СБЕР & ИТМО

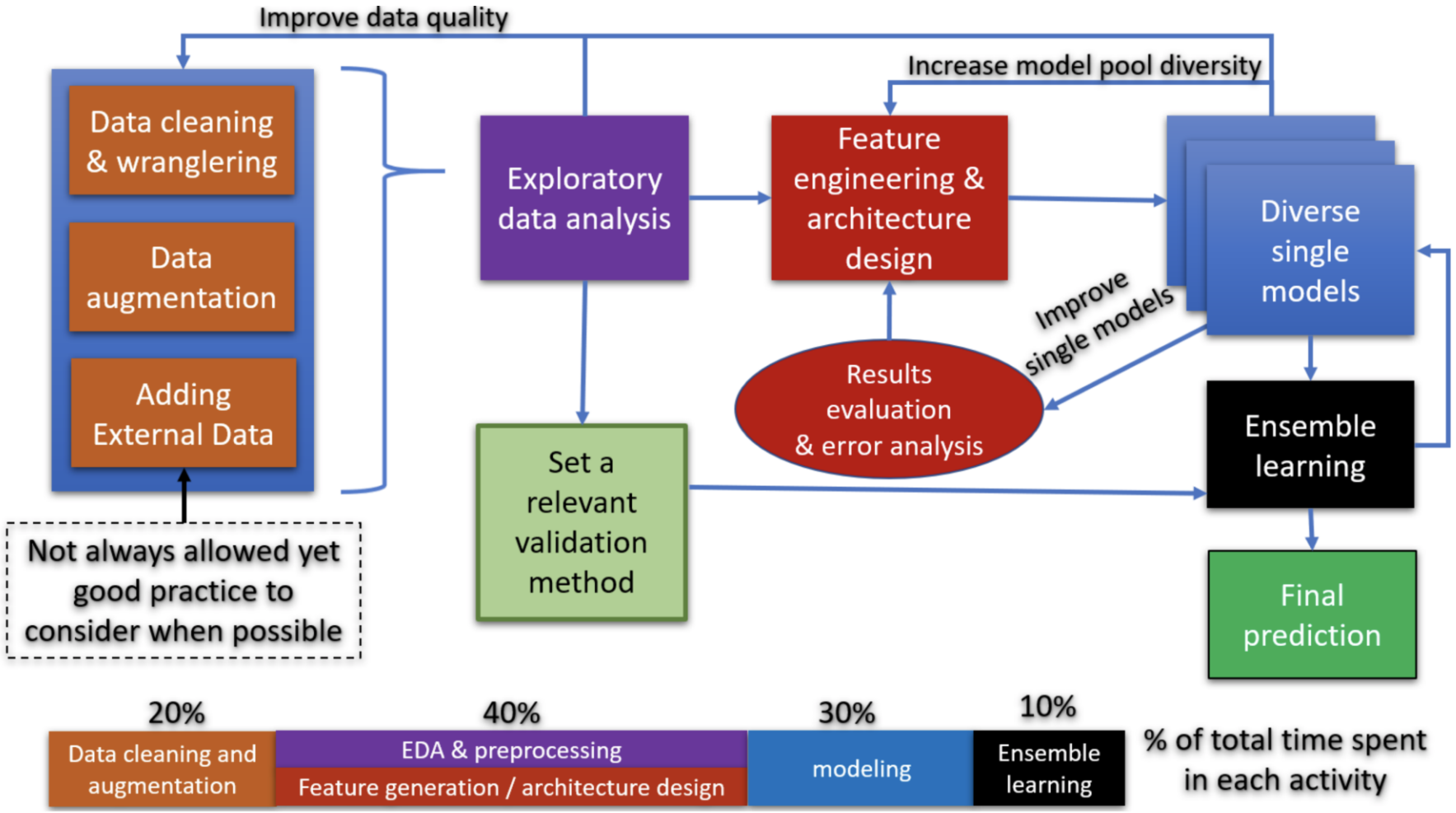
AutoML на Spark: миф, ставший реальностью

Рыжков Александр, Sber AI Lab

Бутаков Николай, университет ИТМО

- 10.10.2023

Типичный pipeline решения ML задач



- **Довольно сложно…** А есть ли способ решить задачу машинного обучения проще?

Что такое AutoML?

- **AutoML в узком смысле** – инструмент для автоматического решения задачи машинного обучения
- **AutoML в широком смысле** – технология, помогающая автоматизировать весь процесс моделирования и другие этапы, включая:
  - подготовку данных/feature engineering
  - отбор признаков
  - построение модели, оптимизация параметров
  - валидацию/построение отчетов
  - внедрение и инференс моделей в проме/мониторинг

Open source решения:

- **LightAutoML (Sber AI Lab)**
- H2O AutoML (H2O.ai)
- AutoGluon (Amazon)
- TPOT
- AutoSklearn

…

Проприетарные решения:

- H2O Driveless AI
- Google Cloud AutoML
- IBM AutoAI
- Microsoft Azure AutoML

…

Мотивация разработки собственного AutoML

Требуется AutoML, который поддерживает разные модальности – числа, категории, даты, тексты, изображения и т.п.

Часто датасет бывает не одной таблицей, а набором связанных таблиц – нужно уметь собрать плоский датасет

Необходимость построения интерпретируемых моделей

Решение ML задачи – не только модель, но и отчет о разработке, интерпретация, интеграция со средами инференса (ML Space в SberCloud), мониторинг и т.п.

Разные типы задач – классификация (бинарная/мультикласс), регрессия, uplift моделирование, multilabel и т.д.

