This is a story about
our participation in
one of our customer's
project

Scala SWAT

TACKLING A 1 BILLION MEMBER SOCIAL NETWORK

Artur Bańkowski
artur@evojam.com
@abankowski

evojam

SCALAR
Scala Conference in Central Europe

Previous Experience

Datasets I have previously worked on were much smaller!

~10 millions records

~60 millions documents

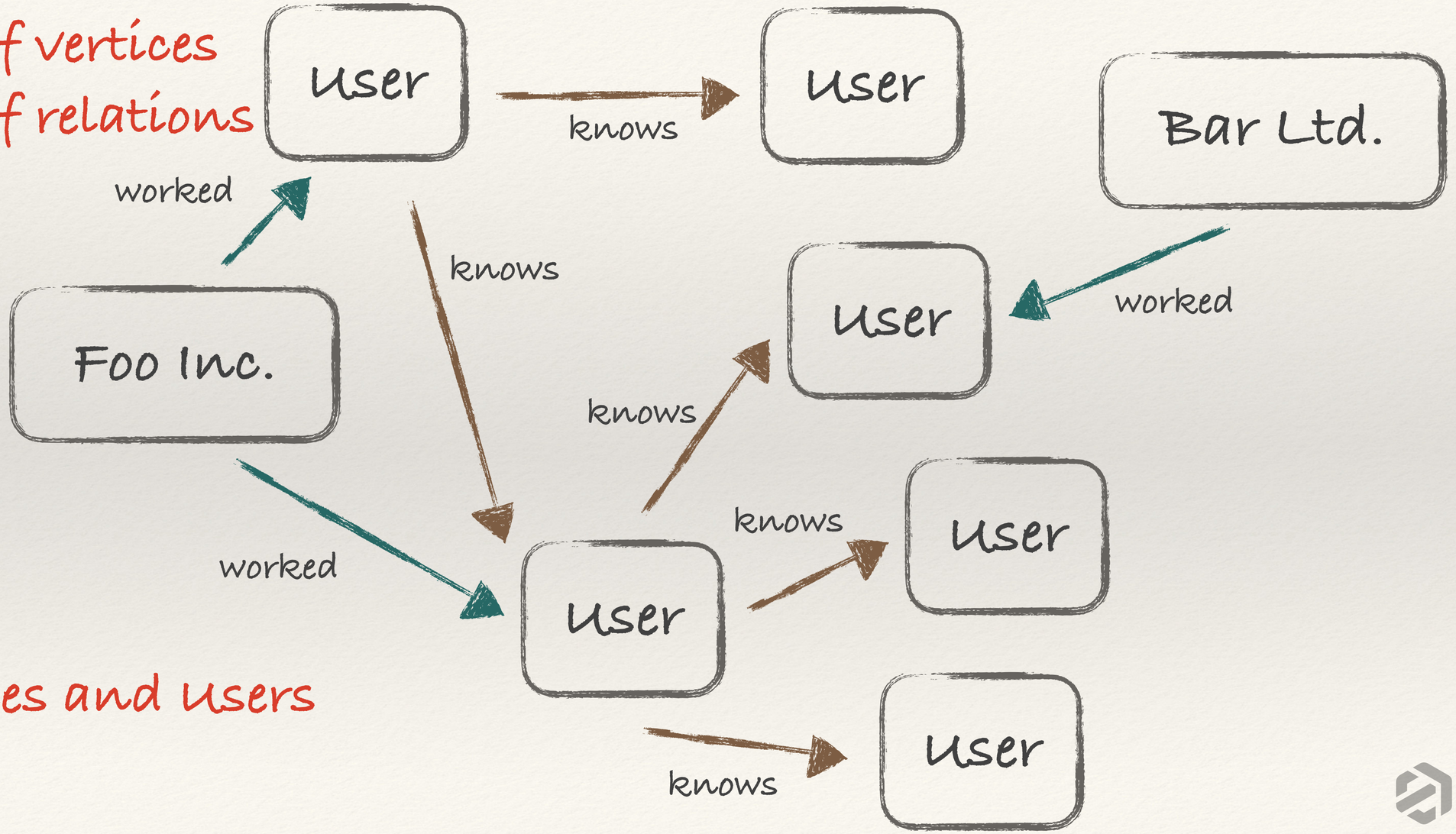
~60 million vertices graph (non-commercial)

This time we were about to deal with a much bigger social network, which can easily be represented as a graph. We were going to join the team…

evojam

Graph structure

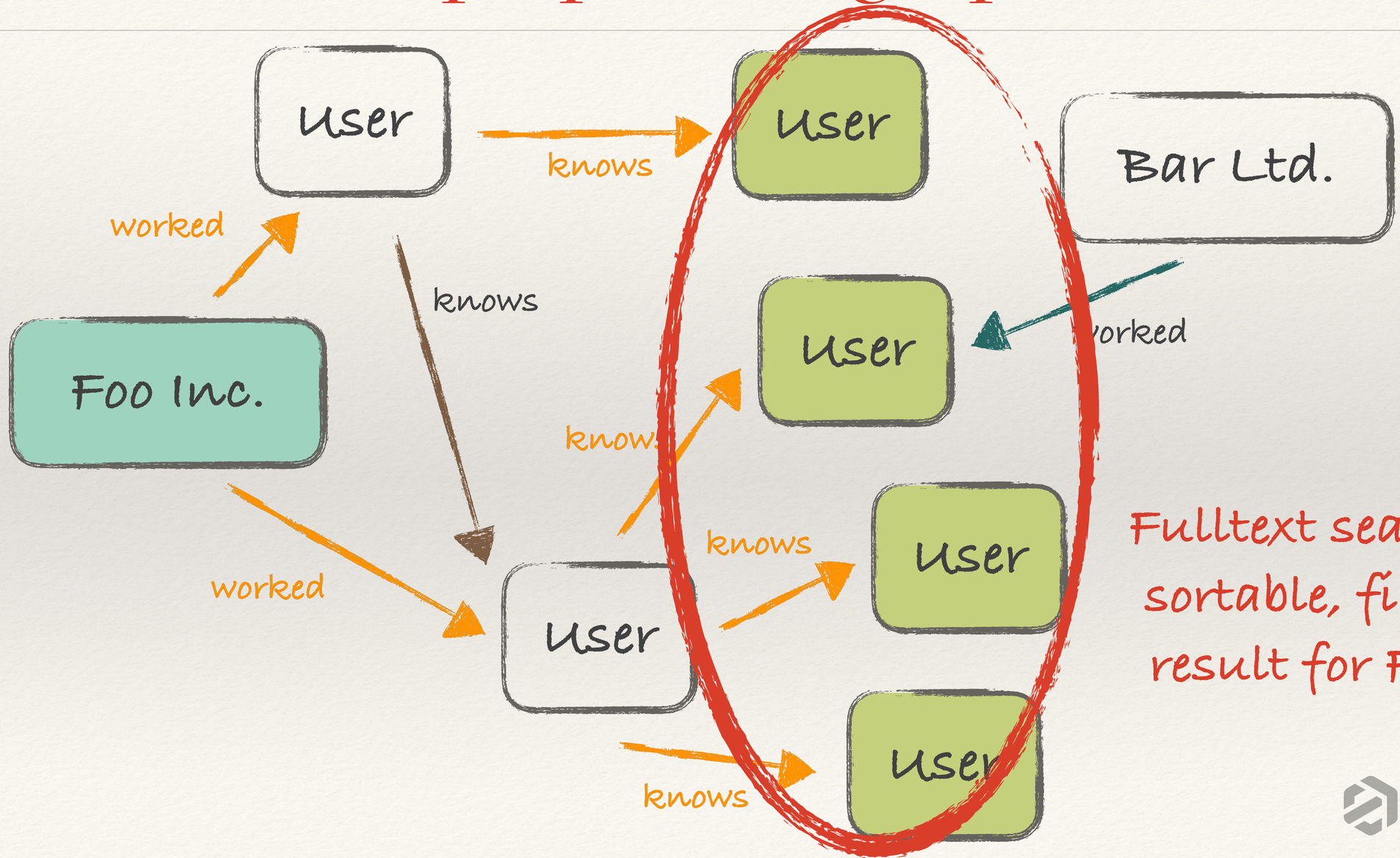
2 types of vertices
2 types of relations



