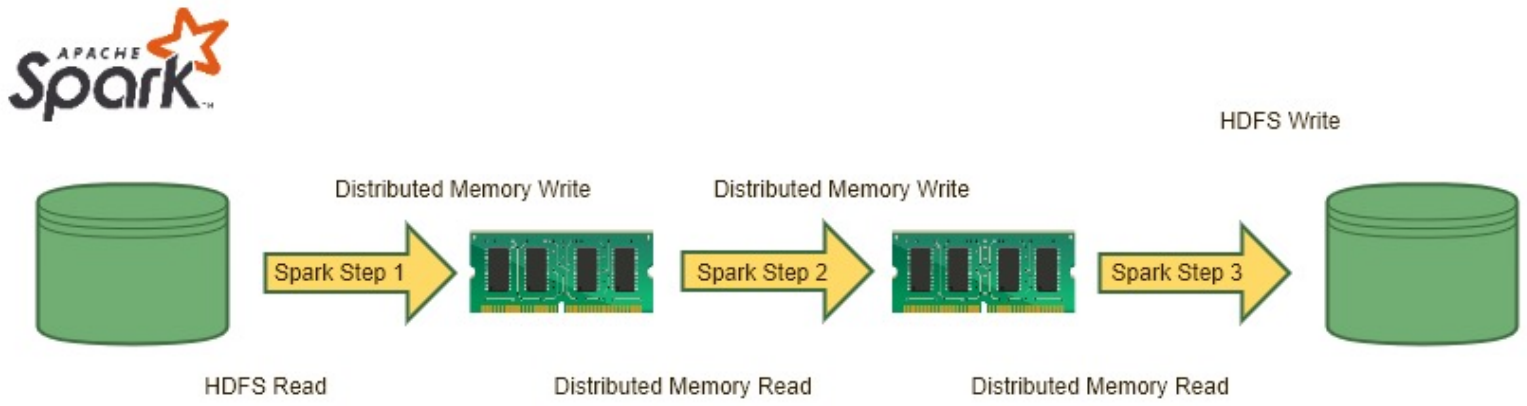
Apache Spark

О чем поговорим

- Apache Spark и MapReduce
- RDD
- DataFrame и Dataset
- Модули Spark
- PySpark
- Конфиги Spark



Apache Spark vs MapReduce

Apache Spark

Это еще один фреймвор для распределенных вычислений.

Его можно сравнивать с уже знакомым нам MapReduce. API есть на Scala, Java и Python (PySpark).

Внутри много модулей, таких как Spark Streaming, Spark ML, Spark SQL. О них мы еще сегодня поговорим.

Рассмотрим пример

- Есть таблица a – покупки пользователей (user, product…)
- Есть таблица b – информация о пользователях (user, country)
- Хотим получить покупки продуктов по странам
- Нужно сделать join по user

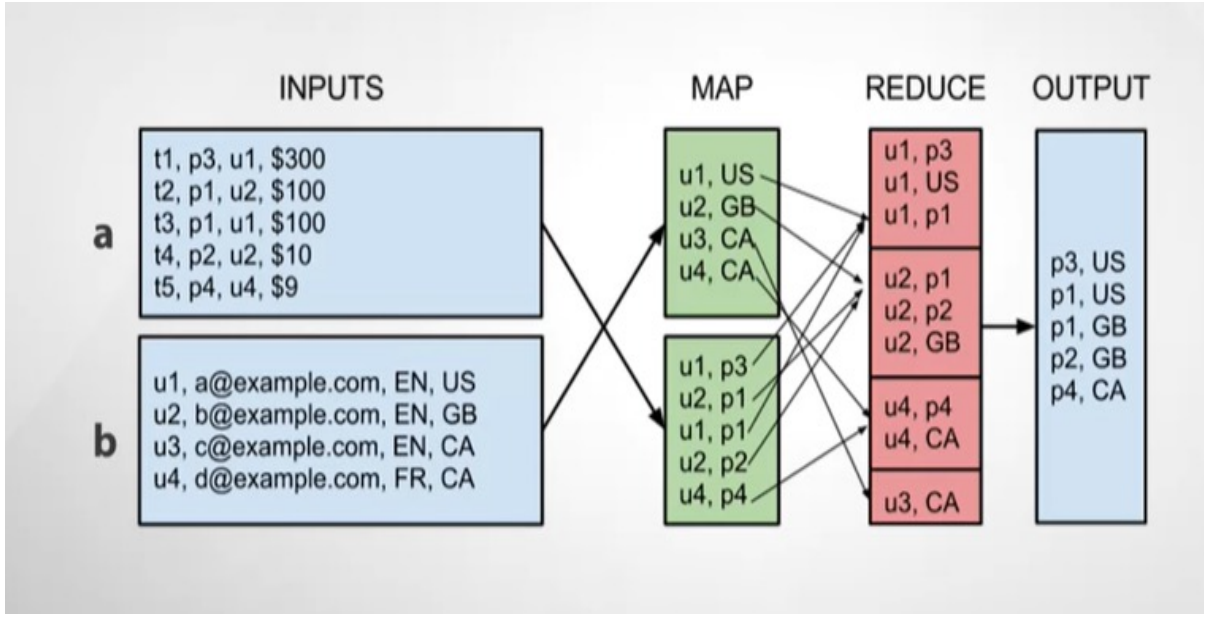
Что-то вроде такого:

```
select a.product, b.country
from a join b
on
a.user = b.user
```

Ну и дальше можно накидывать всякие group by.