Быстрое решение большого количества ML подзадач единой бизнес постановки – brute-force подход слишком медленный (ориентир на «модель за 10 минут с качеством среднего DS-а на midrange железе»)

Максимальная кастомизация выстраиваемых pipeline-ов, в том числе с написанием своих модулей для SOTA алгоритмов, при едином внешнем интерфейсе

Высокая технологичность и эффективность решения – запуск решения на GPU и Spark, использование данных из БД

LightAutoML (LAMA)

- **Кросс-платформенный модульный фреймворк** включающий в себя набор пресетов пайплайнов для решения **end-to-end** типовых задач, а также возможность разработки кастомных пресетов на разном уровне абстракции

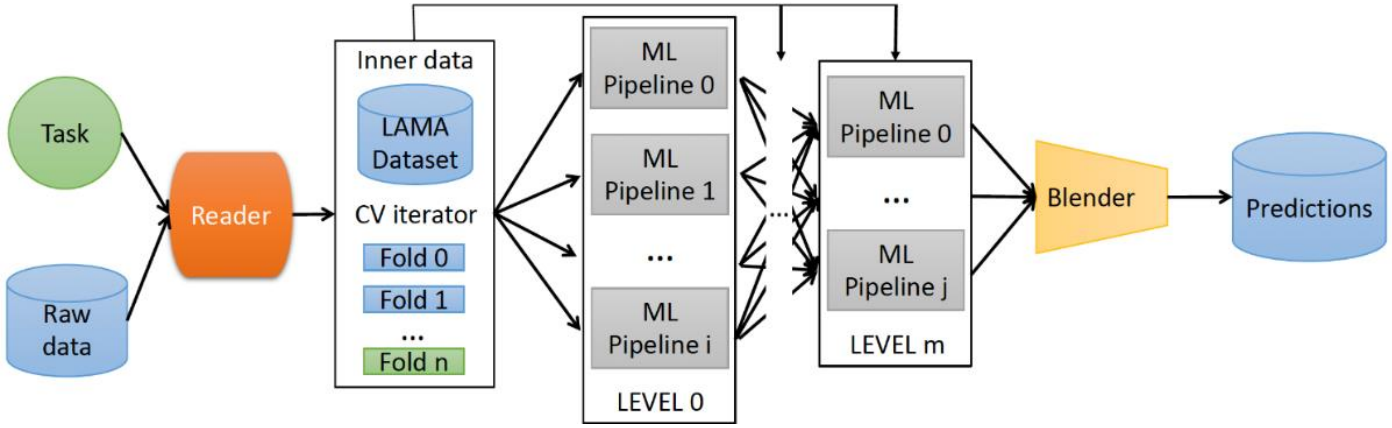
- **Целевая аудитория** – разработчики и DS

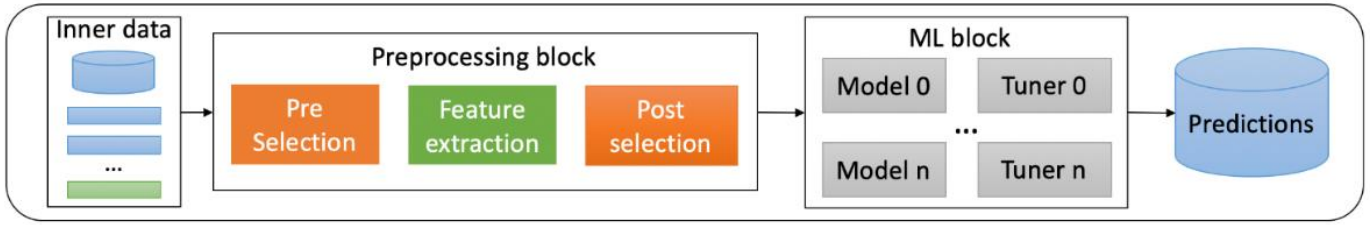
- **На сегодняшний день реализованы**
  - ✓ BlackBox preset
  - ✓ WhiteBox preset
  - ✓ NLP preset

GitHub

830 + 592

157 тыс.
Скачиваний LightAutoML



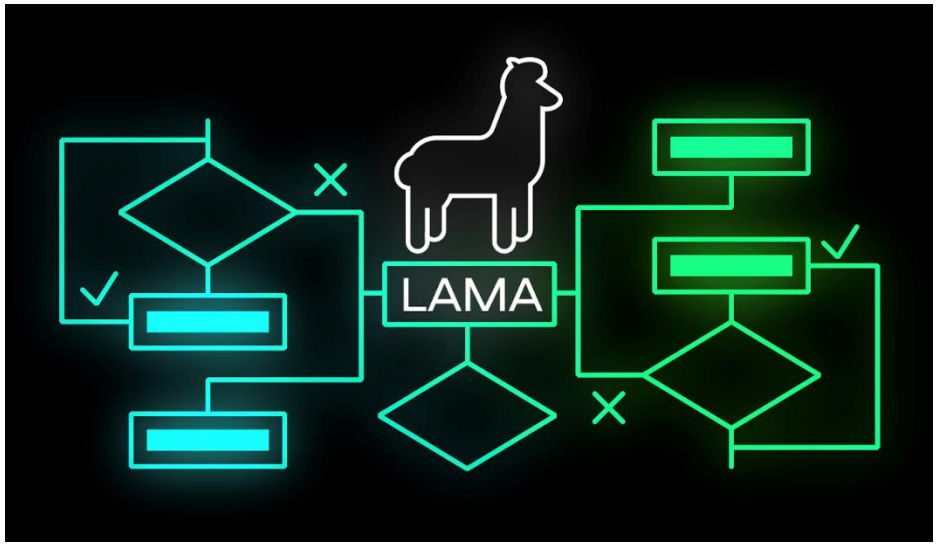


https://github.com/sb-ai-lab/LightAutoML

SBER AI Lab

Что такое SLAMA?