Companies and Users

evojam

The concept: provide graph subset



Fulltext searchable, sortable, filterable result for Foo Inc.

evojam

835,272,759

~~1 billion~~ challenge

When we finally put our hands on the data, we have realized that datasize is smaller than expected

835,272,759 vertices

751,857,081 users

83,415,678 companies

6,956,990,209 relations

What more have we found?

Subsets from thousands to millions users

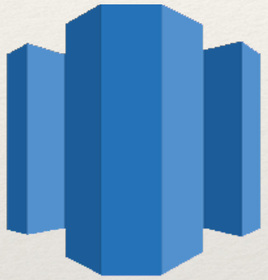
evojam

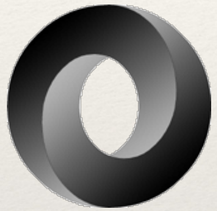
Existing app workflow

This was already used by final customers

extract subset
eg: 3m profiles

data duplication





JSON files

750m profiles

only manually selected
60m incl. duplicates

Manual process,
takes few days from
user perspective

One Engineer-Evening
Multiple custom scripts

evojam

PoC - The Goal

Handle *1 billion* profiles **automatically**

Our primary goal

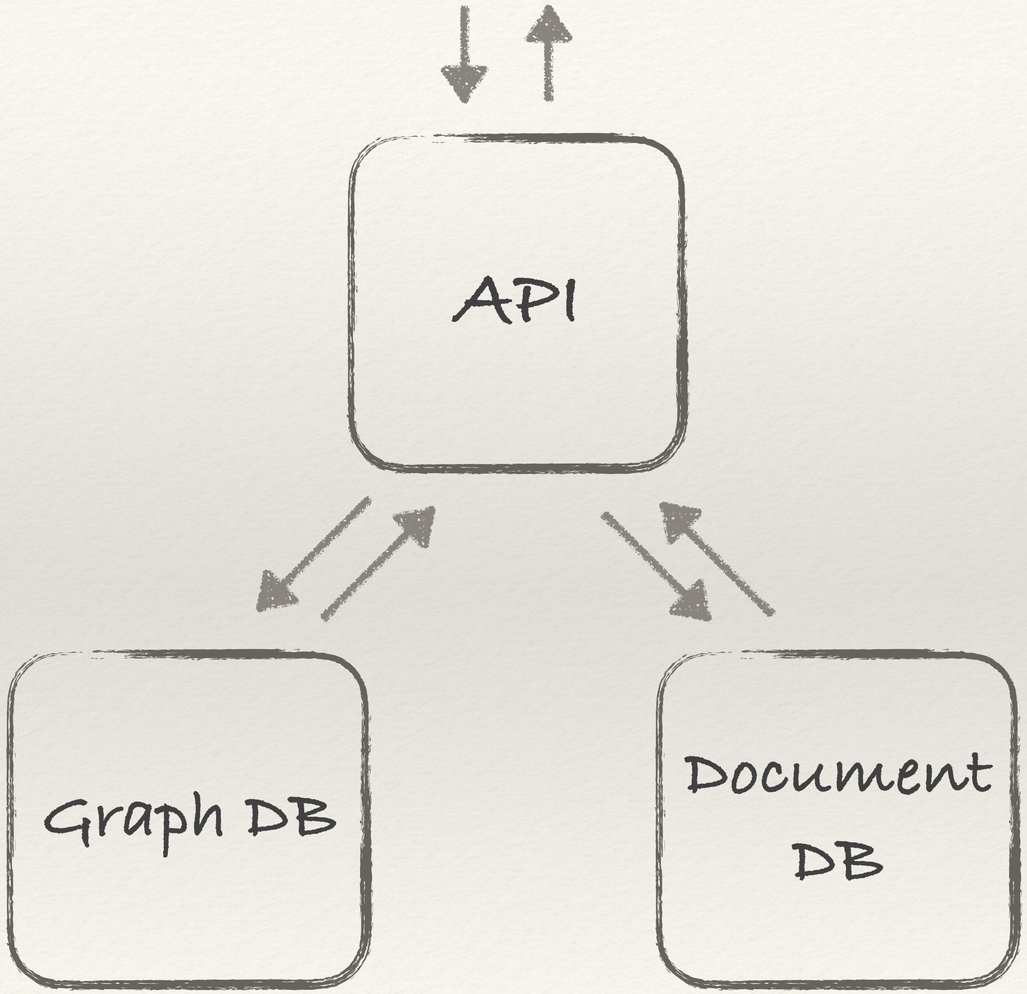
PoC - Definition of done

REST API with search

- Graph traversable on demand
- First results available under 1 minute
- Entire subset ready in few minutes

