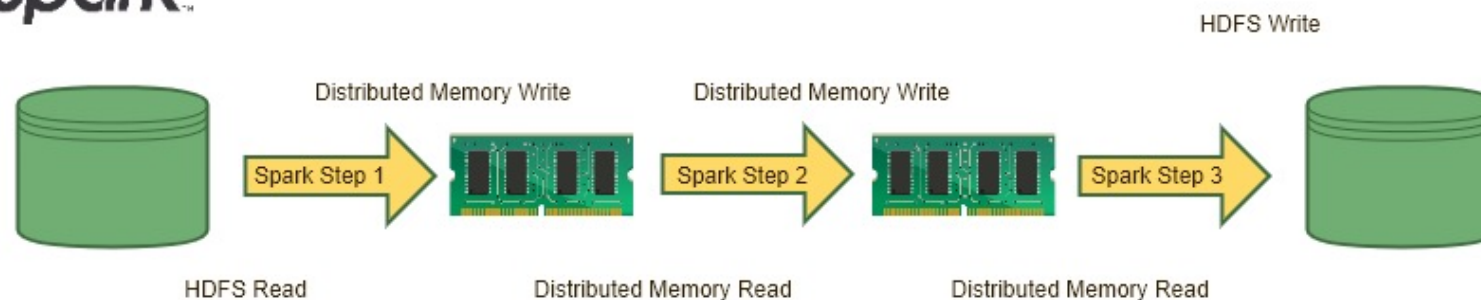




Apache Spark

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 фреймворк 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 на примере

- 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

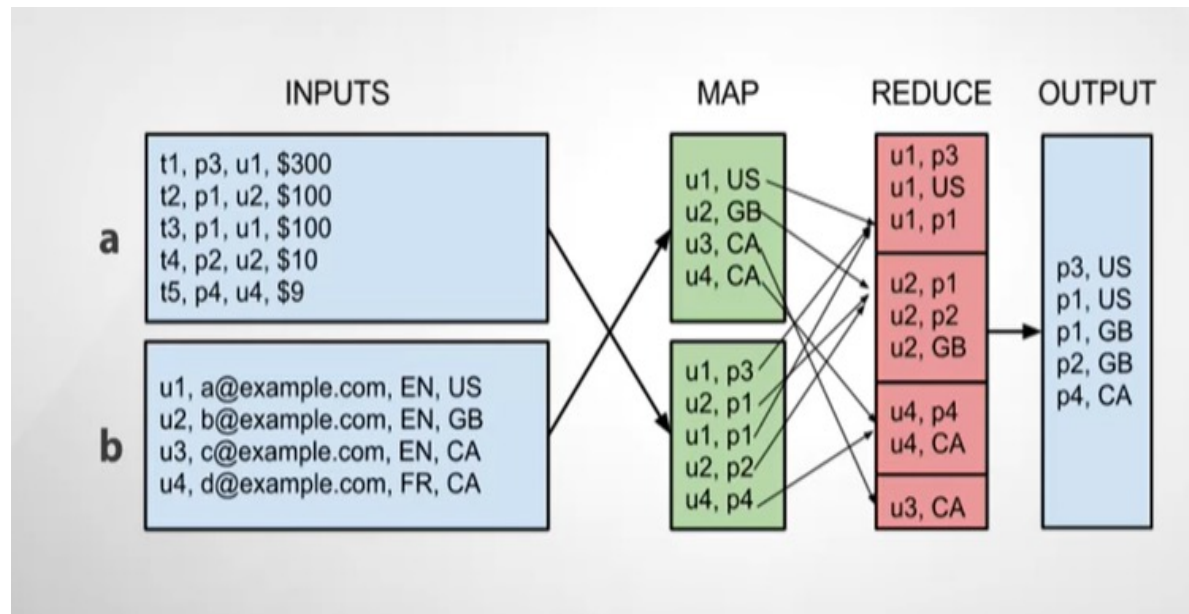
Что-то вроде такого:

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

Ну и дальше можно накидывать всякие group by.