- Это распределенная версия библиотеки LightAutoML (LAMA) под Spark 3+ для обработки больших датасетов
- Горизонтально масштабируется за счет разделения датасета между несколькими машинами
- Работает в кластерных средах Kubernetes / YARN / Spark Cluster
- **Пока** реализована только часть функциональности LAMA по работе с табличными данными: Tabular Preset
- Поддерживает алгоритмы: LinearLGBFS и boosting (lightgbm)
- Новый гибридный data/compute parallel режим для повышения эффективности обработки средних по размеру датасетов

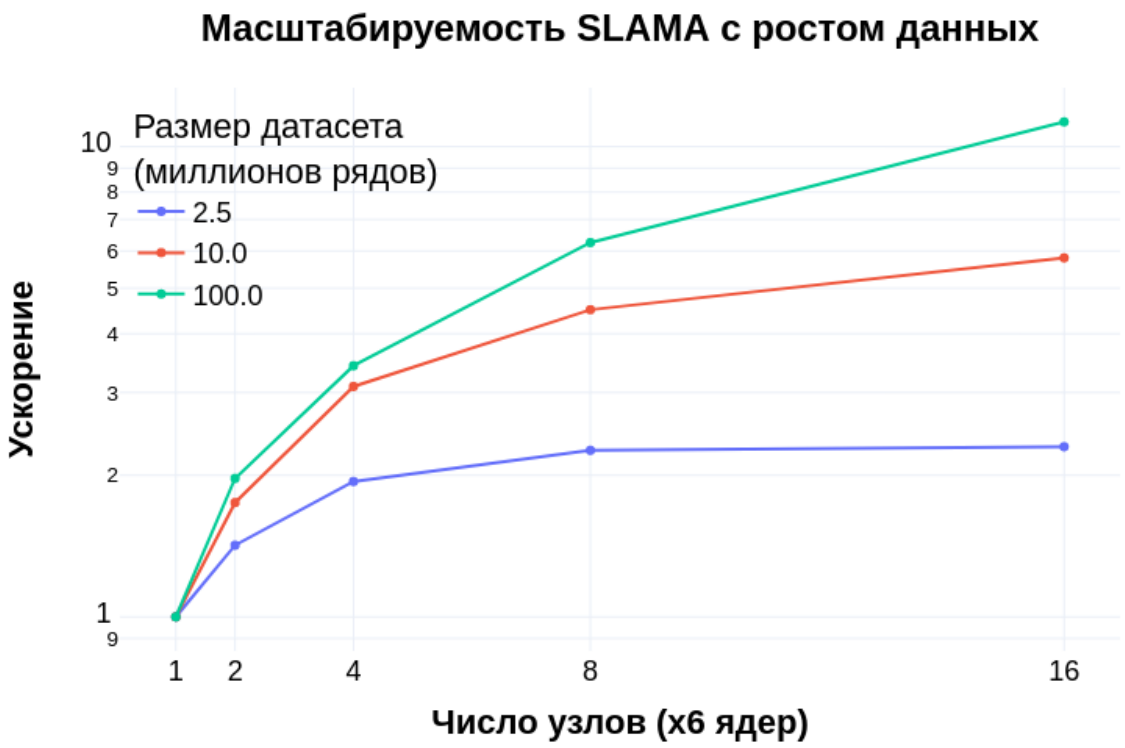


APACHE Spark™

https://github.com/sb-ai-lab/SLAMA

ИТМО

Общая масштабируемость



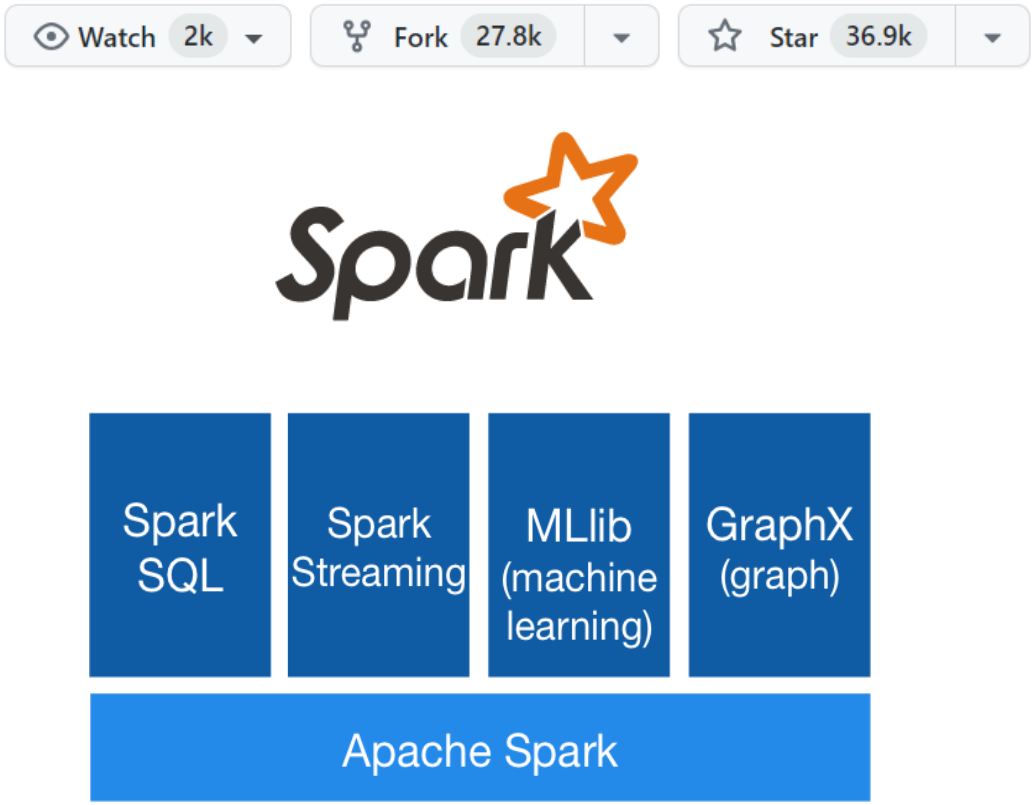
- При добавлении новых узлов (экзекьютеров) получаем ускорение выполнения
- С ростом числа узлов ускорение не всегда кратно количеству добавляемых узлов – мешают синхронизации на итерациях в алгоритмах, некоторые линейные элементы в пайплайнах. Упираемся в закон Амдала.
- При относительно малом количестве данных выходим на плато – нет смысла добавлять больше узлов
- Но чем больше данных, тем ближе мы подходим к линейной масштабируемости, тем больше смысла имеет добавление узлов

Больше инфраструктуры – большего размера датасеты можем обработать и быстрее получаем результат.