PoC - Concept

Whole dataset stored in two engines



All relations

6,956,990,209 relations

All profiles

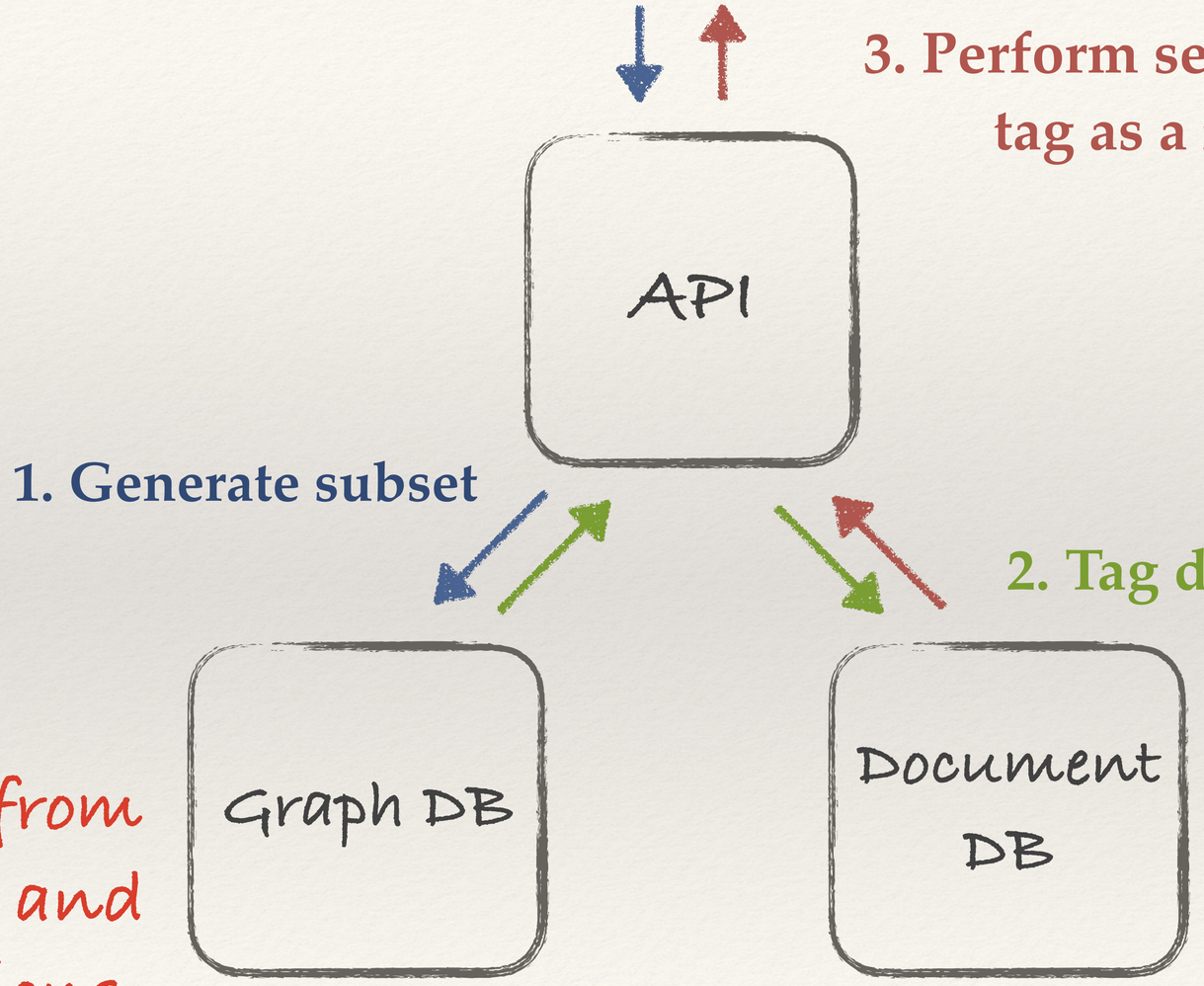
751,857,081 profiles

evojam

Why graph DB?

- Fast graph traversal
- Easily extendable with new relations and vertices
- Convenient algorithm description