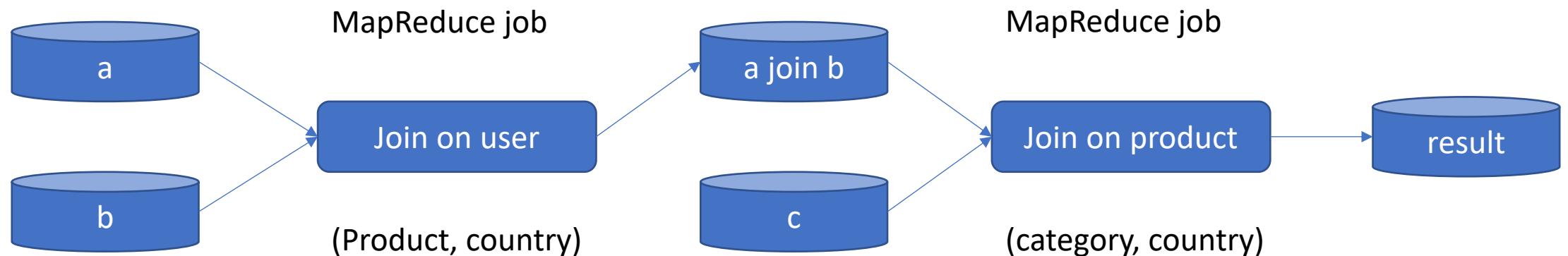
SQL join on MapReduce

- 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

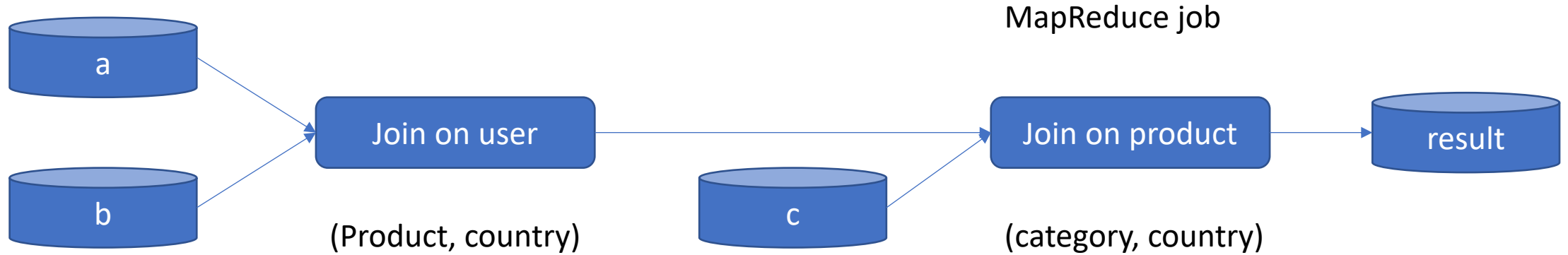
- Есть таблица с, в которой лежит информация о продуктах (product, type)
- Визуализируем join



- 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.