И с ростом данных масштабируемость растет.

Что такое Spark?

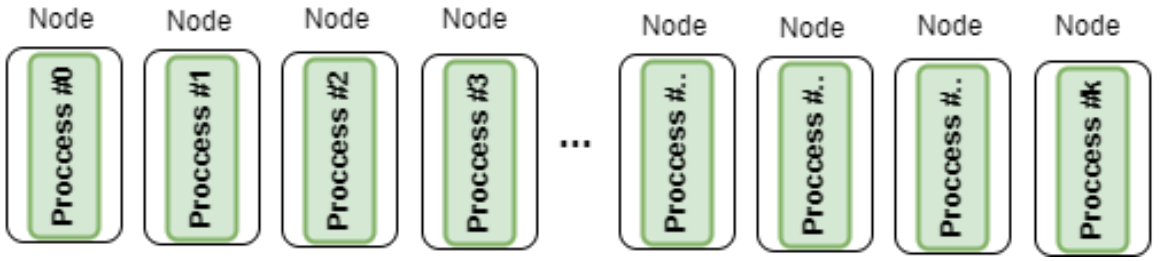
Что такое Spark?



https://spark.apache.org/

- Фреймворк для написания распределенных приложений обработки данных
- Одно из наиболее популярных решений для BigData в кластерных средах.
- Доступен для Java/Scala/Python/R
- Позволяет юзерам писать свои UDF (user-defined function) на Java/Scala/Python/R
- Поддерживает диалект SQL
- Предоставляет стандартную библиотеку для анализа данных и ML: Spark ML

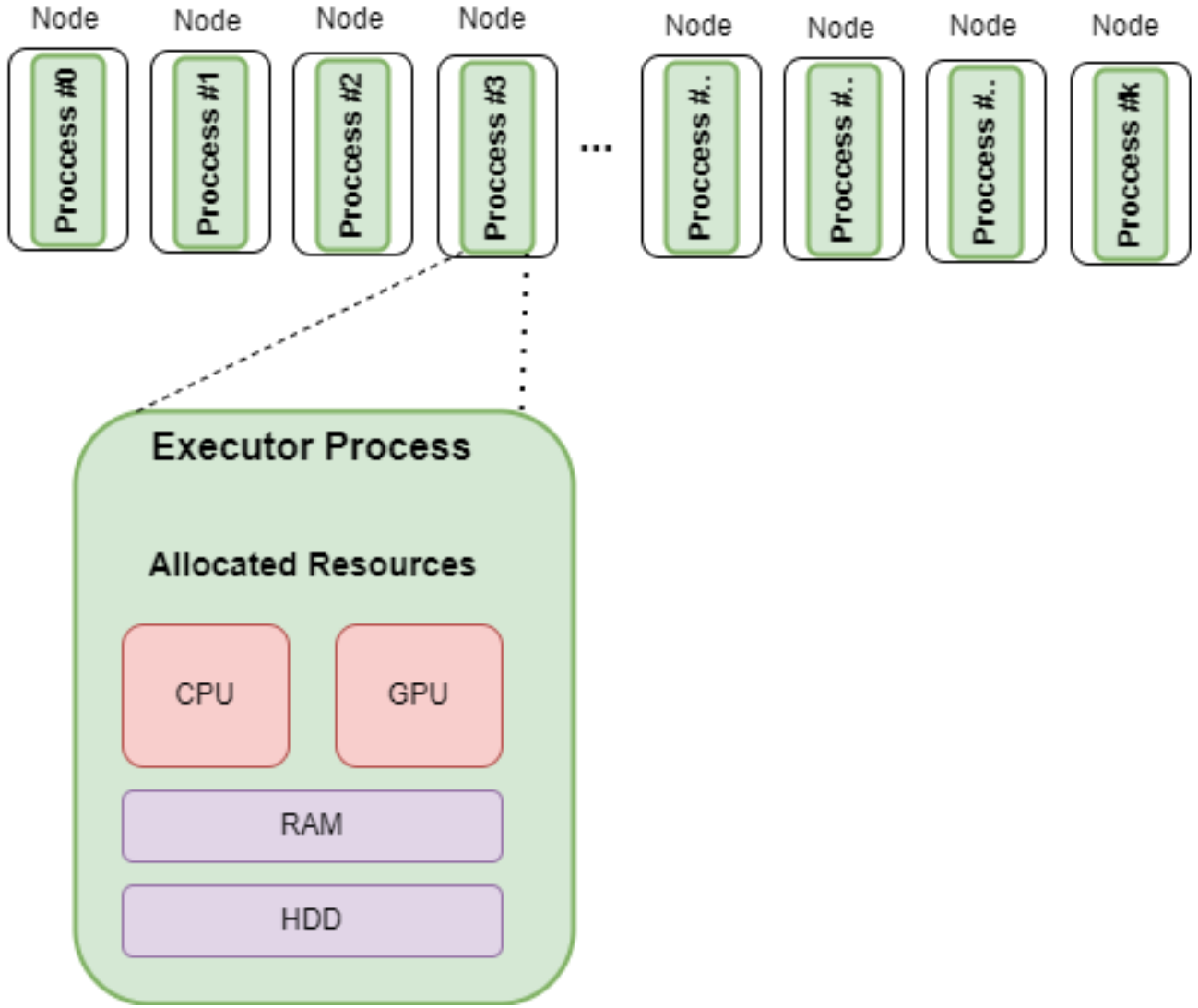
Из чего состоит Spark приложение ?



- Приложение Spark – набор процессов на узлах кластера.

