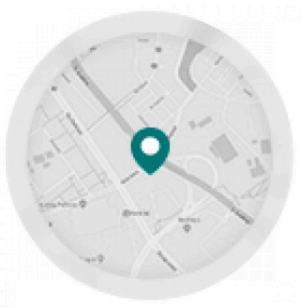
Crunching Petabytes with jumpshot®

whoami

Alexander Hagerf 🇸🇪

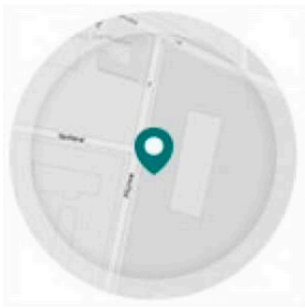
- 🎓 M.Sc. Royal Institute of Technology (KTH)
- 🧮 Engineering Physics, Mathematics and Machine Learning
- 👨🏼‍💻 Senior Software Engineer
- 4 years Java, Stockholm
- 3+ years in Prague
- 1.5 years GoodData
- Soon 2 years in Jumpshot, working in Spark/Scala

Offices



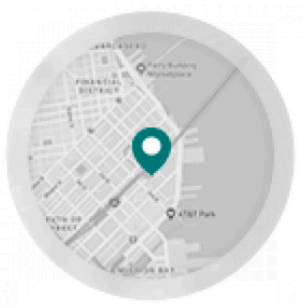
Prague

Pikrtova 1737/1A, 140 00
Prague 4, Czech Republic



Brno

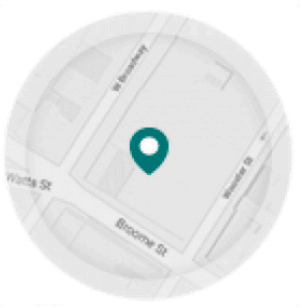
Přízova 5/7,
602 00
Brno, Czech Republic



San Francisco

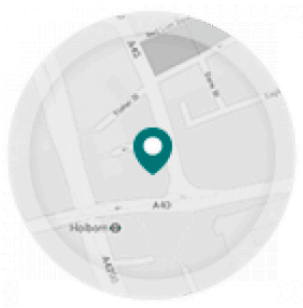
Global Headquarters

333 Bryant St Suite 240
San Francisco, CA 94107



New York

379 West Broadway,
Suite 550
New York, NY 10012

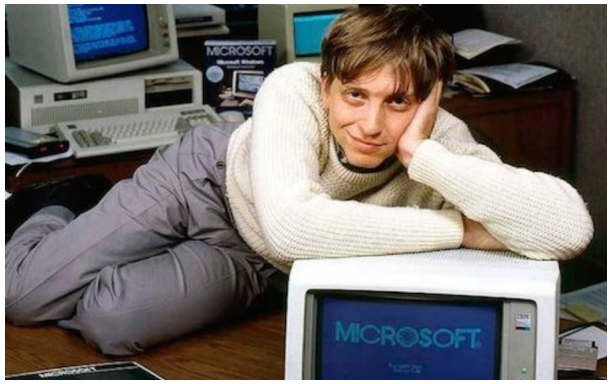


London

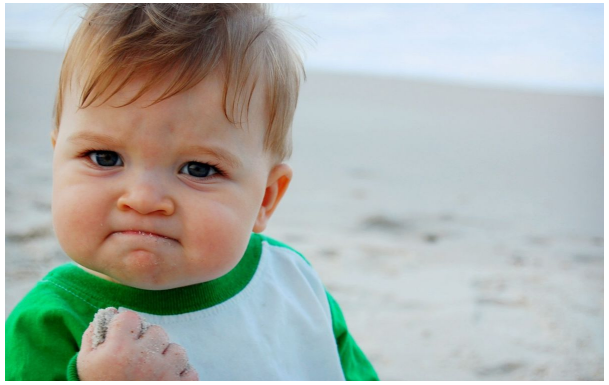
110 High Holborn
London WC1V 6RD United
Kingdom

Outline

Business cases



Tech overview



Lessons learned

jumpshot®

Business cases

What the hell do we do?