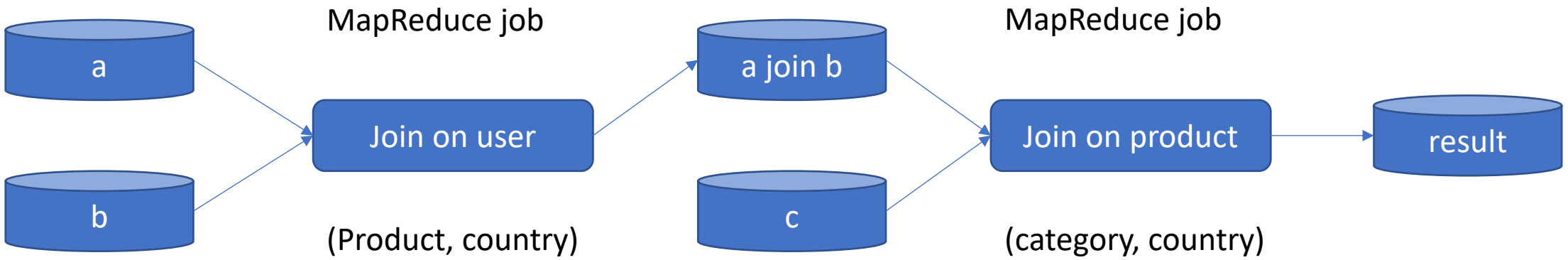
SQL join on MapReduce

- Считаем, что а и б – это не таблички, а просто файлы
- Мар для а – видим юзера, тянем юзера и продукт
- Мар для б – видим юзера, тянем юзера и код.
- Юзер – ключ, шафл раскидает всех юзеров по машинам. А дальше просто все комбинации.



А если несколько join-ов

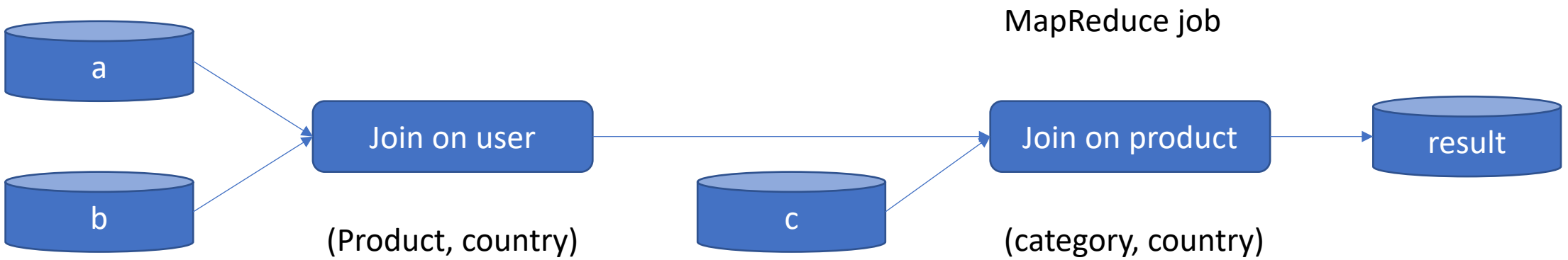
- Есть таблица c, в которой лежит информация о продуктах (product, type)
- Визуализируем join



- MapReduce хранит результаты в HDFS (в том числе результаты первого join сложит в hdfs)
- Тратим время на запись промежуточного результата и после на его же считывание с диска в HDFS

А чем Spark лучше?

Для Spark 2 join-а это не 2 задачи, которые ничего друг про друга не знают, а единый граф вычислений.