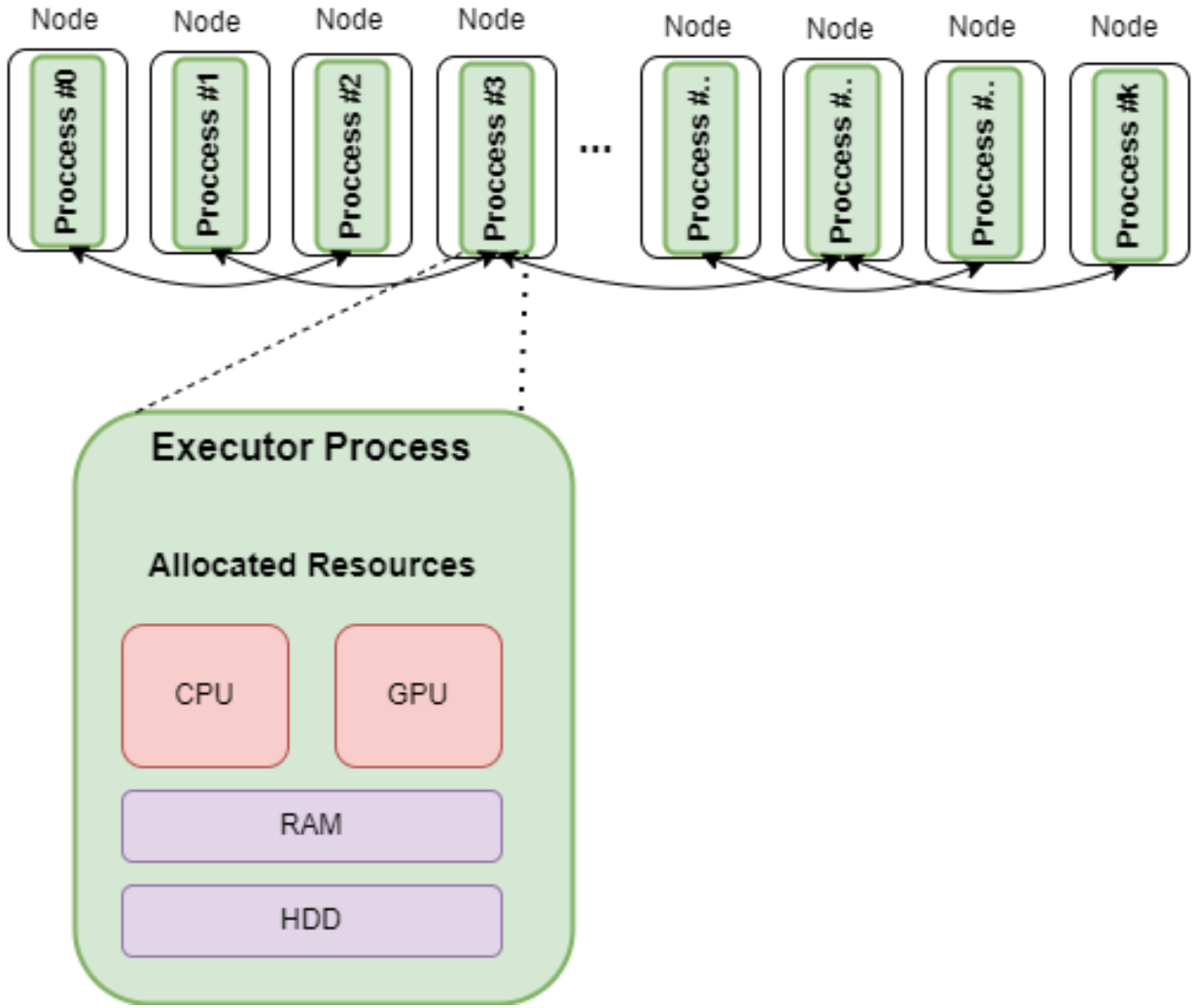
Из чего состоит Spark приложение ?

- Приложение Spark – набор процессов на узлах кластера.
- Процессы экзекьютеры обладают ресурсами (CPU, RAM, etc).



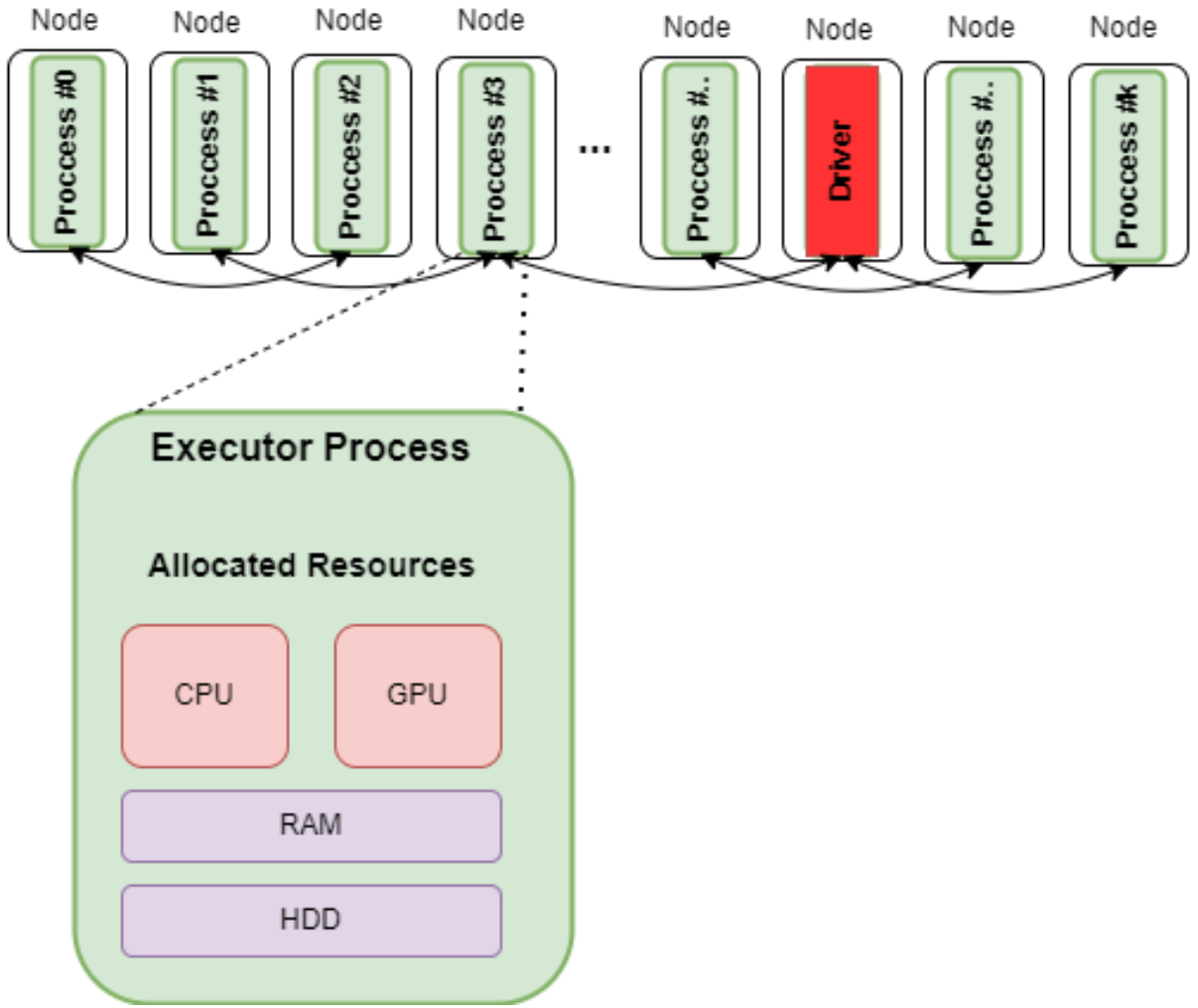
Из чего состоит Spark приложение ?