PoC - Flow



3. Perform search with tag as a filter

Search with unique id as a primary filter

1. Generate subset

Generate subset from graph with users and companies relations

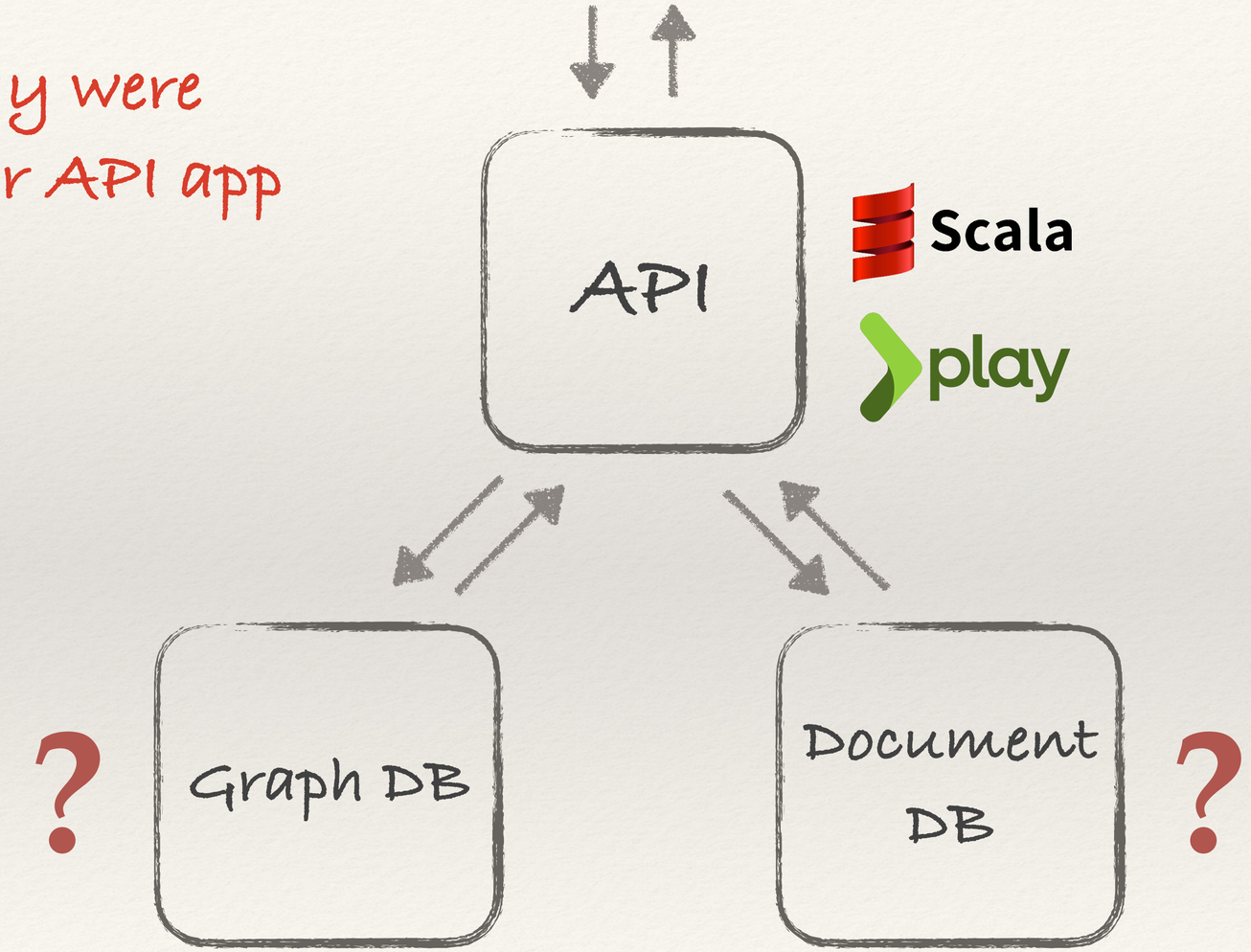
2. Tag documents

Tag documents with unique id in database with all profiles

evojam

Weapon of choice

Scala and Play were natural choice for API app



Some research required for databases

evojam

First Steps: Extraction

To make a research we had to put our hands on the real data

Extract anonymized profiles, companies and relations

Cleanup data, sort and generate input files

few days to pull, streaming with Akka and Slick

evojam

First Steps: Pushing forward

Push profiles for
searching purposes

Document DB

two tools,
highly dependent on db engines

Push vertices and
relations for traversal

Graph DB

evojam

Fulltext Searchable Document DB

- Mature
- Horizontally scalable
- Fast indexing (~3k documents per second on the single node)
- Well documented
- With scala libraries:
  - `https://github.com/`**`sksamuel/elastic4s`**
  - `https://github.com/`**`evojam/play-elastic4s`**

We already had significant experience with scaling Elastic Search for 80 millions

Which graph DB?

Considered three engines, OrientDB and Neo4j in community editions

evojam

Goodbye Titan

OrientDB®

neo4j

- Performed well in ~60M tests
- fast traversing
- Development has been put on hold?

evojam

- No batch insertion, slow initial load
- Stalled writes after 200 millions of vertices with relations
- Horror stories in the internet

Neo4j FTW

https://github.com/**AnormCypher/AnormCypher**

OrientDB®

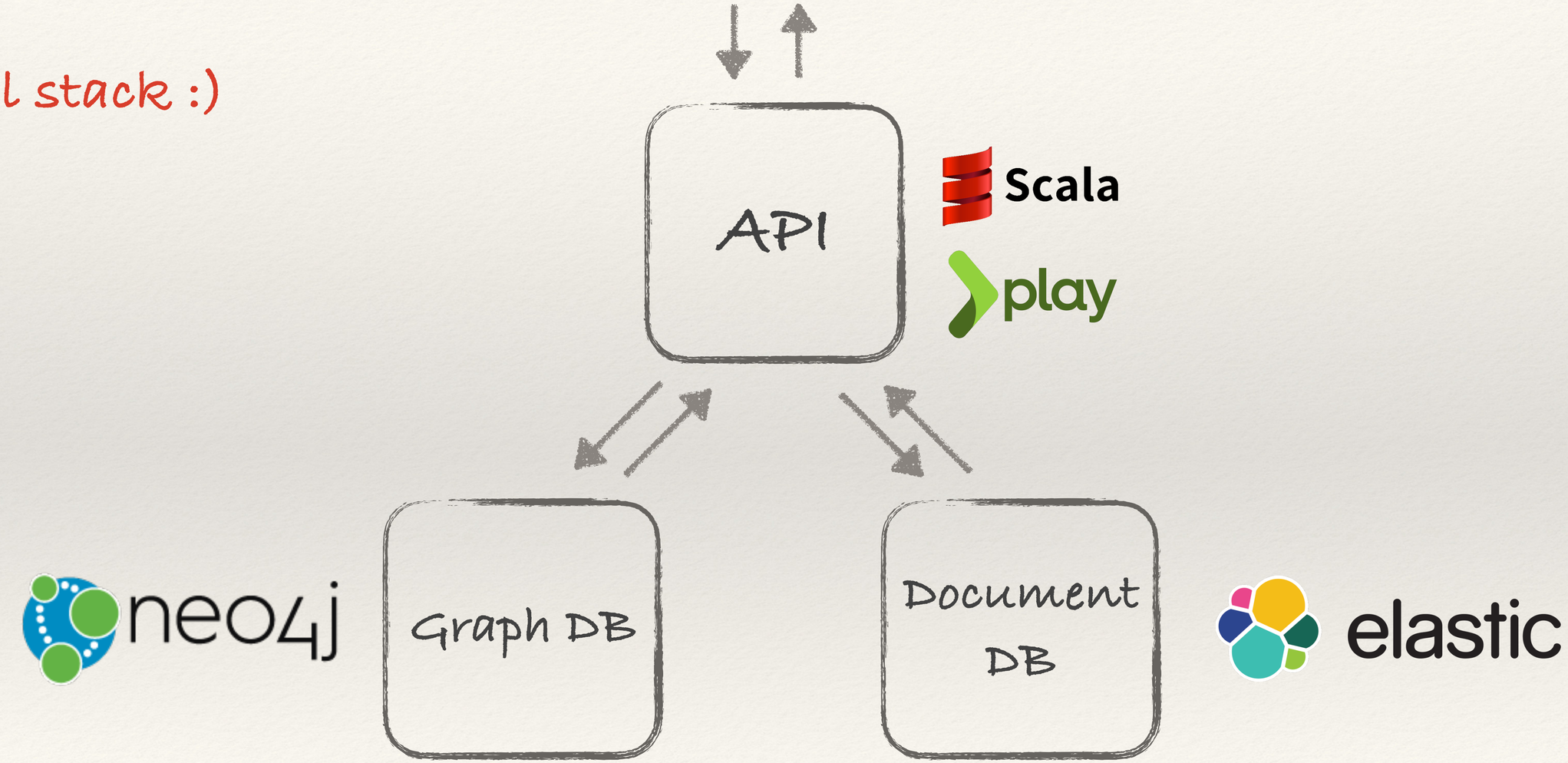
neo4j

- Fast
- Already known
- Convenient in Scala with AnormCypher
- Offline bulk loading
- Result streaming