Visited		Interacted		Added to Cart		Started Checkout		Converted	
Amazon	Walmart	Amazon	Walmart	Amazon	Walmart	Amazon	Walmart	Amazon	Walmart
4,507,338	591,584	66.2%	68.1%	11.7%	8.3%	8.3%	6.6%	7.9%	4.5%
		2,985,801	402,960	529,584	48,979	374,418	39,144	357,202	26,897

First Things First

We purchase varied data sources from different partners and before receiving it our data is completely **anonymised**, **aggregated**, **PII-stripped** and **GDPR compliant**. Thus we cannot, nor do we have any interest in, tracking specific peoples behaviours, personal information or anything of the sort.

We sell metrics and estimates, nothing else.

jumpshot®

Jumpshot is the only company that unlocks walled-garden data to empower marketers to target and expand their customer base.

Our real-time, global panel of 100 million devices tracks 5 billion actions a day to deliver insights into online behavior from every consumer action. We report on behavior with precision and detail, capturing every aspect of a click and purchase, even in the internet's most valuable walled gardens like Amazon, Netflix, and Google.

Our data reflects consistent, real-time behavior, not projections from spotty recruitment panels. We provide key insights into how specific audiences behave throughout the entire digital ecosystem, using real consumer behavior to inform marketing strategies and help quantify the impact of campaigns.

Basically we process and enrich data to create insights, metrics and other analytics

Customers verticals

- **Data Platforms** 💾 - doing data analysis in general
- **Finance** 💼 - using data for financial investments
- **Market Research** 🔬 - analysing e-commerce transactions
- **Ad Tech** 🛍️ - mostly publishers optimising advertisement revenue
- **Brand customers** 🎯 - brands themselves analysing internet behaviour, competitor analysis

Products

Data delivery formats 📦

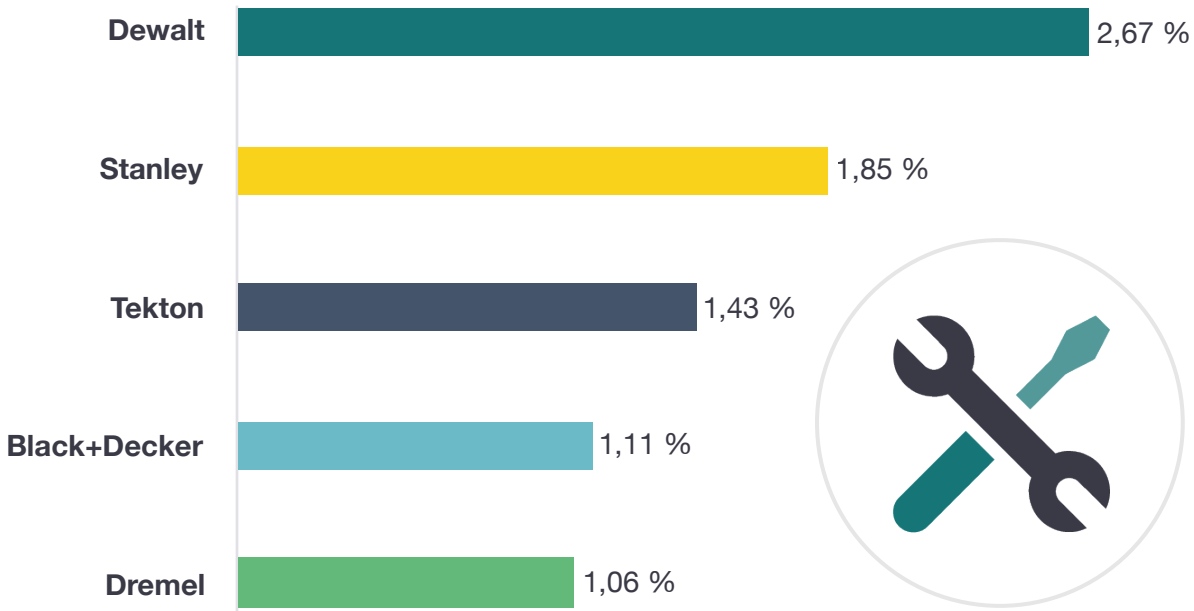
- **Feeds** - File based transfer (shared on Amazon S3, FTP transfer, email ...), supported formats are text (csv, ctrl+a separated etc.) and binary parquet
- **Insights web application** - customers can generate custom report in theirs browser