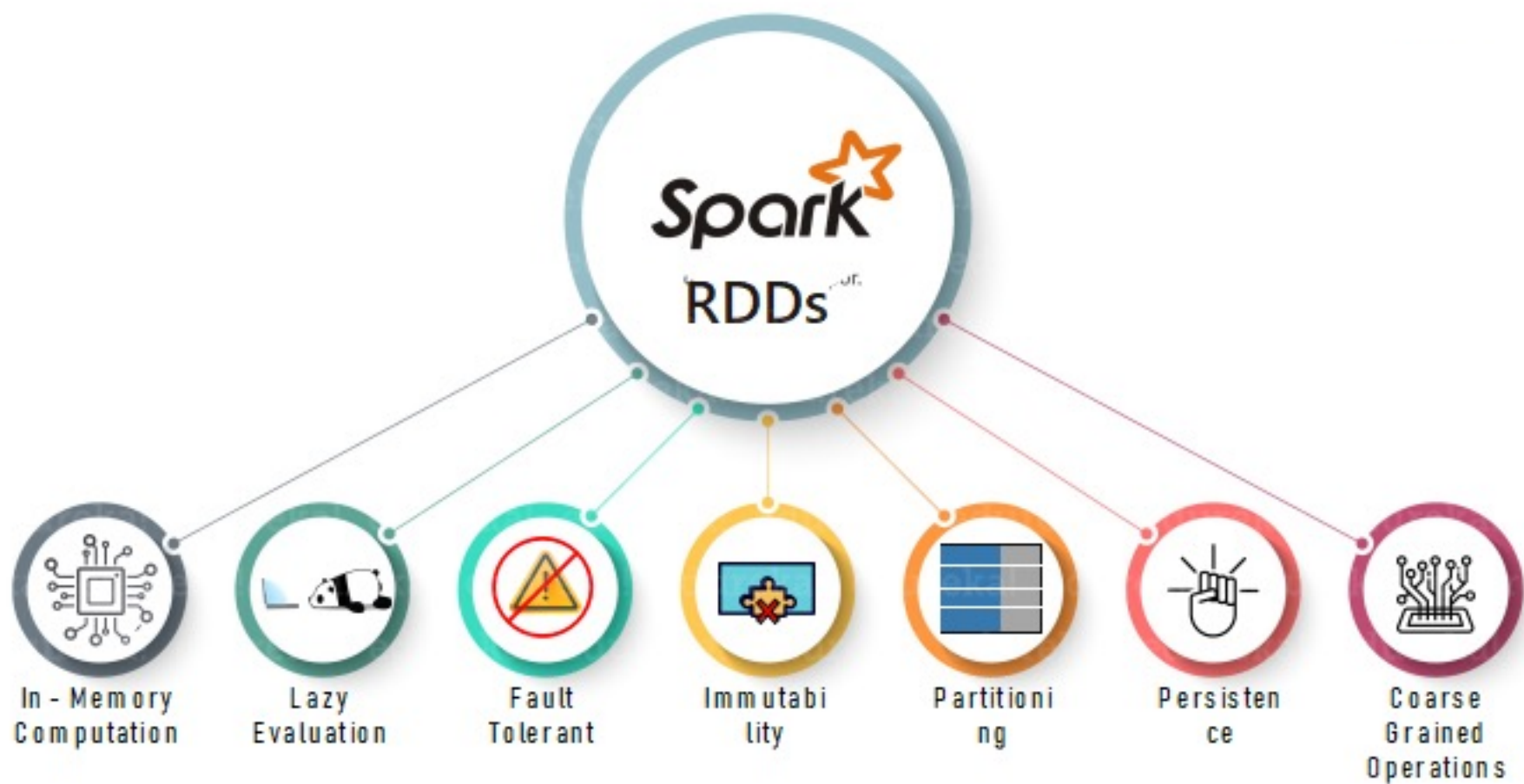
- Все вычисления описываются как DAG (Directed Acyclical Graph)
- Промежуточный результат хранится в памяти или на диске, минуя HDFS

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 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.

