



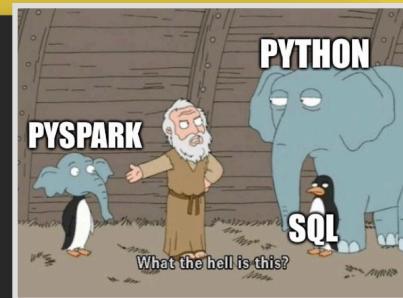
Что это и для чего?





# Python + SQL = PySpark

```
datamart = final_df\  
    .select('card_order_dt',  
            'card_num',  
            'cookie',  
            'url',  
            'amt',  
            'status')\  
    .withColumn('transaction_level',  
               F.when(F.col('amt') > 300, True).otherwise(False))\  
    .withColumn('status_flag',  
               F.when(F.col('status') == "выдана", True).otherwise(False))\  
    .withColumn('partition_date', F.lit(date).cast('string').alias('partition_date'))\  
    .withColumn('load_date', F.lit(date).cast('string').alias('load_date'))\  
    .drop('amt', 'status')
```



- Это по сути те же SQL запросы, но которые пишутся с помощью API Spark
- В Spark можно сохранять в переменные результат огромных запросов





# Что выводит Spark?



card_num	transaction_datetime	status	card_order_dt	url	cookie	amt
0119300157687351	2024-07-02 06:32:10	выдана	2024-07-02 05:56:00	http://example.com/page6	387PbPJnn6	495.92
0172264584552366	2024-07-02 11:38:10	выдана	2024-07-02 10:50:00	http://example.com/page8	hYNB1sx13C	127.58
0175736131090843	2024-07-03 01:00:10	выдана	2024-07-02 23:35:00	http://example.com/page10	5ik2aVEYrz	230.64
0243084951118144	2024-07-02 22:06:10	выдана	2024-07-02 21:06:00	http://example.com/page3	6yojZuAfot	124.68
0261912375736877	2024-07-02 20:35:10	выдана	2024-07-02 20:10:00	http://example.com/page5	6Cna4zRUVh	263.69
0265281487020131	2024-07-02 20:16:10	выдана	2024-07-02 19:58:00	http://example.com/page6	uwKt1srlT9	480.28
0273241962063845	2024-07-02 21:35:10	выдана	2024-07-02 21:27:00	http://example.com/page4	jWAR4JXCd3	325.34
0279120596277629	2024-07-02 18:27:10	выдана	2024-07-02 18:04:00	http://example.com/page1	nsG2REvpQ1	278.21
0284862617881895	2024-07-02 10:19:10	выдана	2024-07-02 09:32:00	http://example.com/page9	cXc1ZGHJ14	96.95
0330327384276660	2024-07-02 16:26:10	выдана	2024-07-02 16:07:00	http://example.com/page7	pBzwQRglpj	223.21
0352288633871827	2024-07-02 02:14:10	выдана	2024-07-02 00:56:00	http://example.com/page1	S5o8Y6kk2Q	118.83
0363957946193526	2024-07-02 11:24:10	выдана	2024-07-02 10:03:00	http://example.com/page4	DuYhoR7v5y	182.85
0385277165882644	2024-07-02 14:36:10	выдана	2024-07-02 14:30:00	http://example.com/page9	TXR4rHqjvi	342.65
0407156816322679	2024-07-02 05:10:10	выдана	2024-07-02 04:07:00	http://example.com/page3	52AQm2FTTX	351.8
0423001802153299	2024-07-02 18:37:10	выдана	2024-07-02 18:19:00	http://example.com/page7	DrKjhPfdsr	384.68
0458751505613552	2024-07-02 11:06:10	выдана	2024-07-02 10:47:00	http://example.com/page2	hBUGawRMf9	299.79
0458771495029206	2024-07-02 23:31:10	выдана	2024-07-02 23:22:00	http://example.com/page4	OadbtgOckr	469.6
0478546133824549	2024-07-02 12:32:10	выдана	2024-07-02 12:31:00	http://example.com/page4	WFryumWlz	455.13
0484391462295158	2024-07-02 07:59:10	выдана	2024-07-02 07:58:00	http://example.com/page9	ZRVzQeeUzf	297.16
0576342477104189	2024-07-02 14:15:10	выдана	2024-07-02 12:50:00	http://example.com/page4	WH9178PSH1	287.44