Data delivery timing ⏱

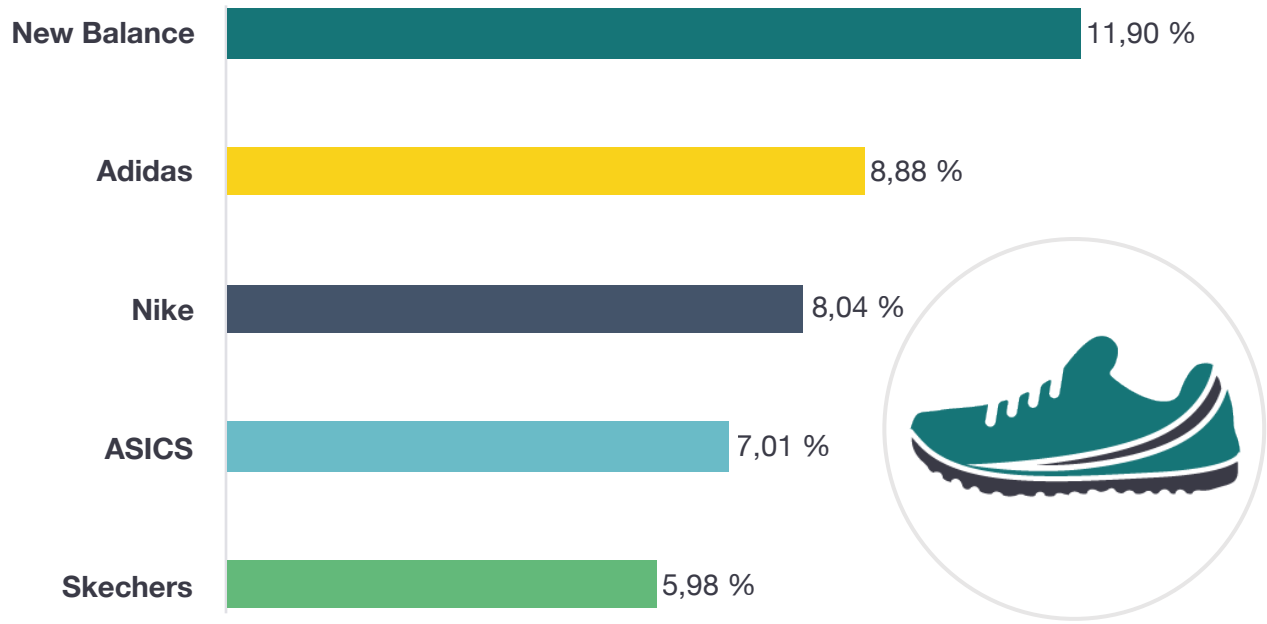
- On a regular basis (mostly daily) - Feeds
- Based on a planned recalculation - Insights app
- Ad-Hoc based on customer request - Data samples before contract is signed, one-time delivery.

Example of Insights

Q1 Top Brands Amazon Market Share,
Home Improvement Tools Category



Q1 Top Brands Amazon Market Share,
Men's Athletic Shoes Category



jumpshot

jumpshot®

Technology Overview

What technology do we use and how do we use it?