- Приложение Spark – набор процессов на узлах кластера.
- Процессы экзекьютеры обладают ресурсами (CPU, RAM, etc).
- Процессы коммуницируют друг-с-другом напрямую (Netty) и через torrent-протокол



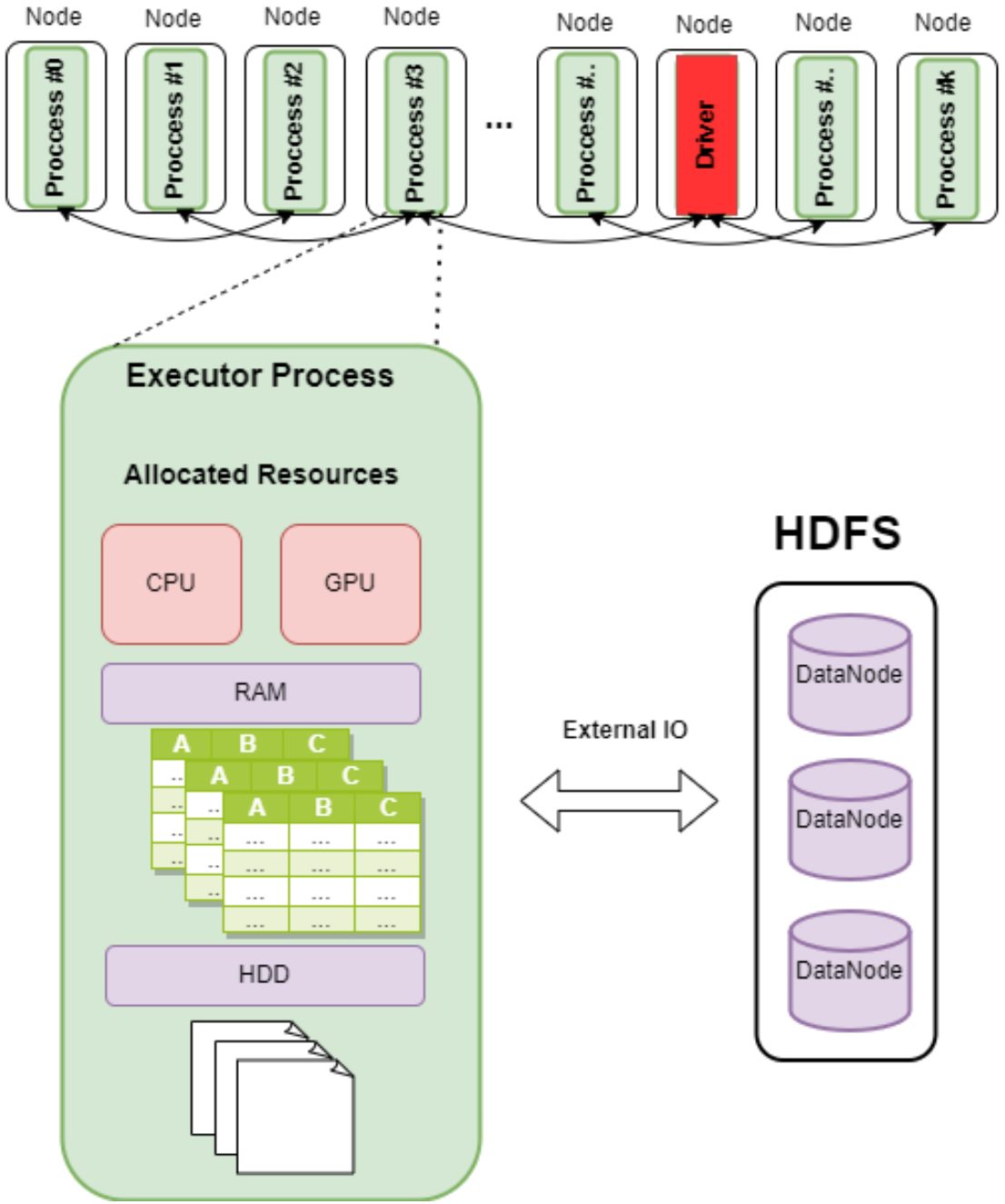
Из чего состоит Spark приложение ?