only showing top 20 rows

- Весь результат работы Spark это просто таблица

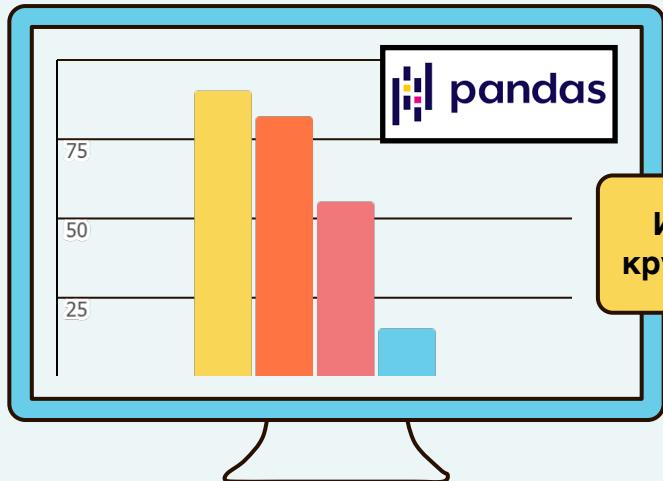
Таблицу можно сохранить в:

- HDFS
- S3
- CSV
- Parquet
- куда хотите





# Pandas vs Spark



Работает на одной машине

\*Pandas - библиотека для python



Работает на многих машинах

\*Spark - фреймворк

# Где запускается Spark?





# Как именно работает Spark?

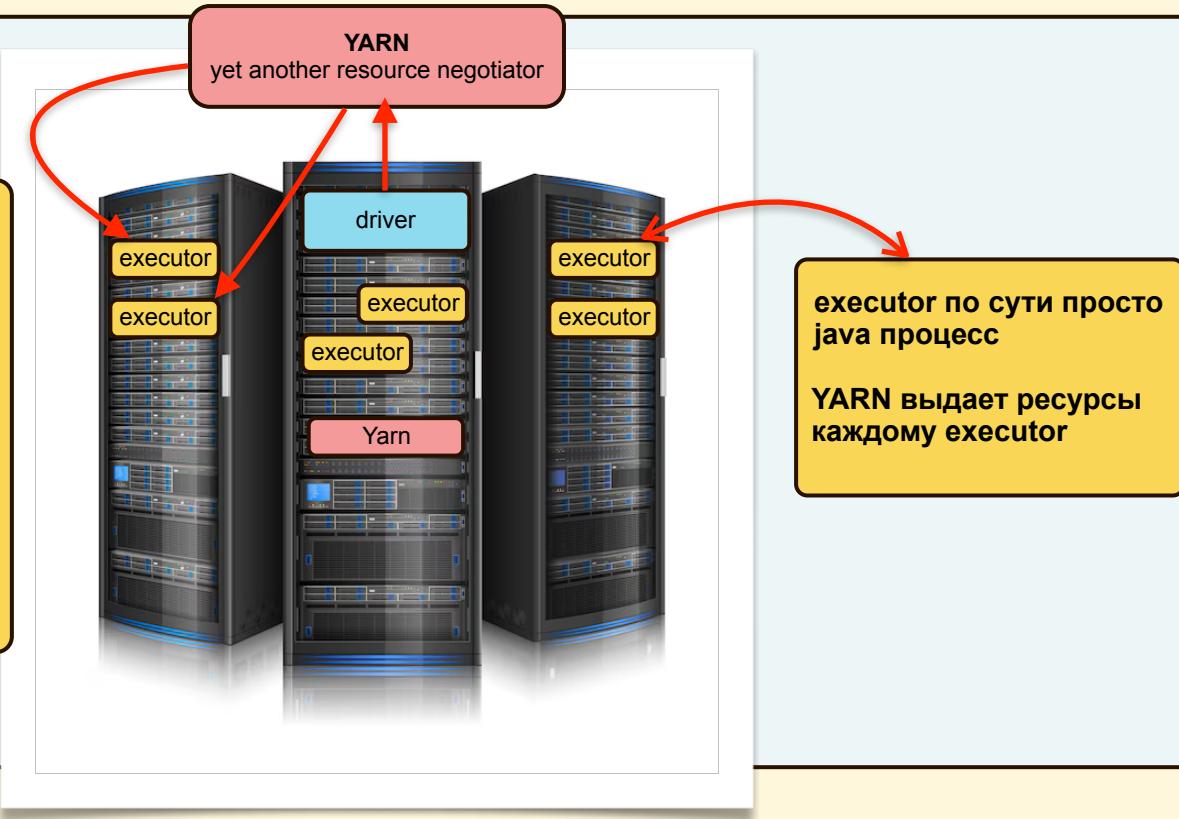
## **YARN - это менеджер ресурсов**

## **Driver может:**

- создавать executor
  - убивать executor