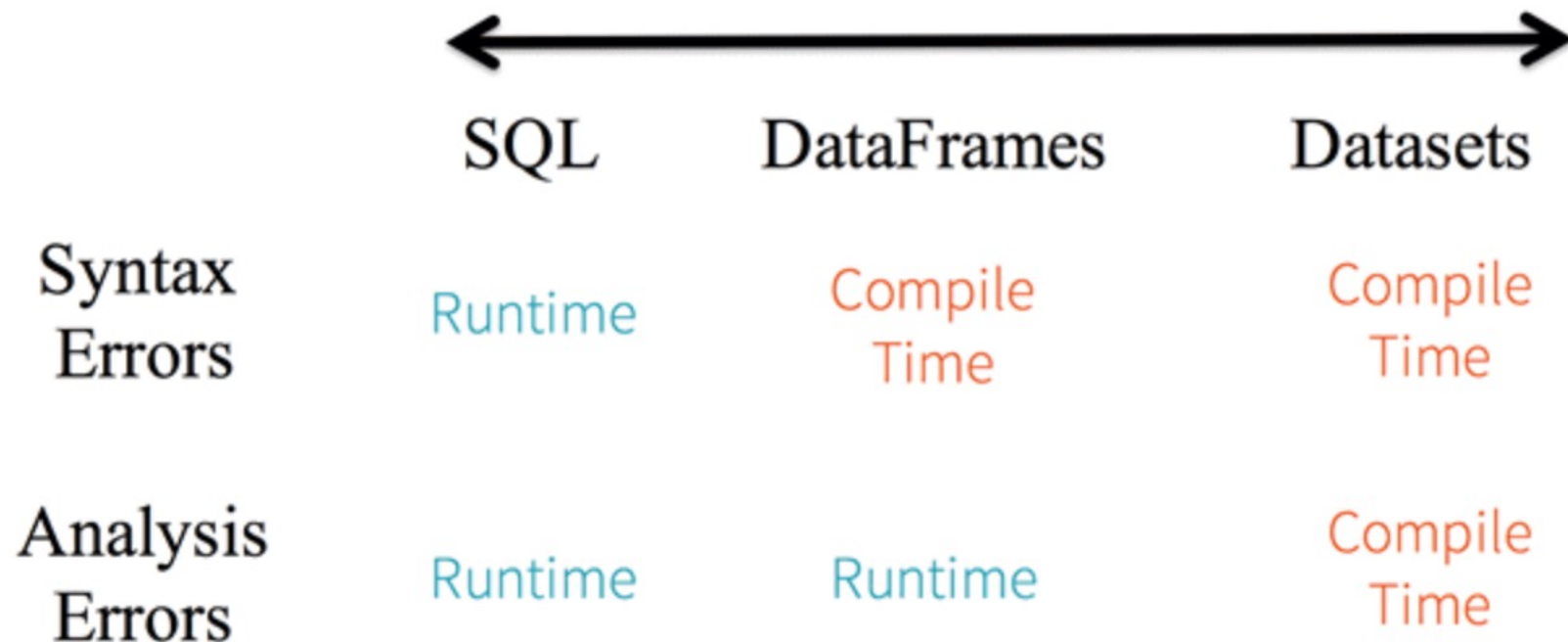
- Автоматически определяет схему данных
- DataFrame обрабатывается быстрее RDD
- DataFrame представляет концепцию схемы для описания данных, позволяя Spark управлять схемой и передавать данные только между узлами гораздо более эффективным способом, чем при использовании сериализации Java
- API DataFrame 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