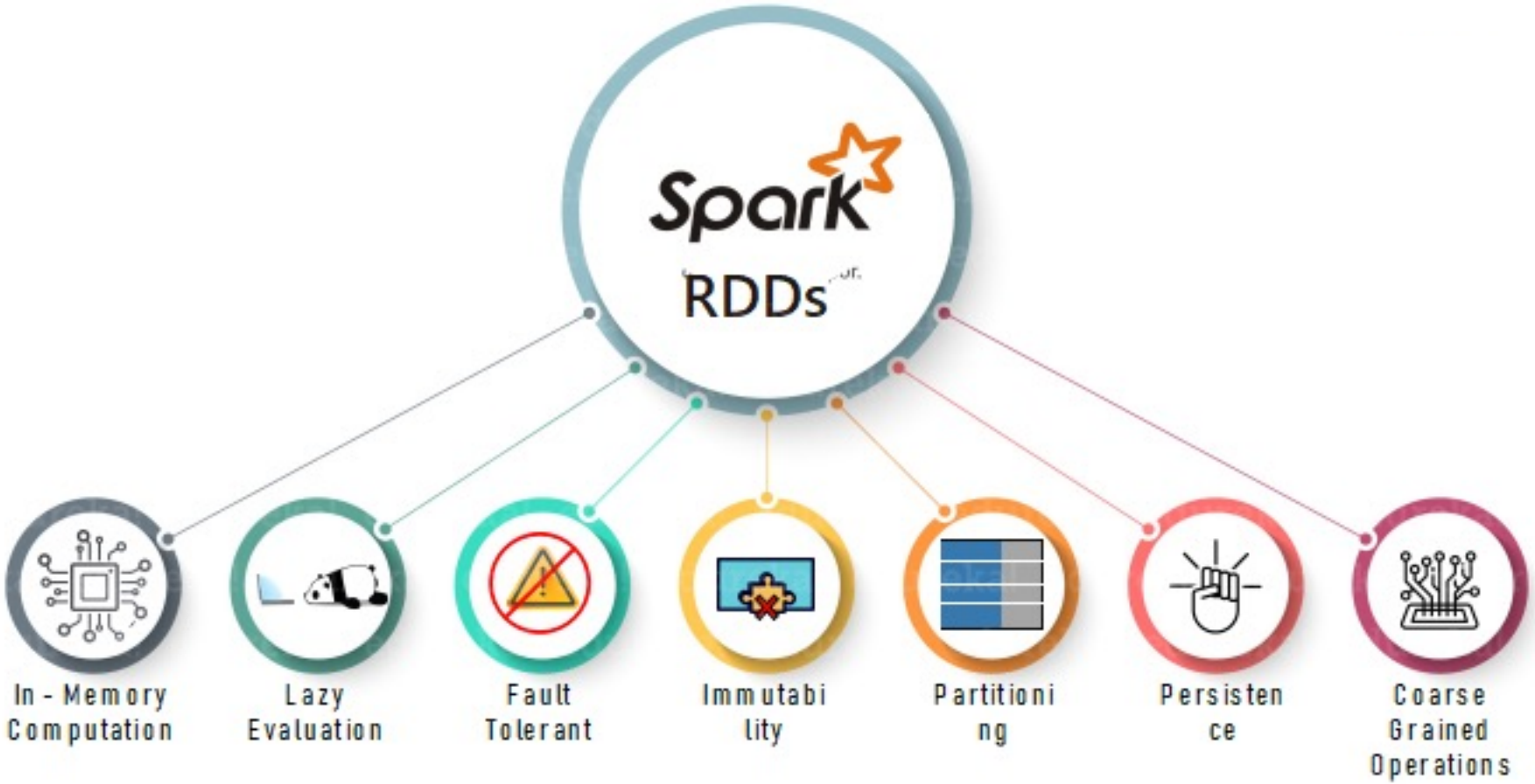
- Все вычисления описываются как DAG (Directed Acyclical Graph)
- Промежуточный результат хранится в памяти или на диске, минуя HDFS

Чем платим за скорость?

- Памяти нужно много, чтобы все стабильно и быстро работало
- Быстрые и большие диски, так как tmp будет быстро забиваться
- Избавляемся от overhead на чтение и запись в hdfs + нет расходов на постоянный запуск YARN

Spark vs MapReduce

	Spark	MapReduce
Область применения	Итерационные, интерактивные вычисления	Тяжелая пакетная обработка данных
Простота использования	Понятный API на Python	Неудобный интерфейс Hadoop Streaming
Утилизация RAM	Забивает память данными, если ее хватает	Все скидывает в HDFS
Итог	Быстро посчитает, если хватит ресурсов, но может упасть. Есть оптимизация	Может больше переваривать данных, но ждать придется долго


Spark
RDDs
In - Memory Computation
Lazy Evaluation
Fault Tolerant
Immutability
Partitioning
Persistence
Coarse Grained Operations

Spark RDD

RDD

Абстракция RDD – Resilient Distributed Dataset. Это основа Spark. Кстати, RDD является неизменяемым объектом!

- Восстанавливаемый распределенный набор данных
- Входы операций должны быть RDD
- Все промежуточные операции также RDD
- Так как известна цепочка вычислений DAG, то потерянные части восстанавливаются из входных данных

Операции над RDD

- Трансформации (RDD -> RDD)

Все трансформации являются ленивыми вычислениями, то есть мы вычисляем тогда, когда нужен результат. Например операции map, reduceByKey, join.

- Действия

Действия приводят к запуску DAG для расчета RDD, отдавая результат. Пример: saveAsTextFile, collect, count, write.

- Иные операции, такие как persist, cache. Они заставляют Spark сохранить RDD в памяти для получения быстрого результата.

DataFrame

Это распределенная коллекция данных в виде именованных столбцов, аналогично таблице в реляционной базе данных. DataFrame работает только со структурированными и полуструктурированными данными, организуя информацию по столбцам, как в реляционных таблицах. Это позволяет Spark управлять схемой данных.

