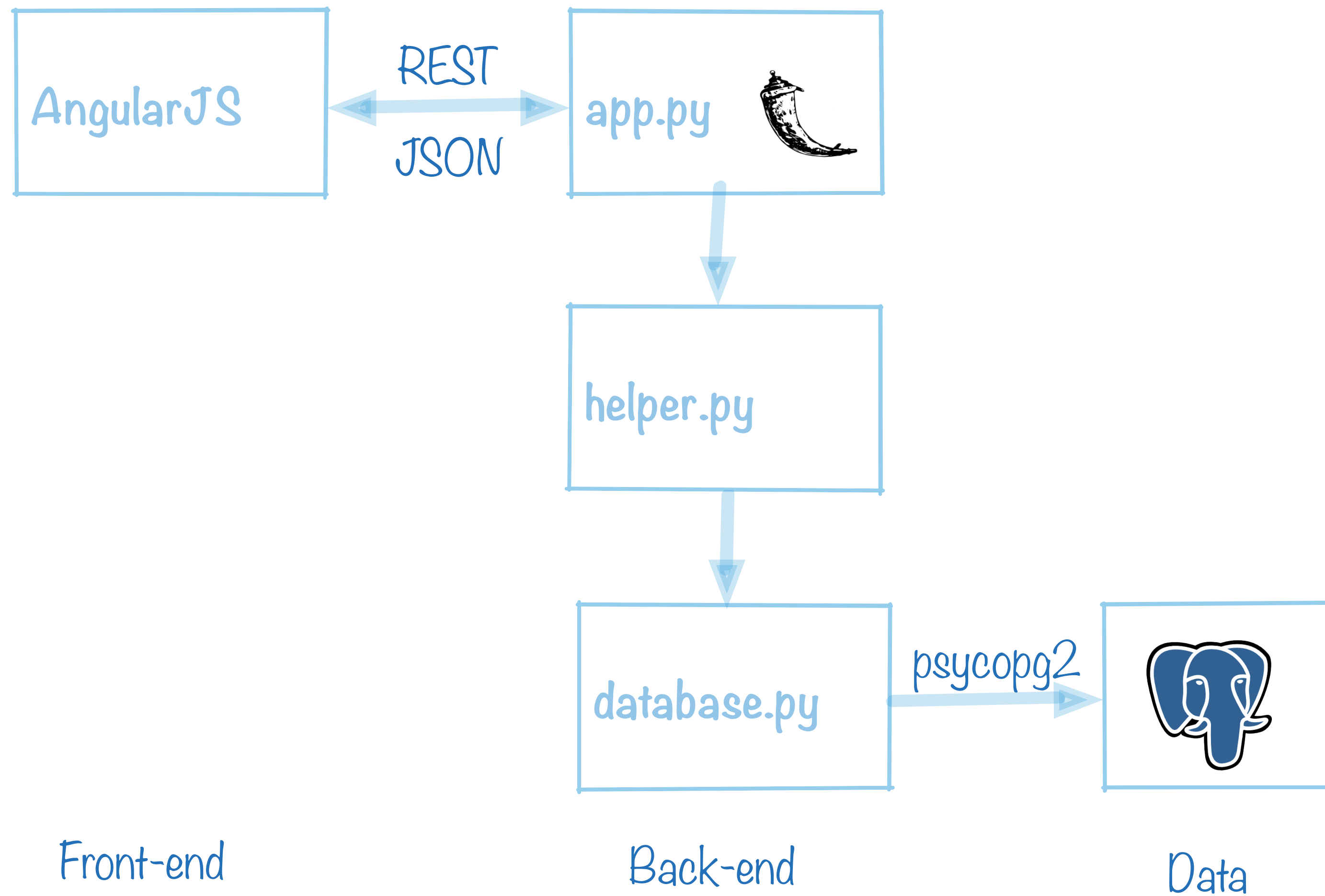


# First question: which database?

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



# Architecture



JS- I



JS-2





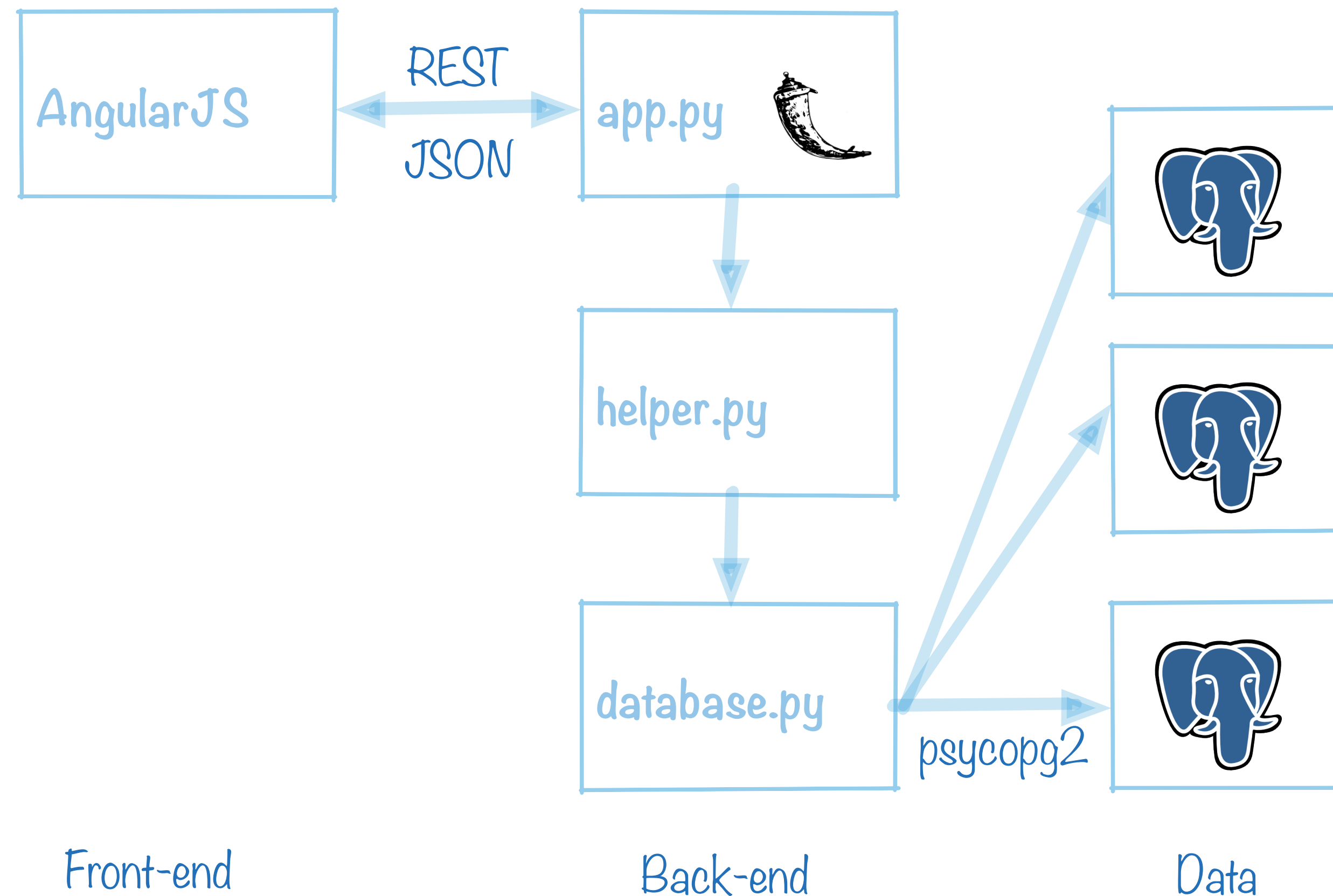
# Scaling issues: app is not realtime

- 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 negligible thread overhead
- Our results, using 3 servers 1.04s instead of 3.0s