  - перезапускать executor

**Мы можем задать требуемое кол-во executor:**

spark.executor.instances -> 6





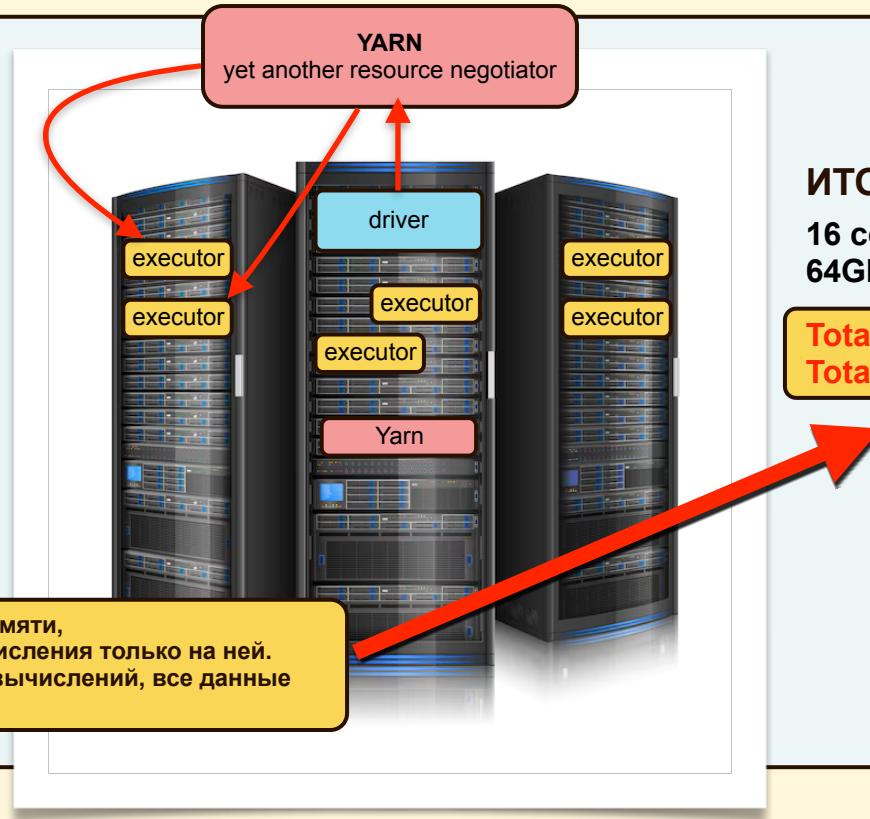
# Как выделяют ресурсы в Spark?

Ресурсы одного сервера  
(сервер = worker = node)  
**16 cores 64Gb RAM**

\* характеристики могут быть разными  
у разных серверов

Например у нас кластер  
из 10 серверов

Нам важно кол-во оперативной памяти,  
так как Spark производит все вычисления только на ней.  
Если Spark остановить в момент вычислений, все данные  
из RAM пропадут