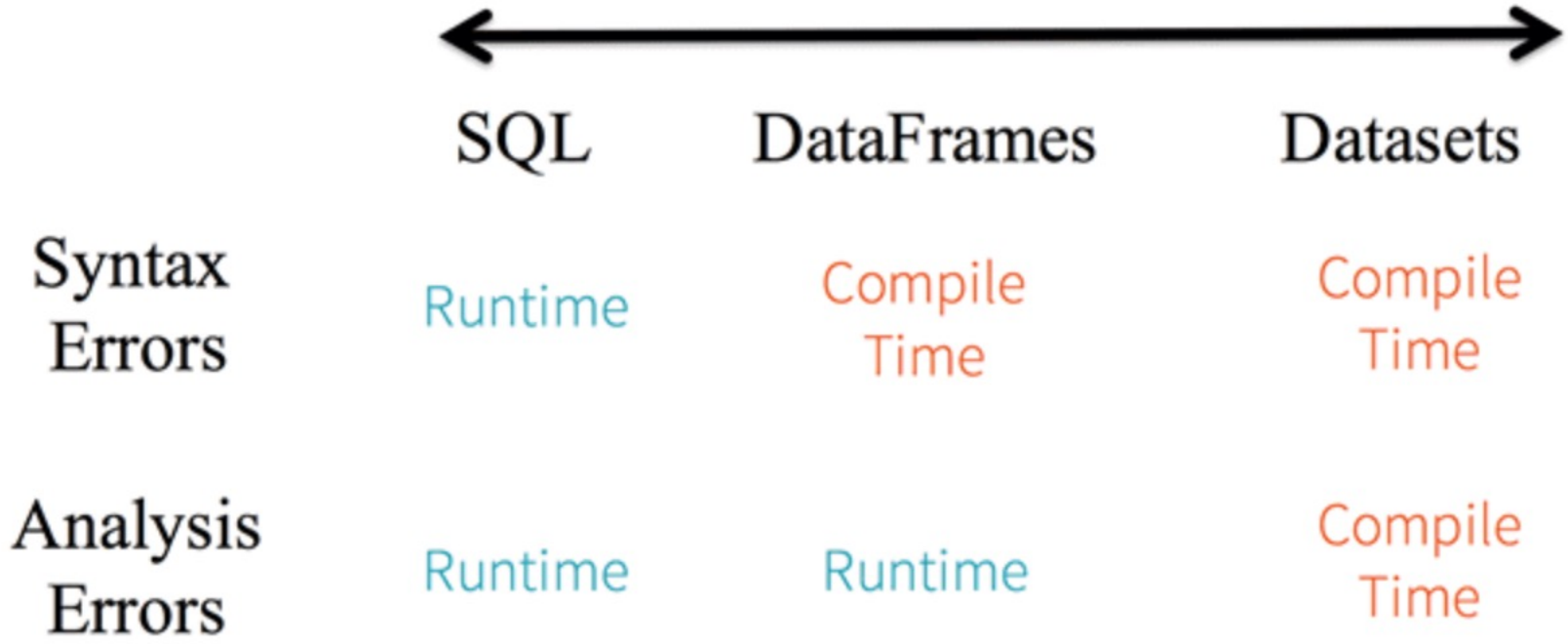
- Автоматически определяет схему данных
- DataFrame обрабатывается быстрее RDD
- DataFrame представляет концепцию схемы для описания данных, позволяя Spark управлять схемой и передавать данные только между узлами гораздо более эффективным способом, чем при использовании сериализации Java
- API DataFrame кардинально отличается от API RDD поскольку это API для построения плана реляционных запросов, который затем может выполнить оптимизатор Sparks Catalyst. API является естественным для разработчиков, которые знакомы с построением планов запросов