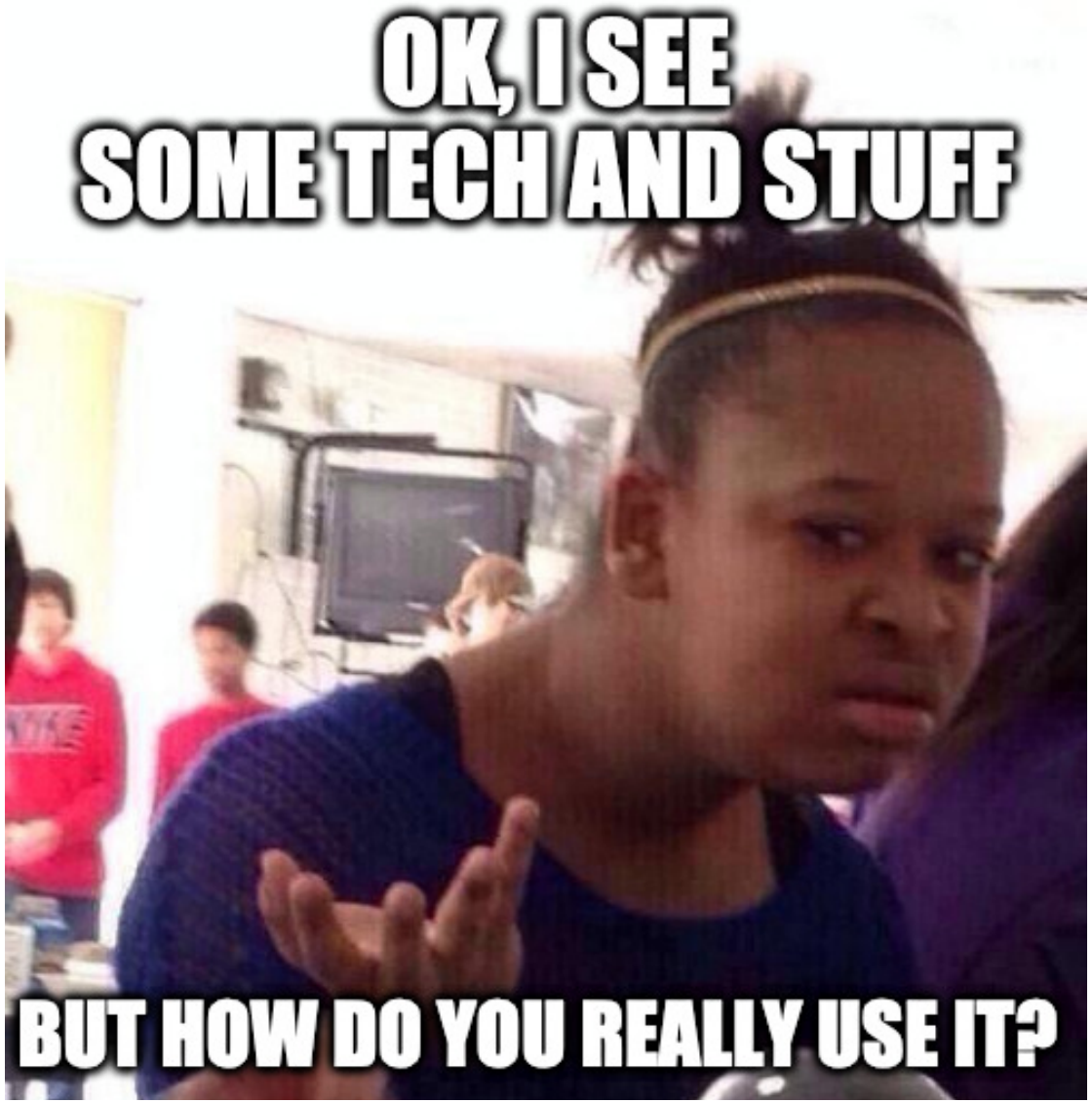
Jumpshot tech stack

Usage	Technology
Programming language	• Scala - for applications running on a regular basis • Python - for supporting tools (data investigation, monitoring etc.)
Orchestration/scheduling	Azkaban, Nomad (for docker applications) or in-house tools
Ad-hoc data analysis	PySpark, Pandas, Jupyter notebooks
Build tool	Gradle
Data processing	Spark v2.x, Kafka, Yarn
Testing	TDD and BDD tools created in-house (Scala)
Monitoring	Icinga, ViktorOps, in-house tools
Code maintenance	GitLab
Artifacts repository	Nexus
Automation	GitLab CI/CD, Jenkins
Distributed filesystem	HDFS
Concurrency/REST API	Akka

Data size

Overview

- 5+ data sources
- Tens of millions of devices
- Raw data > 1 PB/month
- Saved on cluster ~10 TB per day