ИТОГО

**16 cores x 10 nodes  
64Gb RAM x 10 nodes**

**Total memory = 640Gb RAM  
Total cores = 160 cores**

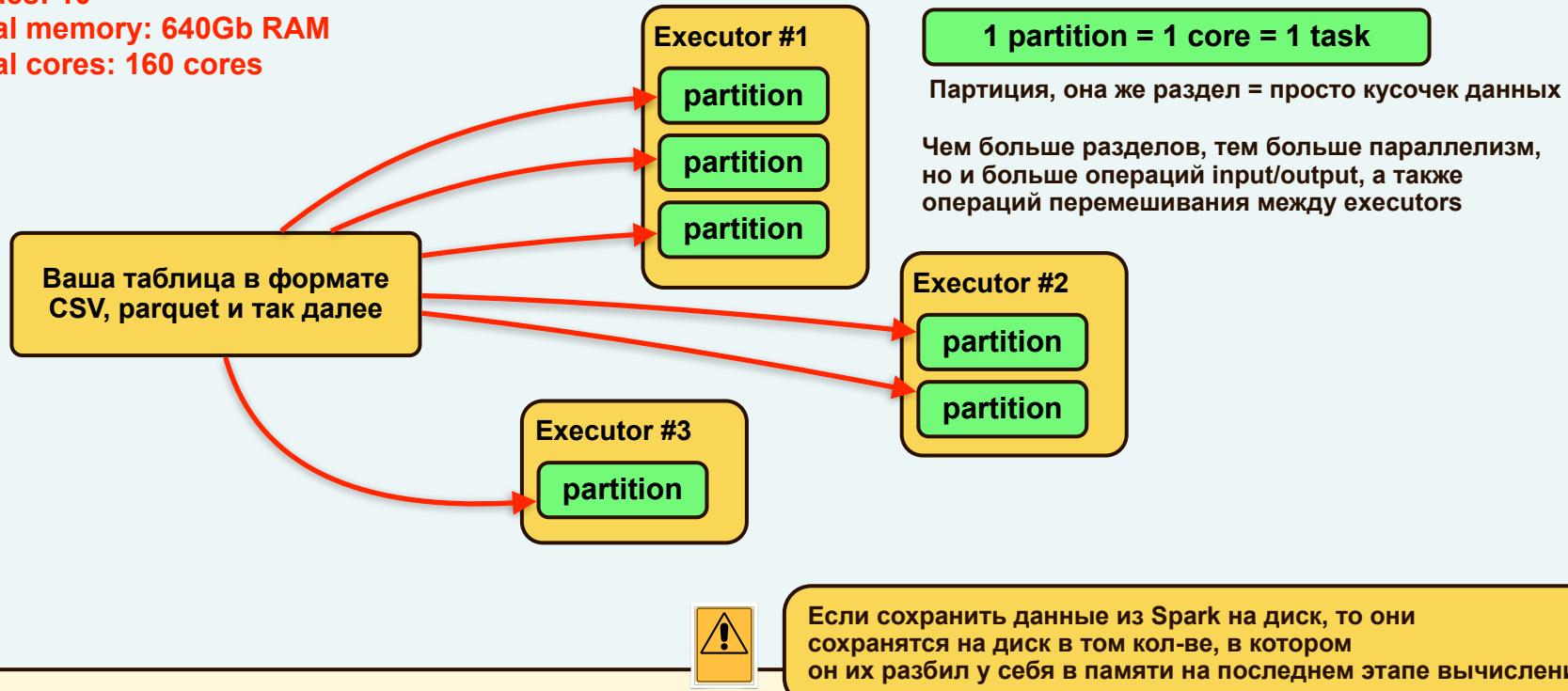


# Как Spark читает данные с диска?

Nodes: 10

Total memory: 640Gb RAM

Total cores: 160 cores



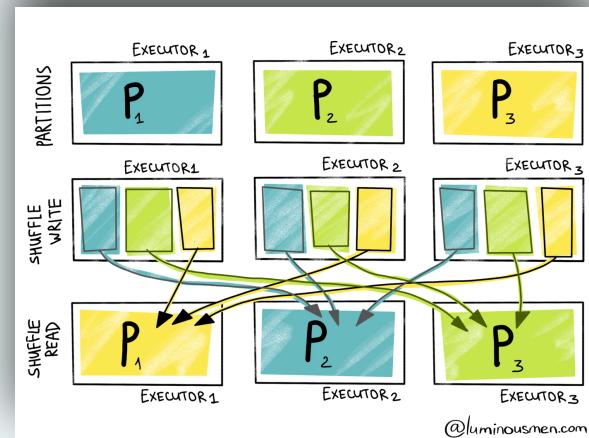
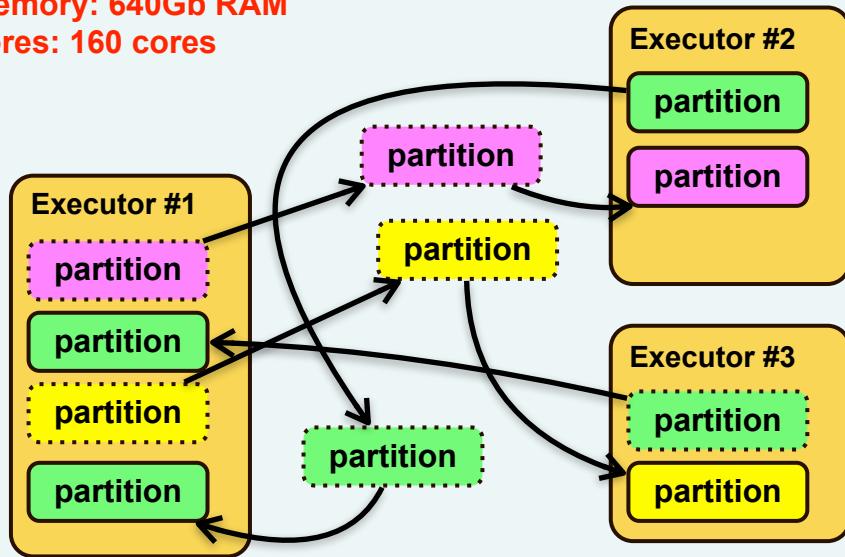