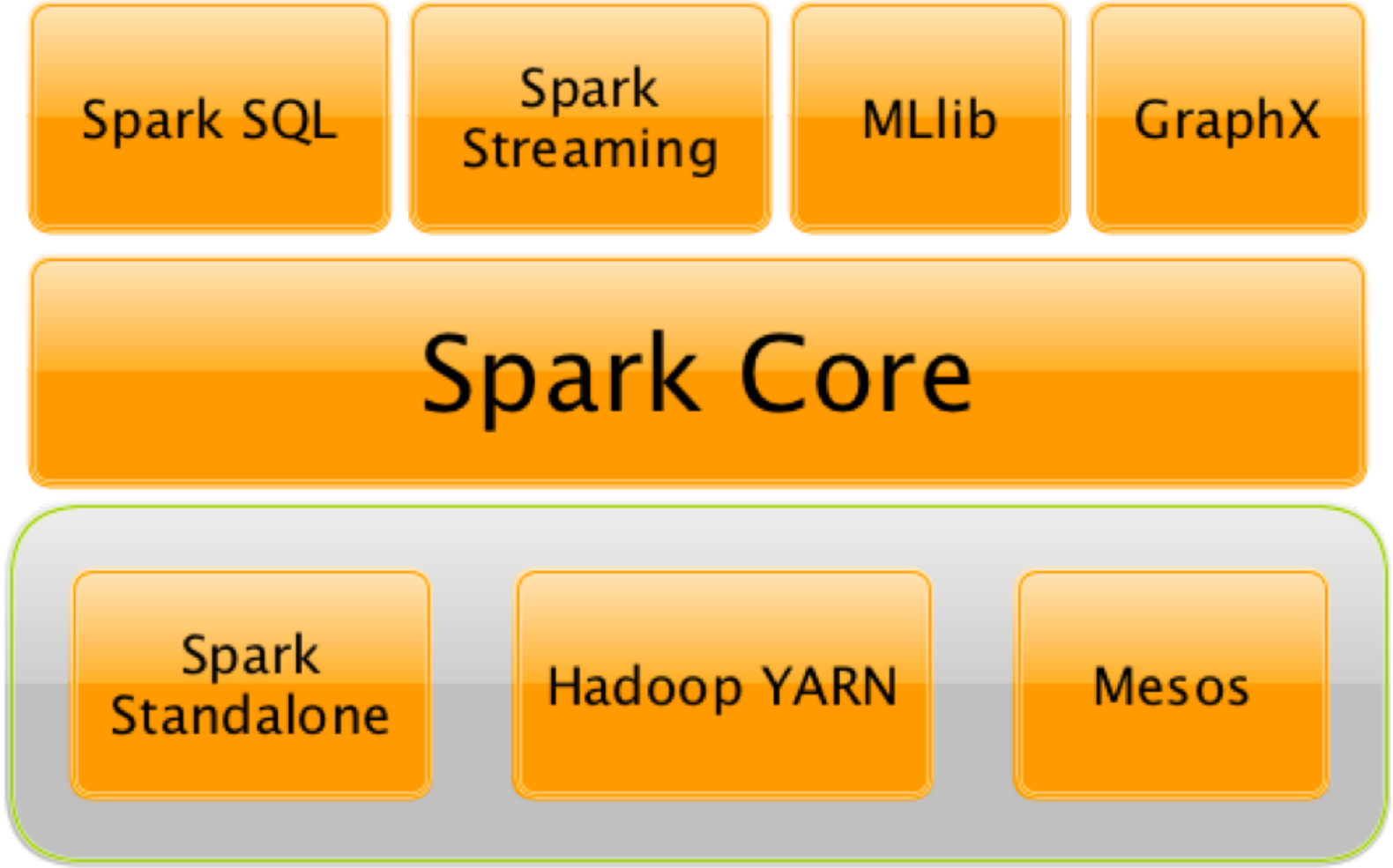
Dataset

Это расширение API DataFrame, обеспечивающее функциональность объектно-ориентированного RDD-API (строгая типизация, лямбда-функции), производительность оптимизатора запросов Catalyst и механизм хранения вне кучи. DataSet эффективно обрабатывает структурированные и неструктурированные данные, представляя их в виде строки JVM-объектов или коллекции. Для представления табличных форм используется кодировщик (encoder).

- Автоматически определяет схему данных
- Dataset обрабатывается быстрее RDD
- API-интерфейс Datasets обеспечивает безопасность времени компиляции за счет строгой типизации

DataFrame и Dataset при разработке





Модули Spark

Spark SQL

SQL движок сверху Spark, но не только. Служит для создания DataFrame, одними из основных его классов являются:

- SparkSession – точка входа для создания DataFrame и использования функций SQL
- DataFrame – распределенный набор данных, сгруппированных в именованные столбцы
- Column – столбец в DataFrame
- Row – строка в DataFrame
- GroupedData – агрегационные методы, возвращаемые DataFrame.groupBy()
- functions – список встроенных функций, доступных для DataFrame
- types – список доступных типов данных

и другие.

Spark Streaming

Модуль Streaming служит для доступа к функциональности потоковой передачи и является расширением основного API Spark, которое позволяет Data Scientist’ам обрабатывать данные в режиме реального времени из различных источников, включая (но не ограничиваясь) Kafka, Flume и Amazon Kinesis. Эти обработанные данные могут быть отправлены в файловые системы, базы данных или дэшборды.

В основе Streaming лежит DStream (Discretized Stream), который представляет поток данных, разделенный на небольшие пакеты RDD. Такие пакеты могут интегрироваться с любыми другими компонентами Spark, например, MLlib.

ML и MLlib (уже только MLlib)

В PySpark было два похожих модуля для машинного обучения – ML и MLlib. Они отличались только типом построения данных: ML использовал DataFrame, а MLlib — RDD. Поскольку DataFrame более удобен в работе, то теперь Mllib на DataFrame и осталось API для Mllib с RDD.

Модули машинного обучения богаты разными инструментами, а интерфейс схож со стандартом в лице scikit-learn. Что есть:

- Метрики и статистики
- Пайплайны
- Классика в классификации и регрессии
- Деревья
- Кластеризация
- Тематическое моделирования и иные методы для работы с текстами
- Рекомендательные системы
- Снижение размерности

GraphX

Модуль для работы с графами через RDD.

Стандартные узлы, связи.

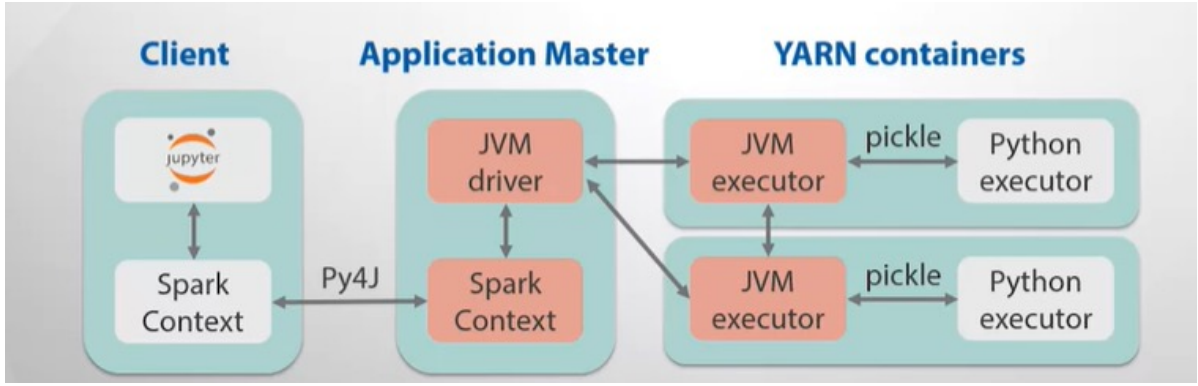
Что еще есть:

- PageRank
- Connected Components
- Triangle Counting
- Label propogation
- SVD++

PySpark

Pyspark on YARN

При создании в Python SparkContext запускается YARN приложение



PySpark – плата за Python

JVM executor передает данные процессу питона, тот их расшифровывает, что-то делает и отдает назад. Поэтому нативный Spark на java или scala быстрее.