Engineering

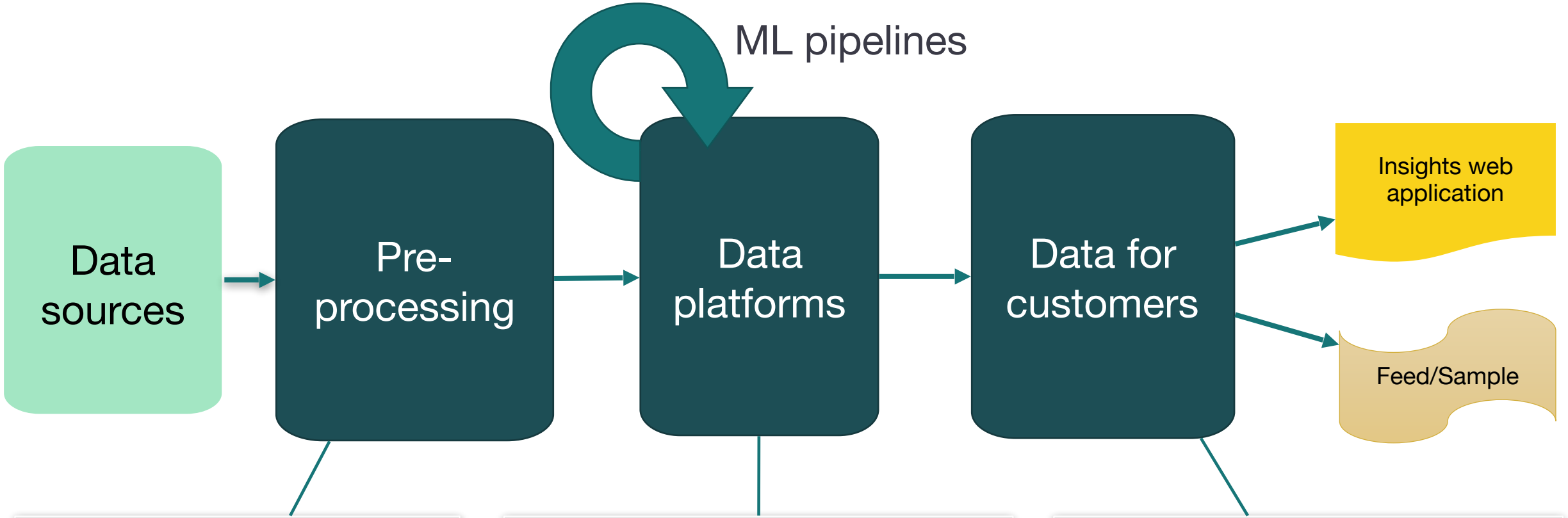
Exploratory 🕵🏻

- **Understanding the data** - what can we do with the data, what are the quirks?
- **Machine Learning** - scale up for whole population, determine brands, build models etc.
- **Monitoring** - how does quantity and quality fluctuate over time?

Building 👷🏼

- **Data processing pipelines** - efficient, reliant pipelines for data
- **Data retrieval applications** - web crawlers, scrapers, tools for manual investigations
- **Data delivery pipelines** - internal systems for versatile data deliveries to internal/external purposes
- **Monitoring** - create systems for anomaly detection, alarms
- **Customer applications** - front- and back-end for end-user facing applications

Data pipeline high level overview



jumpshot

jumpshot®

Lessons learned

How we try to overcome all the problems we have 😎

Pain points

Two physically separate clusters 👯

- Syncing data
- Consistency

Many data platforms 🌓

- Different paths, formats, partitioning, buckets…
- Dependencies, timing & availability
- Versioning

Performance & size of data 🏎

- Resource management
- Timing
- Recalculations

Good developer discipline

Keep It Simple Stupid (KISS)

Separation of Concerns (SoC)

Singe Responsibility Principle (SRP)