# Что такое shuffle?

Nodes: 10

Total memory: 640Gb RAM

Total cores: 160 cores



@luminousmen.com



Операция shuffle по факту это просто перемещение данных между executors. Это достаточно тяжелый процесс и в идеале следует стремиться уменьшать кол-во таких операций.  
Shuffle возникает, когда мы делаем orderBy, groupBy, join и прочие



# Где находится executor?

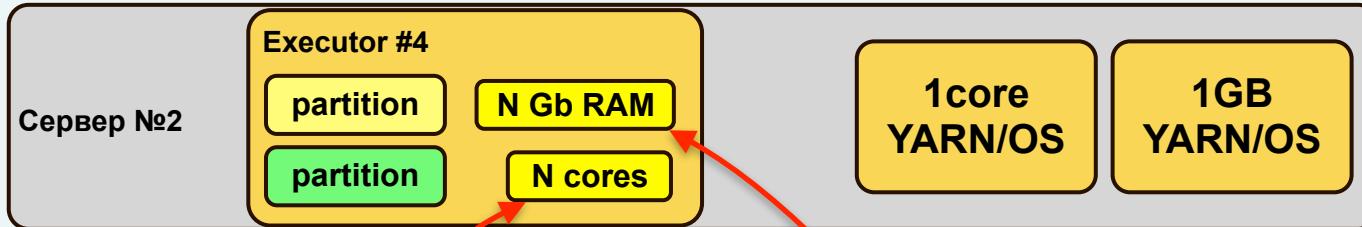
Nodes: 10

Total memory: 630Gb RAM

Total cores: 150 cores

Executors запускаются на серверах, где есть свободные ресурсы

Каждый сервер по умолчанию жрет ресурсы для YARN/HADOOP/OS



Кстати у нас  
теперь всего  
150 ядер  
630Gb RAM

И каждый executor жрет ресурсы сервера



# Сколько нужно executors?

Nodes: 10

Total memory: 630Gb RAM

Total cores: 150 cores

## Сколько выделить executors?

executor



executor



executor



1 core

$$150 / 1 = 150 \text{ executors}$$

Плюсы:

- Хороший параллелизм

Минусы:

- Много операций I/O
- Большое кол-во shuffle

5 cores

$$150 / 5 = 30 \text{ executors}$$

Оптимально.

Рекомендуют от 1-2 cores

\* В целом максимум до 5 cores

15 cores

$$150 / 15 = 10 \text{ executors}$$

Плюсы:

- Малое кол-во shuffle

Минусы:

- Малый параллелизм
- При падении executor придется пересчитывать большой кусок данных



# СКОЛЬКО НУЖНО РЕСУРСОВ?

Nodes: 10

Total memory: 640 Gb RAM

Total cores: 160 cores

Total memory: 630 Gb RAM

Total cores: 150 cores

Total memory: 629 Gb RAM

Total cores: 149 cores

Node  
16 cores, 64 Gb RAM

Cores: 5 / executor

1core, 1Gb for Hadoop/YARN/OS (Per Node)

1core, 1Gb for Application Master = Driver

Executor cores: 5

Total executors: 149 cores / 5 cores = 29 executor

Executor memory: 629 Gb / 29 executors = 21 Gb (no overhead)

Overhead: max(384Mb, 10% of executor memory)

Executor memory: 21 Gb - 2.1 ~ 18 GB

num executors: 29  
executor cores: 5  
executor memory: 18

Memory per core:  
18 Gb / 5 cores ~ 3 Gb



# Как устроена память executor?

num executors: 29  
executor cores: 5  
executor memory: 18

Executor #1 (On heap Memory)

18Gb

Execution  
Memory

Память для  
вычислений

5.4Gb

Storage  
Memory

Память для  
кэширования  
`spark.memory.storageFraction`

5.4Gb

Хранение пользовательских структур данных

User Memory

~6.9Gb

40% of available memory by default

Reserved Memory для внутренних объектов

300Mb

`spark.memory.fraction`

default: 60% from  
executor memory

По умолчанию память для  
кэша и вычислений  
раздается поровну

Но разницу можно увеличить,  
если изменить параметр  
`spark.memory.storageFraction`

Executor #1 Дополнительная память

`spark.executor.memoryOverhead`  
max(384Mb, 10%from10Gb)

1.8Gb

Off-heap memory  
default: 0

Накладные расходы VM

Память, где нет Java  
garbage collector