Spark работает с Python объектами в сериализованном виде (pickle), они будут десериализованы для обработки в Python, поэтому элементы RDD занимают память два раза:

- В сериализованном виде в JVM
- Десериализованный объект в Python

То есть мы тратим:

- Память
- Время на сериализацию/десериализацию

Итог

- Программа на Spark – набор операций над RDD
- Вычисления в Spark ленивые, трансформации просто анализируются, чтобы посчитать надо делать actions
- SparkContext – точка входа для работы с Spark
- PySpark будет медленнее Spark и менее эффективным по памяти, но мы получаем возможность работать с Python объектами

Конфиги в Spark

Когда мы поднимаем SparkContext, то работоспособность программы будет сильно зависеть от конфига. Ниже описано многое, но не все (для остального есть Документация Spark).

- spark.app.name - имя приложения, отображаемое в UI логах
- spark.driver.cores – количество ядер, используемых в процессах драйвера. Используется в кластерном режиме.
- spark.driver.maxResultSize - ограничение общего размера сериализованных результатов всех разделов для каждого действия Spark (например, collect) в байтах. Задания будут прерваны, если общий размер превысит этот предел. Наличие высокого предела может привести к ошибкам нехватки памяти в драйвере (зависит от spark.driver.memory и накладных расходов на память объектов в JVM). Установка надлежащего предела может защитить драйвер от ошибок, связанных с нехваткой памяти. Но если это зарезервировать, то больше никому не достанется.
- spark.driver.memory - объем памяти, используемой для процесса драйвера, т. е. где инициализируется SparkContext.
- spark.executor.memory - объем памяти, используемой для каждого процесса исполнителя
- spark.local.dir - каталог, используемый для "нуля" пространства в Spark, включая выходные файлы карты и RDDS, которые хранятся на диске.
- spark.master - выбор менеджера ресурсов (yarn, local, mesos, kubernetes)
- spark.python.worker.memory – объем памяти, который будет использован процессом python в момент агрегации
- spark.python.worker.reuse – переиспользовать или нет python worker. Если да, то будет использовано фиксированное число workers, не переподнимая их постоянно.

- spark.ui.port – порт UI
- spark.serializer – сериализатор объектов. По умолчанию org.apache.spark.serializer.JavaSerializer, он может все, но долго. Лучше использовать org.apache.spark.serializer.KryoSerializer
- spark.default.parallelism – стандартное количество партиций, возвращаемое после трансформаций типа join, reduceByKey, а также количество партиций для параллельной обработки.
- spark.files.maxPartitionBytes - максимальное количество байт для упаковки в партицию при считывании файла
- spark.network.timeout - стандартное время ожидания ответа по сети
- spark.dynamicAllocation.enabled - следует ли использовать динамическое распределение ресурсов, которое увеличивает и уменьшает количество исполнителей, зарегистрированных в этом приложении, в зависимости от рабочей нагрузки. Не забирает все ресурсы у соседа, если не нужно.
- spark.shuffle.service.enabled – если наш job больше не требует большого количества ресурсов – верни их в общий пул при самой затратной операции shuffle.
- spark.sql.broadcastTimeout – таймаут для ожидания в broadcast join
- spark.yarn.executor.memoryOverhead - объем оперативной памяти (в мегабайтах), выделяемый каждому исполнителю
- spark.executor.cores - количество ядер, используемых на каждом исполнителе
- spark.executor.instances - сколько исполнителей выделить
- spark.dynamicAllocation.initialExecutors – сколько при динамической аллокации выделять исполнителей в самом начале
- spark.dynamicAllocation.minExecutros – сколько минимально можно использовать исполнителей для задачи при динамической аллокации
- spark.dynamicAllocation.maxExecutros – сколько максимально можно использовать исполнителей для задачи при динамической аллокации

Spark.master

- local – поднимет локальный спарк с 1 ядром
- local[*] – локальный спарк со всеми ядрами
- local[10] – локальный спарк и 10 ядер
- yarn – использовать YARN на кластере

Собираем конфиг

Тут мы соберем свой конфиг, начнем с базовых вещей, которые не зависят от мощностей, а далее разберем, как сделать остальные кастомные настройки.

Таким образом план:

- Набираем те конфиги, которые лучше использовать сразу;
- Обсуждаем, как работает под капотом распределение ресурсов и постановка задач;
- Кастомизуем конфиг исходя из новых знаний;
- Обсуждаем, в каком направлении можно двигаться дальше.

Некоторые моменты

- Все начинается с конфига и многое зависит от него
  - Мы уже знающие и помним, что pyspark сериализует и десериализует объекты python. По умолчанию используется JavaSerializer, но намного быстрее KryoSerializer, поэтому spark.serializer лучше ставить с org.apache.spark.serializer.KryoSerializer + задать размер буфера spark.kryoserializer.buffer.max, если вдруг не хватит памяти (редкость), по умолчанию 64m, должно быть не больше 2048m.
  - Динамическая аллокация ресурсов spark.dynamicAllocation.enabled = True хорошая вещь, но работает на эвристиках и при резком росте ресурсов может упасть. В этом случае ручная настройка памяти драйвера, executor’а, размера максимального результата помогут. Если используете динамику, не забывайте про set('spark.shuffle.service.enabled', 'true’) для контроля и возврата неиспользуемых ресурсов.
  - Помните, что если задаете статические параметры ресурсов, то на них динамическая аллокация не распространяется.