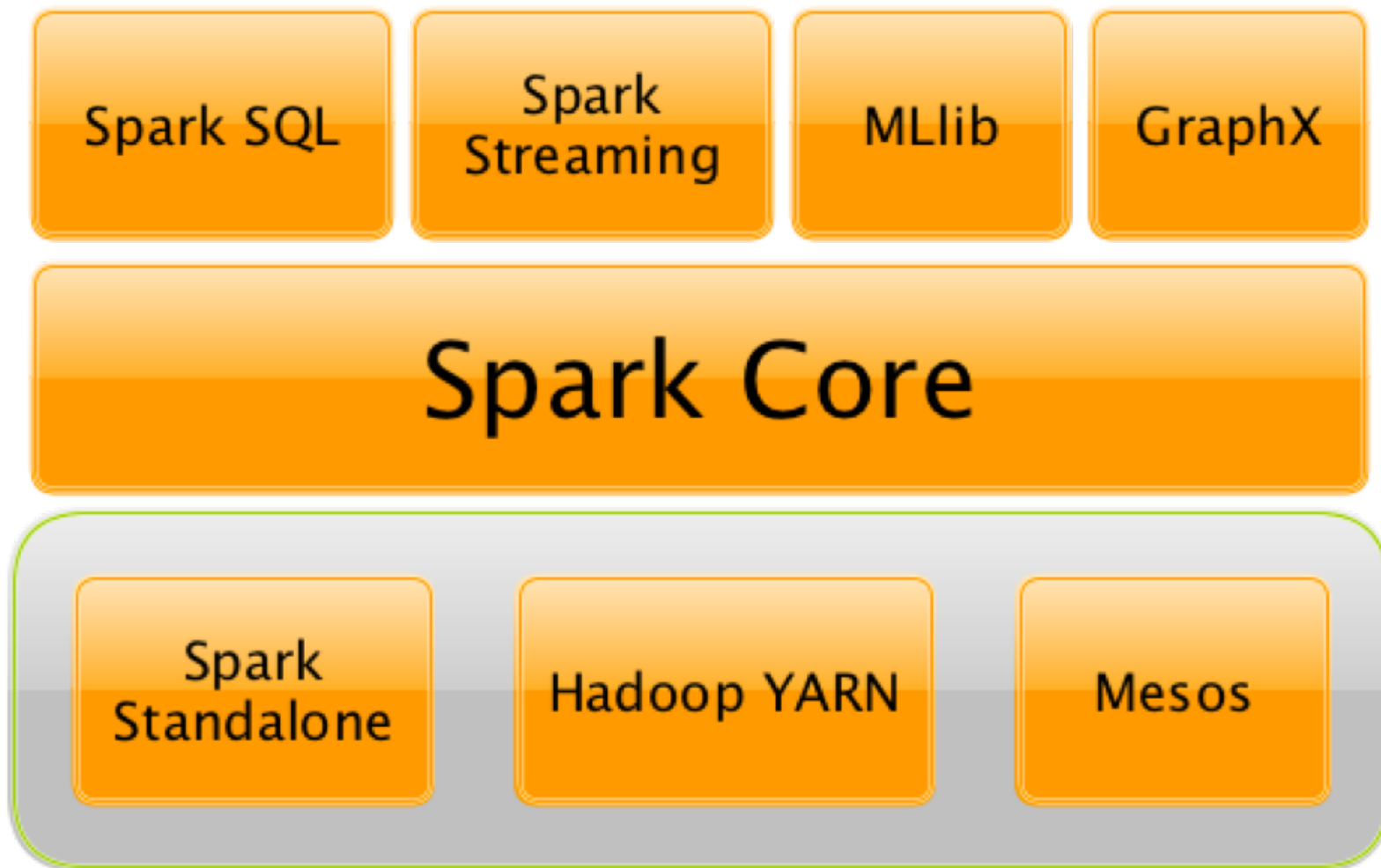
Dataset

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 с точки зрения разработки





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:

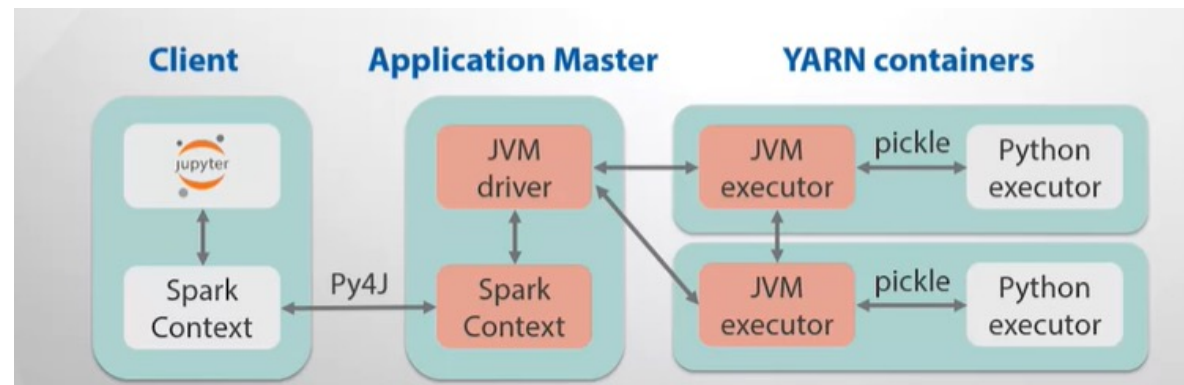
- PageRank
- Connected Components
- Triangle Counting
- Label propagation
- SVD++



PySpark

PySpark на 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.

- `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 требует большого количества ресурсов – верни их в общий пул при замой 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 тачек занять на кластере

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