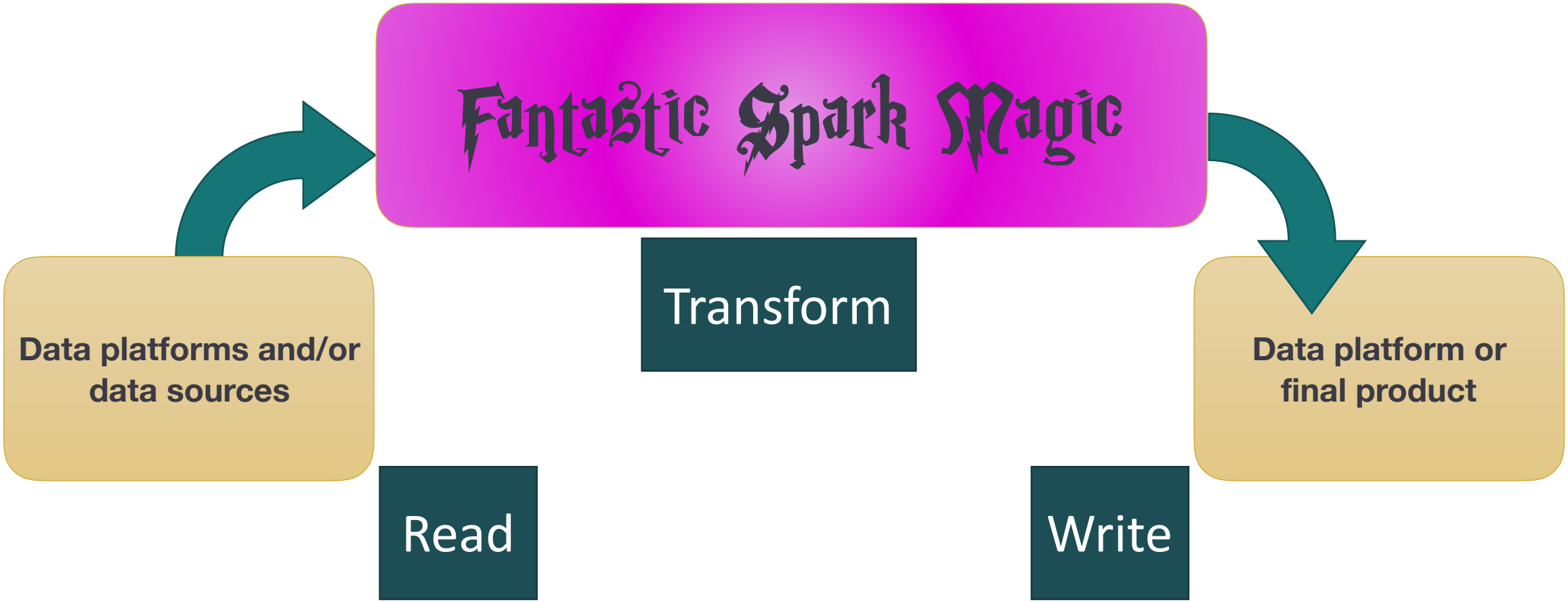
Don’t Repeat Yourself (DRY)

Test Driven Development (TDD)

Behaviour Driven Development (BDD)

Pair programming

How do Spark applications look?



Spark Helpers library

Spark Readers 📖

- Classes for reading data platforms placed on HDFS
- Tested and maintained by team responsible for data
- Supports higher abstraction functionality

```
import com.jumpshot.spark.readers.streaming_media
val streamingReader: StreamingReader = production

val data = streamingReader.read(("2018-12-01", "2018-12-02").dateRange)
```

Spark Helpers library

Spark Writers 

- Higher abstraction of writing various formats
- Tested, and proven efficient
- Takes care of (re-)partitioning, shuffling, bucketing

```
import org.apache.spark.sql._
import com.jumpshot.spark.writers._

SingleFileDatasetWriter.writeAvro(data, "/home/alexander.hagerf/avro")

val partitionedWriter1 = PartitionedDatasetWriter("year", "month", "day")

partitionedWriter1.writeParquet(data, "/home/alexander.hagerf/parquet")
partitionedWriter1.writeCsv(data, "/home/alexander.hagerf/csv", ",", SaveMode.Append)

val partitionedWriter2 = partitionedWriter1.withNumPartitions(500)

partitionedWriter2.writeText(data, “/home/alexander.hagerf/text")
```