Дефолтные конфиги

Что точно берем?

- 'spark.serializer’ – сериализация;
- 'spark.dynamicAllocation.enabled’ – динамическая аллокация;
- 'spark.shuffle.service.enabled’ – возврат ресурсов после динамики;
- 'spark.ui.port’ – порт UI, помним, что не все порты могут быть досутпны.

Получаем:

```
from pyspark.sql import SparkSession
from pyspark import SparkContext, SparkConf

conf = SparkConf().set('spark.ui.port', '4050’)\
                  .set('spark.app.name’, ‘My application’ )\
                  .set('spark.serializer', 'org.apache.spark.serializer.KryoSerializer’)\
                  .set('spark.dynamicAllocation.enabled', 'true’)\
                  .set('spark.shuffle.service.enabled', 'true')
```

Дефолтные конфиги

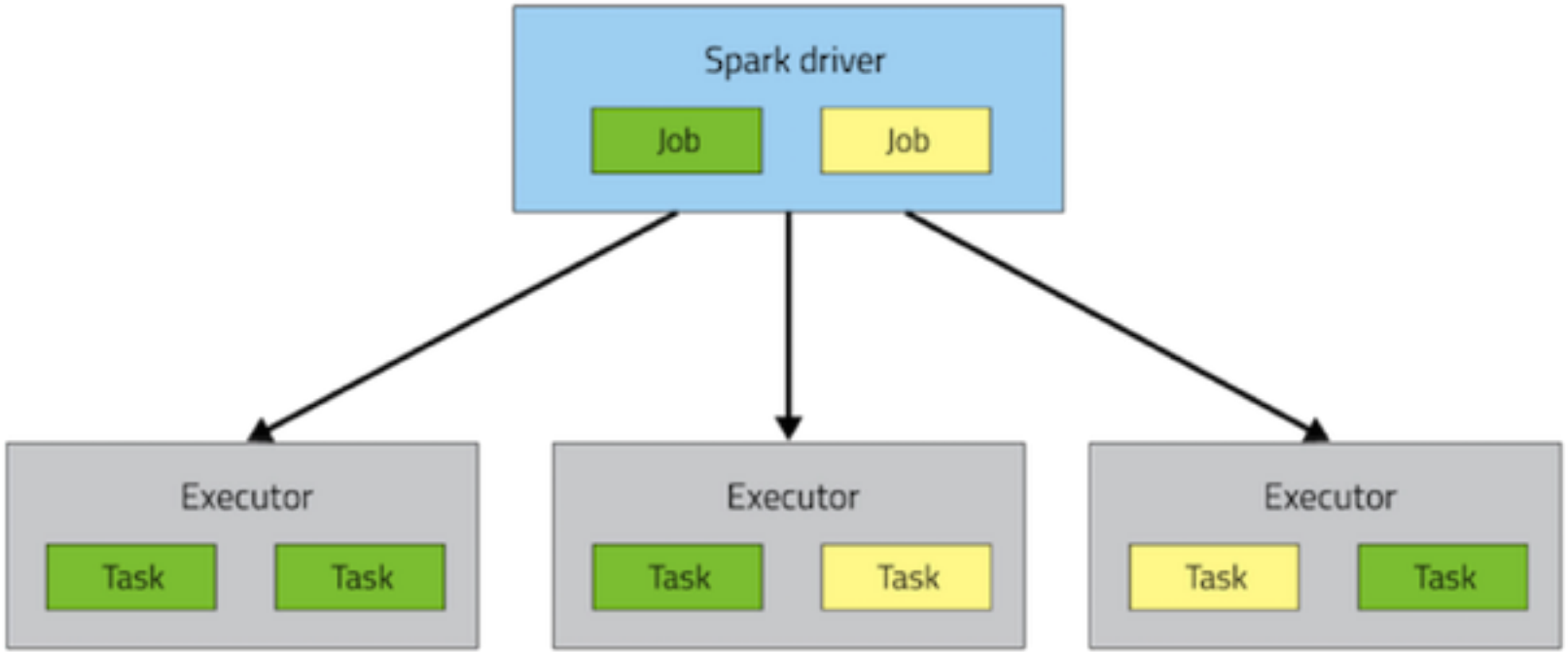
- Сериализатор сначала использует небольшой объем памяти, если не хватает, то увеличивает его до максимума spark.kryoserializer.buffer.max. По умолчанию это 64m, максимум можно поставить 2048m. Попробуйте ставить 256m, далее уже от задачи.

```
conf = SparkConf().set('spark.ui.port', '4050')\
                  .set('spark.app.name', 'My application')
                  .set('spark.serializer', 'org.apache.spark.serializer.KryoSerializer')\
                  .set('spark.dynamicAllocation.enabled', 'true')\
                  .set('spark.shuffle.service.enabled', 'true')\
                  .set('spark.kryoserializer.buffer.max', '256m’)
```

И этого уже будет достаточно для решения огромного числа задач! Но что если мы хотим сами настроить, а может и улучшить?

Можно, причем многое, особенно в части разделения ресурсов.

Что под капотом



Драйвер определяет задания, говорит кому что делать, а executor выполняет выданную от driver работу.
Драйвер обеспечивает высокоуровневое управление рабочим процессом.