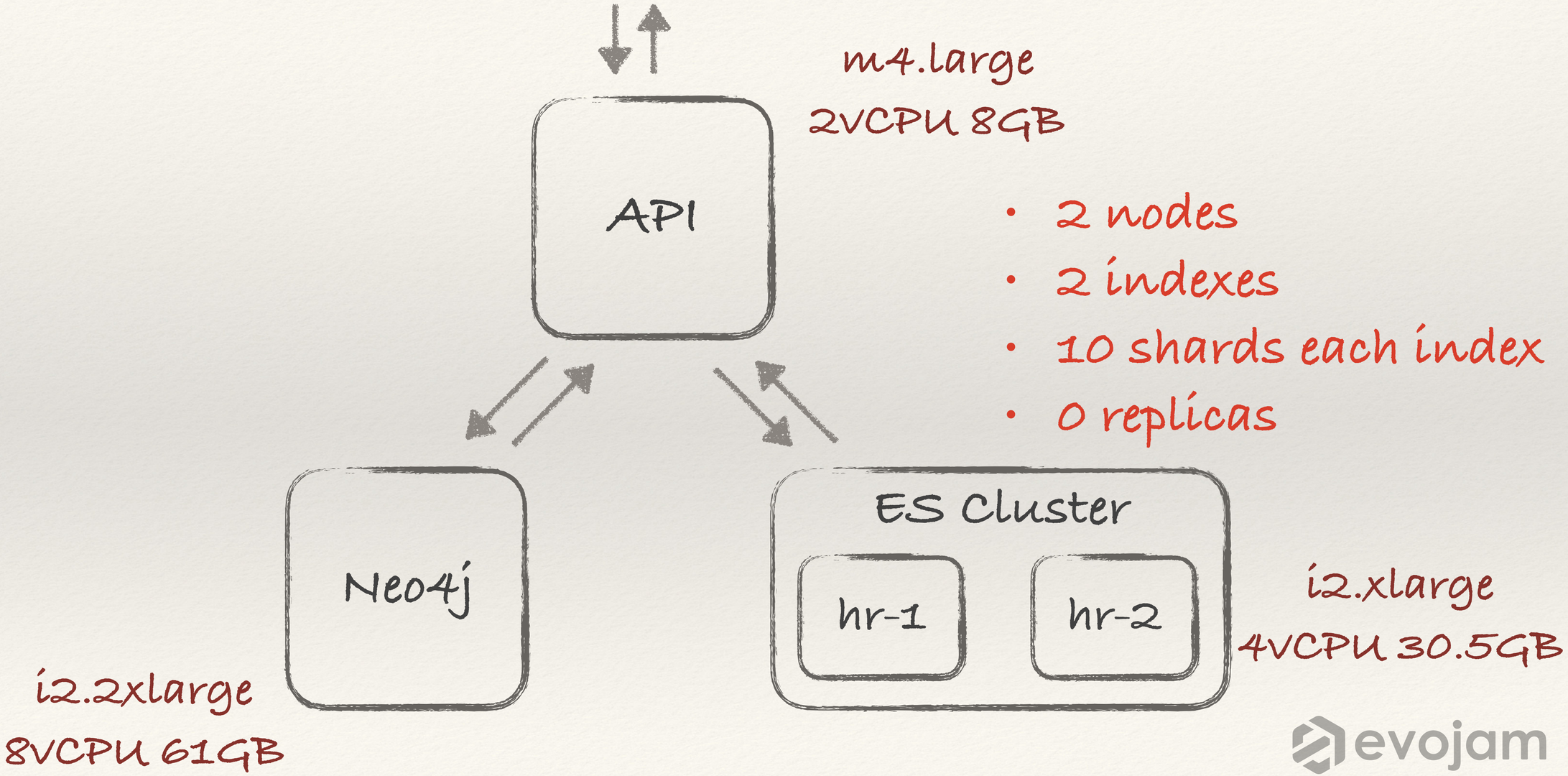
evojam

Weapon of choice

Final stack :)



Final Setup on AWS



evojam

Step #1 - Bulk loading into Neo4j

```
Importing the contents of these files into data/graph.db.

[…]

IMPORT DONE in 3h 24m 58s 140ms. Imported:
  835273352 nodes
  6956990209 relationships
  0 properties
```

It took 12 hours on 2 times smaller Amazon instance

evojam

Step #2 - Bulk loading into ES

Akka advantage: CPU utilization, parallel data enrichment, human readable

1. Create Source from CSV file, frame by \n
2. Decode id from ByteString, generate user json
3. Group by the bulk size (eg.: 4500)
4. Throttle
5. Execute bulk insert into the ElasticSearch