Spark Helpers library

Spark tools 🛠️

- Common features/functions for building efficient transformations
- Classes for building better tests: unit, integration and BDD

```
import com.jumpshot.spark.test.SparkIntegrationTest

@RunWith(classOf[JUnitRunner])
class StreamingReaderIntegrationTest extends FlatSpec with Matchers with SparkIntegrationTest {

  var reader: streamingReader = _

  override def beforeAll(): Unit = {
    super.beforeAll()
    reader = streaming.v1.production
  }

  "Streaming reader" should "not be empty on 2018-10-15" in {
    val result = reader.read(LocalDate.of(2018, 10, 15))
    result should not be empty
…
```

Know your frameworks

Learn Scala properly

a. Functional paradigm

b. Eco-system, libs

c. Efficient executions

Learn Spark properly

a. DataFrame/Dataset

b. Functionality, best practises

c. “Under the hood”

Scenario

Given:

Two data sources that have then been pre-processed:

a) ad-data `[device-hash, ad-url, timestamp]`

b) click-data `[device-hash, url, timestamp]`

Goal:

Find out which ads have been clicked within 30 minutes of being shown.

Other information:

Urls in both dataset don't necessarily have to be 100% the same, minor variations in encoding, added parameters etc. can be present

Attempt 1

Proposed solution:

Read the data as data frames in Spark, join on device-hash and filter out tuples that don't match in timing and url similarity

Implementation:

```
ads.alias("a").join(clicks.alias("c"), $"a.hash" === $"c.hash")
      .filter($"c.timestamp" - $"a.timestamp" < 1800000L)
      .filter(levenshtein($"a.url", $"c.url") /
         greatest(length($"a.url"), length($"c.url")) < 0.1)
```

Result:

Running never finishes. Not even on just one hour of data.

Analysis

`Spark UI and logs:`

A lot of CPU is used in levenshtein calculation

Attempt 2

Proposed solution:

Levenshtein is $O(n * m)$ while sift4-approx is $O(n + m)$. Replace the string distance algorithms!

Implementation:

```
ads.alias("a").join(clicks.alias("c"), $"a.hash" === $"c.hash")
      .filter($"c.timestamp" - $"a.timestamp" < 1800000L)
      .filter(sift4($"a.url", $"c.url") /
         greatest(length($"a.url"), length($"c.url")) < 0.1)
```

Result:

It still never finishes! 😡 Not even on just one hour of data.

Analysis cont.

`Size of data:`

24h: ~7 million devices, ~5 billion clicks, ~250 million ads

=> ~714 click/device * ~36 ads/device * 7 million devices

179 928 000 000 combinations!

Analysis cont.

Problems with joins in Spark:

BroadcastJoin can be efficient on small data

SortMergeJoin is standard for bigger data, and creates all tuples then filters them.

All joins in Spark are implemented using **cogroup**

Algorithm proposal:

Instead of generating all possible tuples, lets only generate those that can be in the correct time range, and have similar lengths before doing deeper analysis on string similarity.

Attempt 3

Implementation:

```
ads.groupByKey(_.hash).cogroup(clicks.groupByKey(_.hash)){

(hash, adIterator, clickIterator) =>

   if (clickIterator.isEmpty || adIterator.isEmpty)
     Iterator.empty else {

   val sortedClicks = clickIterator.toArray.sortBy(_.timestamp)

   adIterator.flatMap(ad => {
     getTimestampIndex(ad.timestamp)
       .fold(Array.empty) { startIndex =>
         val endIndex = getTimestampIndex(ad.timestamp + 1800000L)
           .getOrElse(sortedClicks.length)
         util.Arrays.copyOfRange(sortedClicks, startIndex, endIndex)
           .filter(urlsMatch(ad, _))
       }.map(click => Result(hash, click.timestamp, click.url, ad.url)
     )
   })
}
```

Attempt 3

`Result:`

With some additional tweaking, one day finishes in ~30 minutes.

Thank You!

alexander.hagerf@jumpshot.com