Что под капотом

- Каждому исполнителю (executor) в приложении Spark выделяется одинаковое фиксированное количество ядер и одинаковый фиксированный размер памяти;
- Один executor может выполнять несколько задач параллельно, если у него более одного ядра в распоряжении;
- Application master, является невыполняющим контейнером (non-executor container) и обладает специальной возможностью запрашивать контейнеры у YARN, в процессе своей работы также потребляет ресурсы, которые необходимо учитывать. Это наш driver.
- Выделение исполнителям слишком большого объема памяти обычно приводит к чрезмерным задержкам в процессе сборки мусора (garbage collection). Примерно 64 ГБ – оптимальный верхний предел для одного исполнителя.
- У HDFS-клиента возникают проблемы при большом количестве одновременно выполняющихся потоков. Как максимум пять задач на исполнитель позволяют использовать всю пропускную способность записи, поэтому необходимо, чтобы количество ядер на исполнитель было меньше этого значения.
- Применение «маленьких» исполнителей (например, с одним ядром и объемом памяти, достаточным для запуска только одной задачи) лишает преимуществ от выполнения нескольких задач на одной JVM;
- Collect и прочие вещи отправляют данные на драйвер.

Выделяем ресурсы

Пусть у нас есть одна тачка со следующими параметрами:

- 16 ядер
- 128 Gb оперативной памяти

Важно! Мы не выделяем 100% ресурсов для контейнеров YARN, потому что на каждом узле необходимо зарезервировать некоторые ресурсы для работы операционной системы и демонов Hadoop!

Словарь:

- spark.executor.instances – сколько executor’ов у нас будет
- spark.executor.cores - сколько ядер у одного исполнителя
- spark.driver.cores – количество ядер драйвера
- spark.driver.memory – объем памяти драйвера
- spark.executor.memory - объем памяти исполнителя

Ну что, настраиваем?

Выделяем ресурсы

1 ядро и немного памяти надо оставить для для драйвера, а все остальное отдаем исполнителям, но сделать 1 исполнителя с 15 ядрами плохо, вспоминаем предыдущие слайды. Хотя может иногда быть полезным.

Лучше всего начать с такого:

- 1 ядро на spark.driver.cores
- 3 исполнителя spark.executor.instances
- 5 ядер у каждого исполнителя spark.executor.cores
- Память делим честно 32 Гб каждому исполнителю spark.executor.memory и 32 отдадим драйверу spark.driver.memory, но на самом деле драйверу будет много, если вы не будете отдавать кучу данных ему.

А если кластер и 10 тачек? Ну тогда конфигурируем одну машину, количество исполнителей умножаем на 9 (1 тачку не трогаем).

Выделяем ресурсы

```
conf = SparkConf().set('spark.ui.port', '4050')\
                  .set('spark.app.name', 'My application')
                  .set('spark.serializer', 'org.apache.spark.serializer.KryoSerializer')\
                  .set('spark.dynamicAllocation.enabled', 'true')\
                  .set('spark.shuffle.service.enabled', 'true')\
                  .set('spark.kryoserializer.buffer.max', '256m')\
                  .set('spark.executor.memory', '32G')\
                  .set('spark.executor.instances', '3')\
                  .set('spark.executor.cores', '5')\
                  .set('spark.driver.cores', '1')\
                  .set('spark.driver.memory', '32G')
```

Куда двигаться дальше

Чаще всего хватит и просто динамической аллокации, но она работает на эвристиках и бывает падает при резком увеличении требований, а также может забрать все, что есть:

- Корректируйте размер памяти драйвера и исполнителя
- Корректируйте размер буфера сериализатора
- Ограничивайте аппетиты динамической аллокации
- spark.driver.maxResultSize – фиксируйте сразу максимальный объем оперативной памяти для результатов драйвера
- spark.default.parallelism – корректируйте число партиций

И иные вещи, связанные с python workers и так далее.

Рекомендуемый конфиг (ЦРМ)

```
conf = SparkConf().setAppName(app_name)\
    .setMaster("yarn")\
    .set('spark.dynamicAllocation.enabled', 'True')\
    .set('spark.executor.memory','48g')\
    .set('spark.driver.memory','48g')\
    .set('spark.yarn.executor.memoryOverhead','36g')\
    .set('spark.driver.maxResultSize','512g')\
    .set('spark.executor.cores', '15')\
    .set('spark.driver.maxResultSize','512g')\
    .set('spark.dynamicAllocation.initialExecutors', 1)\
    .set('spark.dynamicAllocation.minExecutors', 1)\
    .set('spark.dynamicAllocation.maxExecutors’, 5)\
    .set('spark.port.maxRetries', '150')\
    .set('spark.driver.cores', '1')\
    .set('spark.yarn.driver.memoryOverhead', '128g')\
    .set('spark.shuffle.service.enabled', 'true')\
    .set('spark.sql.legacy.parquet.datetimeRebaseModeInRead', 'CORRECTED')\
    .set('spark.kryoserializer.buffer.maxValue', '2044018')\
    .set('spark.sql.autoBroadcastJoinThreshold', -1)\
    .set('spark.sql.broadcastTimeout', -1)\
    .set("spark.sql.catalogImplementation", "hive")
```

Тут 15 ядер на исполнителя, так как это оптимально для всех задач

Динамика, максимум 5 исполнителей на человека

Запрет на broadcast, так как spark может переоценить свои силы, а мы пытаемся собрать универсальную

Сразу стоит оговориться, что это попытка сделать универсальный для всех конгфиг, который потянет большинство задач