evojam

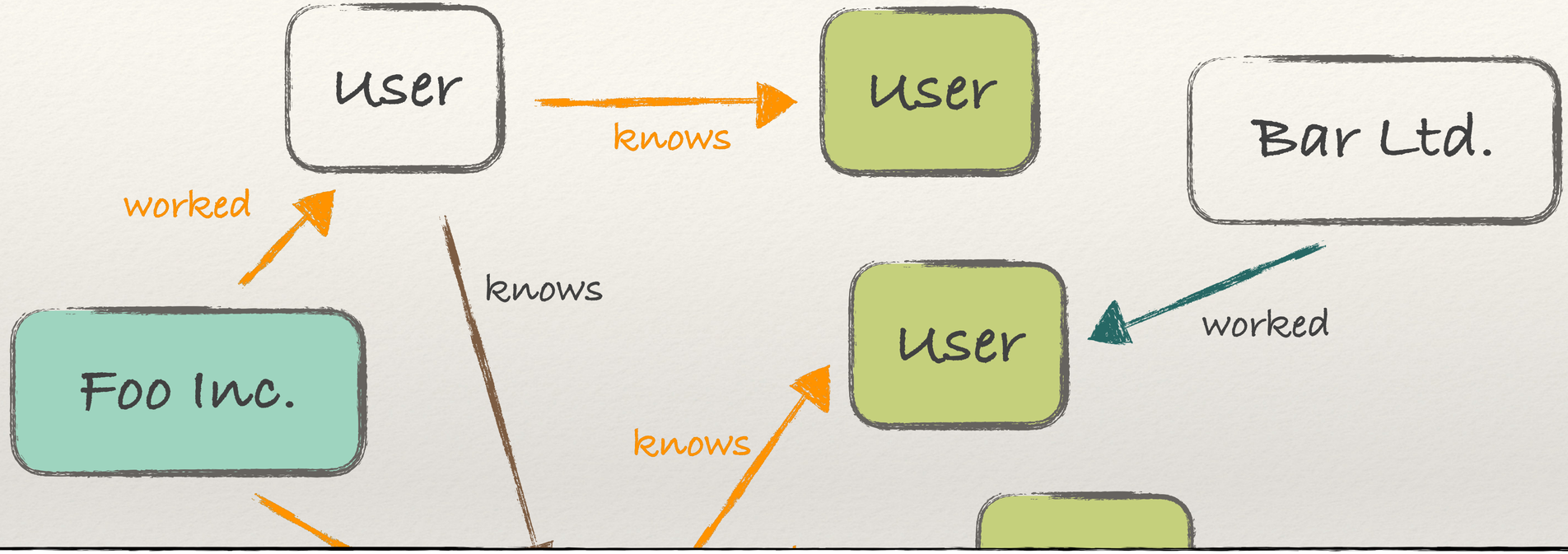
Step #2 - Bulk loading into ES

Flow description with Akka Streams

```
FileIO.fromFile(new File(sourceOfUserIds))
  .via(
    Framing.delimiter(ByteString('\n'),
    maximumFrameLength = 1024,
    allowTruncation = false))
  .mapAsyncUnordered(16)(prepareUserJson)
  .grouped(4500)
  .throttle(
    elements = 2,
    per = 2 seconds,
    maximumBurst = 2,
    mode = ThrottleMode.Shaping)
  .mapAsyncUnordered(2)(executeBulkInsert)
  .runWith(Sink.ignore)
```

evojam

Step #3 - Tagging




```
MATCH (c:Company)-[:worked]-(employees:User)-[:knows]-(aquitance:User)
WHERE ID(c)={foo-inc-id} AND NOT (aquitance)-[:worked]-(c)
RETURN DISTINCT ID(aquitance)
```

Neo4j traversal query in CYPHER

Akka Streams to the rescue

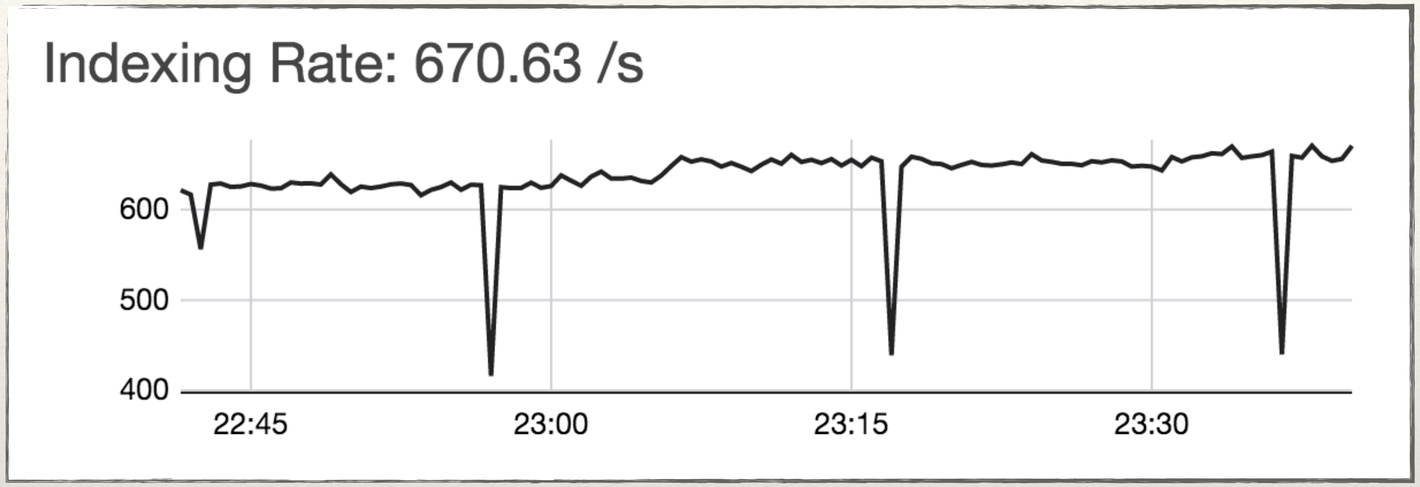
```
val idEnum : Enumeratee[CypherRow] = _

val src =
  Source.fromPublisher(Streams.enumeratorToPublisher(idEnum))
    .map(_.data.head.asInstanceOf[BigDecimal].toInt)
    .via(new TimeoutOnEmptyBuffer())
    .map(UserId(_))
    .mapAsyncUnordered(parallelism = 1)(id =>
      dao.tagProfiles(id, companyId))
```