- Приложение Spark – набор процессов на узлах кластера.
- Процессы экзекьютеры обладают ресурсами (CPU, RAM, etc).
- Процессы коммуницируют друг-с-другом напрямую (Netty) и через torrent-протокол
- Driver - особый процесс занятый оркестровкой



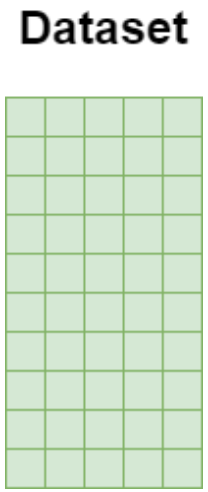
Из чего состоит Spark приложение ?

- Приложение Spark – набор процессов на узлах кластера.
- Процессы экзекьютеры обладают ресурсами (CPU, RAM, etc).
- Процессы коммуницируют друг-с-другом напрямую (Netty) и через torrent-протокол
- Driver - особый процесс занятый оркестровкой
- Экзекьютеры выполняют задачи, которые им дает Driver, хранят данные и пишут/читают внешнее хранилище.



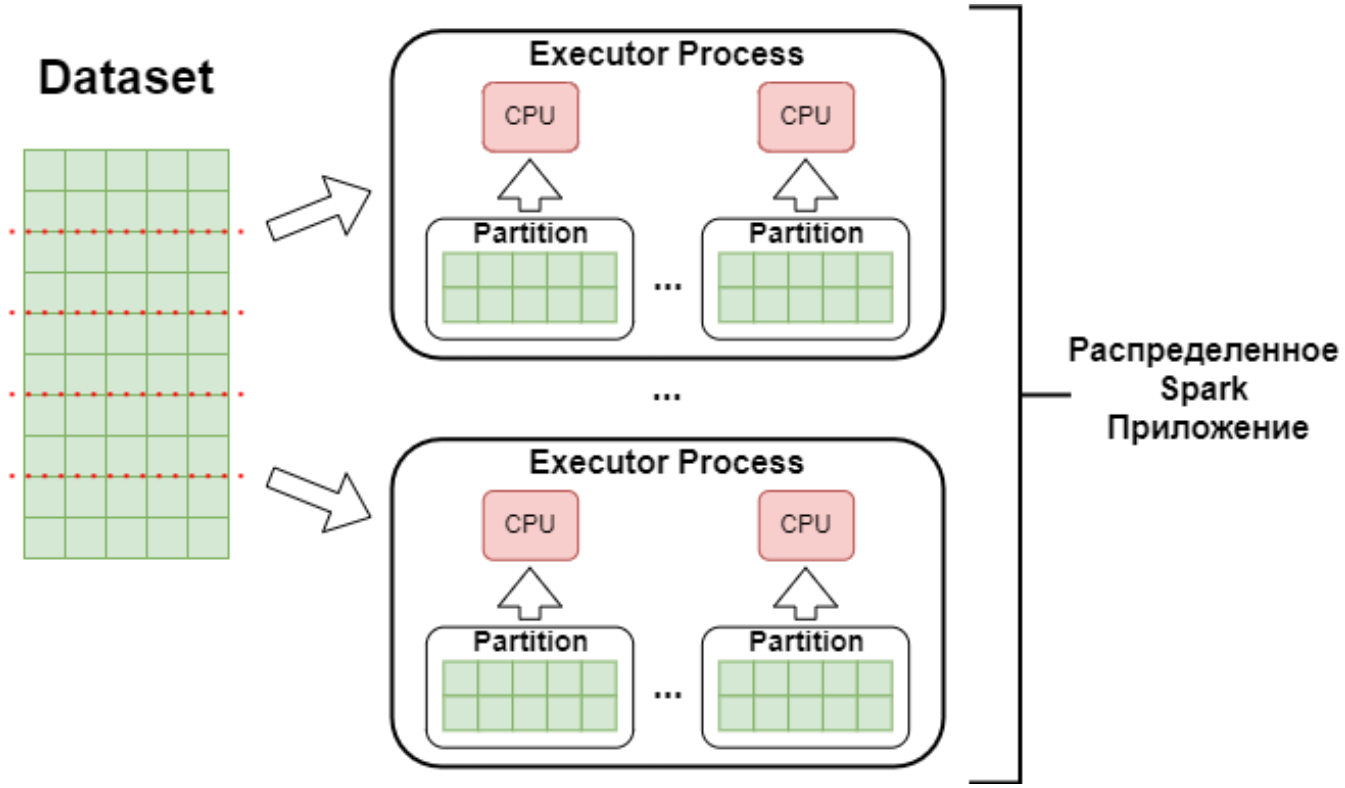