# Update/Inserts

- Short answer, 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 implemented
  - (PR are welcome)



# Our insert approach

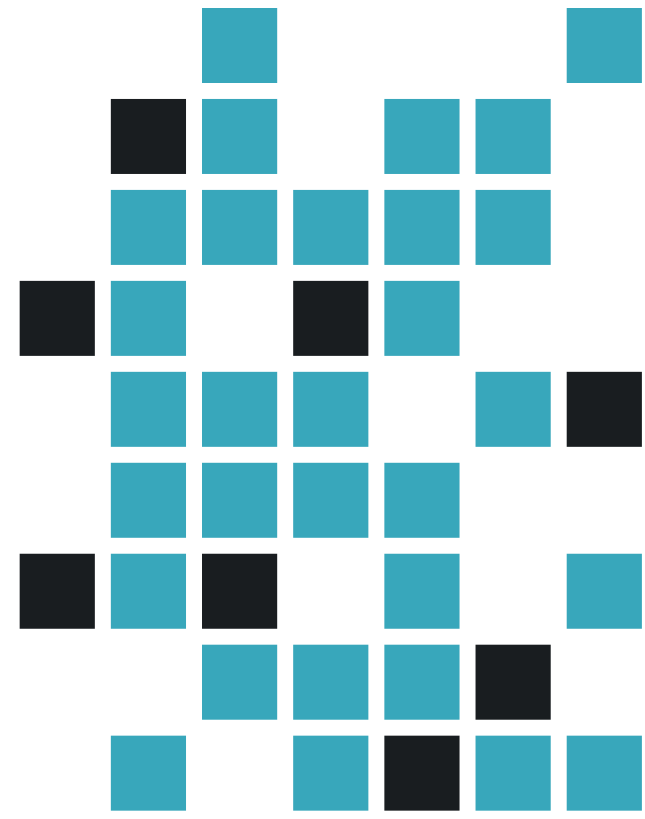
- Load data in batches, coordinated with ansible
- To determine the shard, we compute the hash + modulo in hive



# 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*