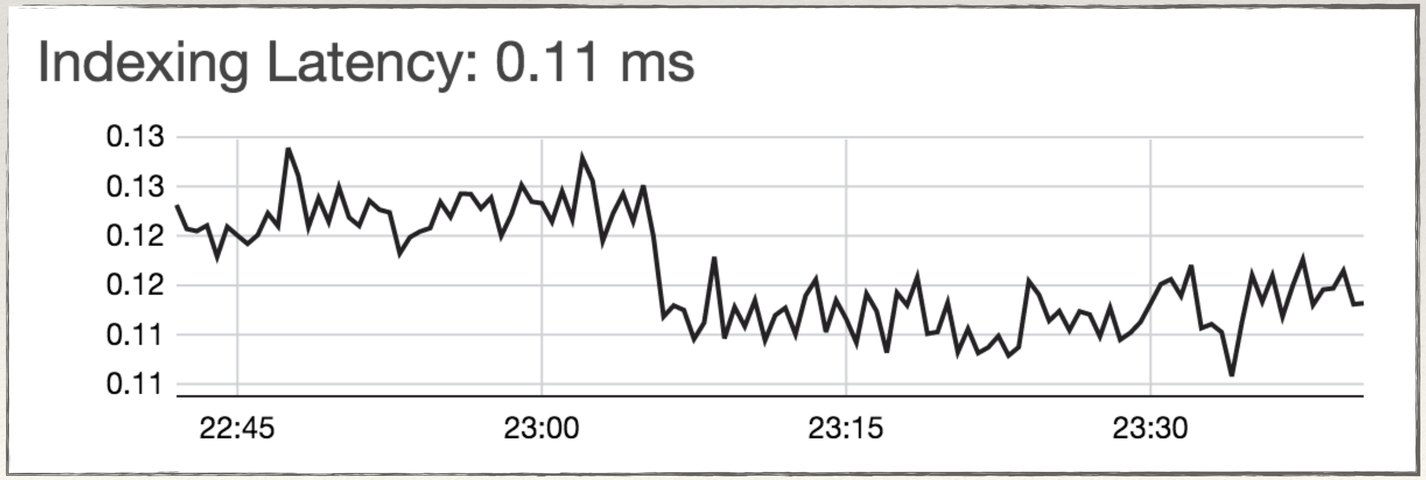
Readable flow

Buffering to protect Neo4j when indexing is too slow

Timeout due to the bug in the underlying implementation

Bottleneck

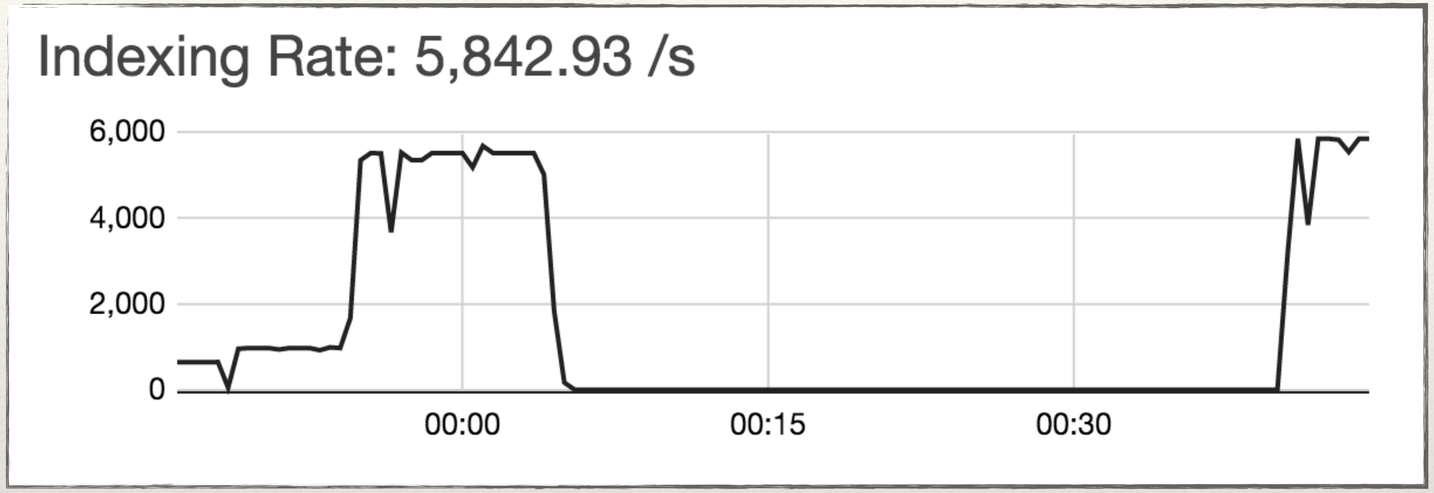




1.5 hour for 3 million subset, that's too long!

evojam

Bulk update with AkkaStream tuning

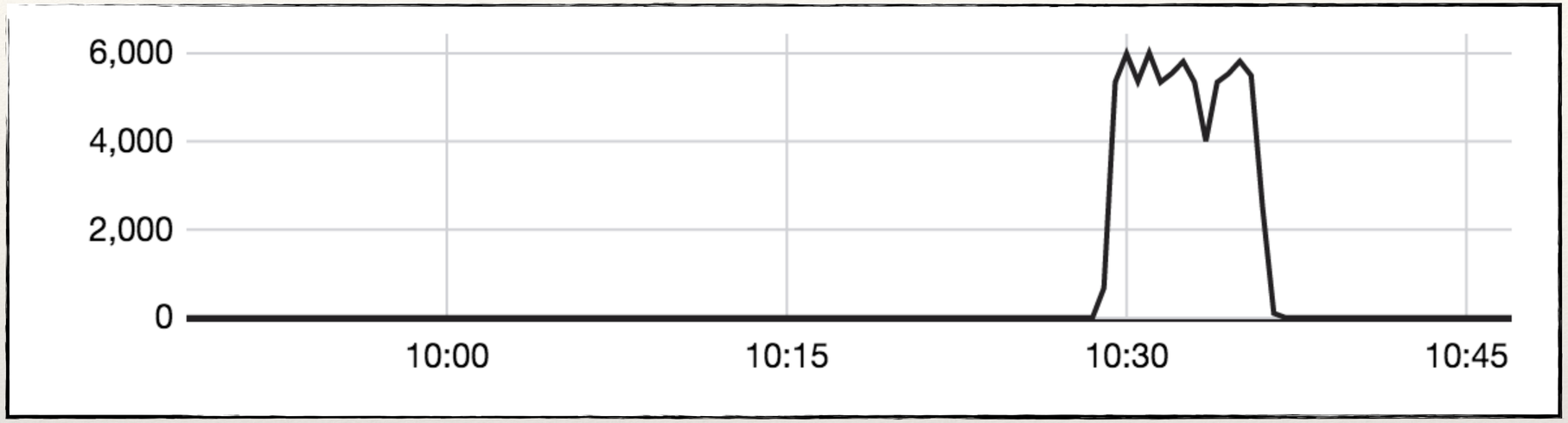



```
src
  .grouped(20000)
  .throttle(
    elements = 2,
    per = 6 second,
    maximumBurst = 2,
    mode = ThrottleMode.Shaping)
  .mapAsyncUnordered(parallelism = 1)(ids =>
    dao.bulkTag(ids, ...))
```

Bulk tagging, few lines

evojam

Tagging Foo-Company



2,222,840 profiles matching criteria

~14 seconds until first batch is tagged

~7 minutes 11 seconds until all tagged

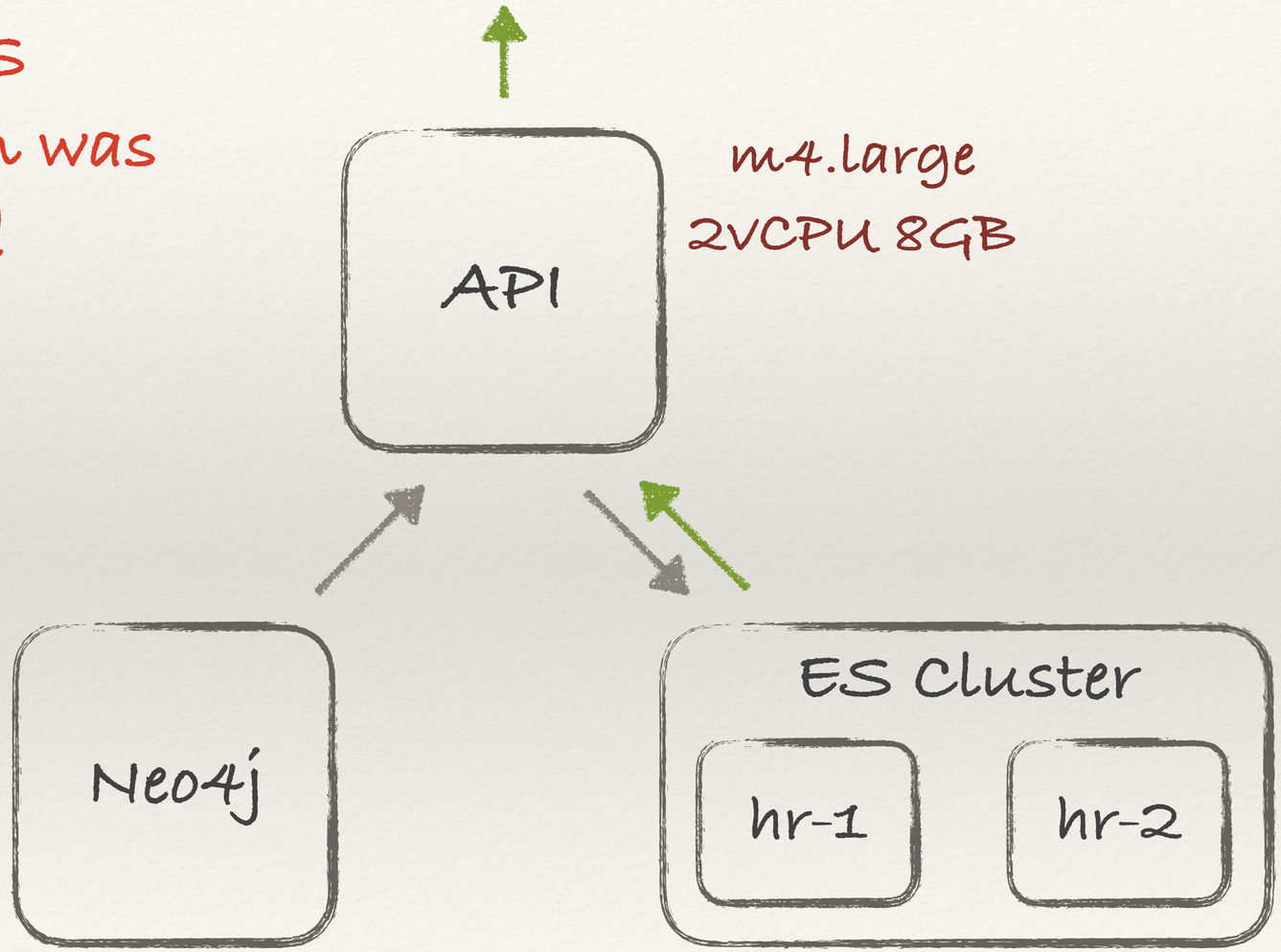
reference implementation - **few hours**

Sample subset :)

evojam

Step #5 - Search

With data ready in ES
search implementation was
pretty straightforward



evojam

Step #5 - Search benchmark

```
GET /users?company=foo-company&phrase=John
```

Phrases based on real names and surnames, used during profiles enrichment

Fulltext search on 2 millions subset

2000 phrases for the benchmarking

Response JSON with 50 profiles

Searching in 750 millions database

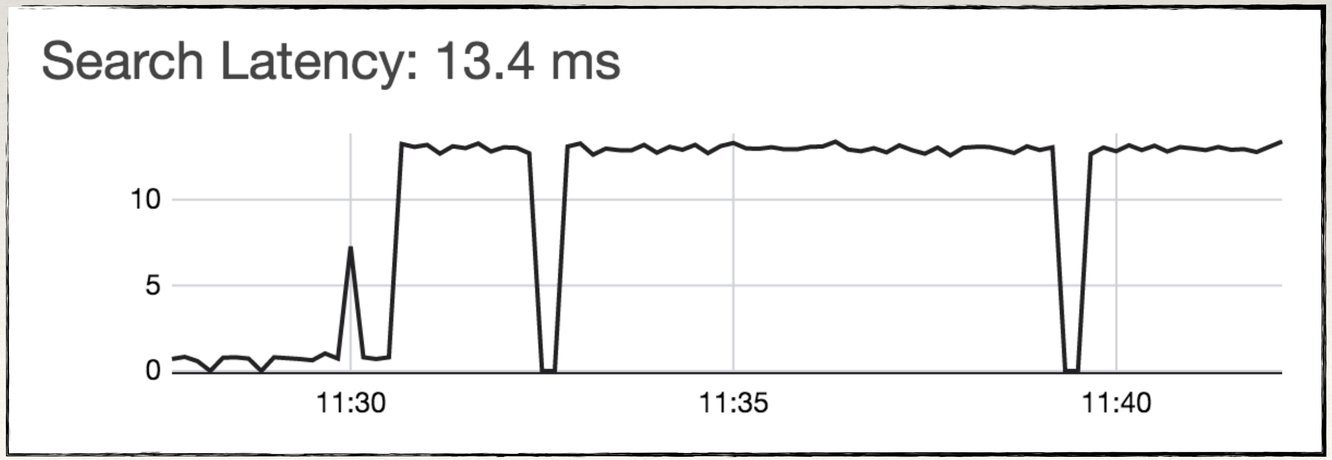
Random requests ordering

Step #5 - Search under siege

50 concurrent users

Response time ~ **0.14s**

constant search rate



constant latency

evojam

Objective achieved

14 seconds until first batch is tagged

7 **minutes** until 2 millions ready

search response time from API ~**0.14s**

Summary

reference implementation	PoC
manual	automatic
few days	14 seconds
few days	7 minutes
~40 millions	~750 millions
no analytics	GraphX ready

Tool ready for data scientists

We can implement core traversal modifications almost instantly

evojam

Artur Bańkowski
artur@evojam.com
@abankowski

Summary

http://neo4j.com/

https://www.elastic.co/

https://playframework.com/

http://doc.akka.io/docs/akka-stream-and-http-experimental/2.0/scala/stream-index.html

Try at home!

https://snap.stanford.edu/data/

Sample graphs ready to use