DataParallel модель в Spark

- Датасет разрезается *построчно* на отдельные части - *партиции*



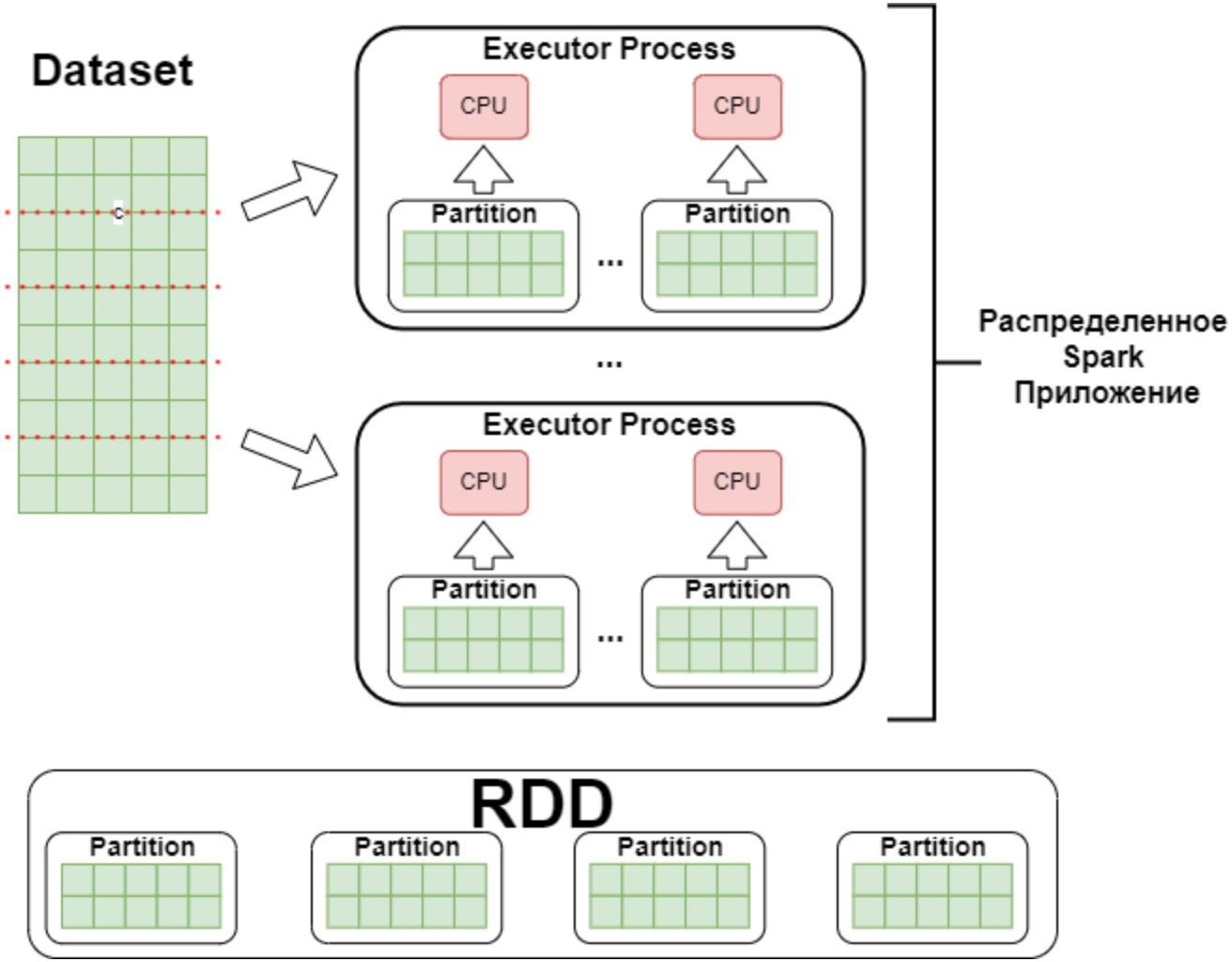
DataParallel модель в Spark

- Датасет разрезается *построчно* на отдельные части - *партиции*
- Партиции распределяются между экзекьютерами Spark-приложения.



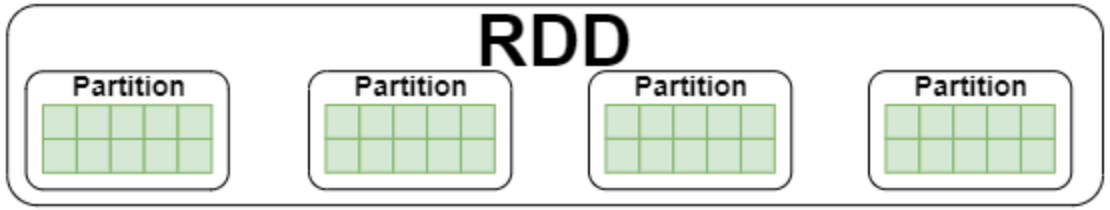
DataParallel модель в Spark

- Датасет разрезается *построчно* на отдельные части - *партиции*
- Партиции распределяются между экзекьютерами Spark-приложения.
- Набор распределенных партиций образует RDD (Resilient Distributed Dataset), являющейся базовой абстракцией датасета с которой работает Spark.



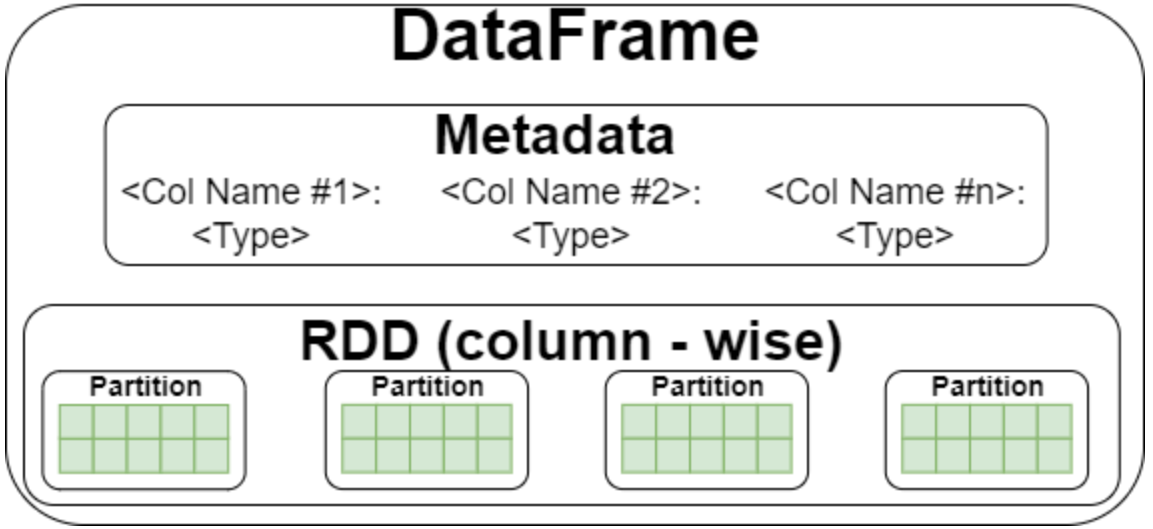
RDD: свойства

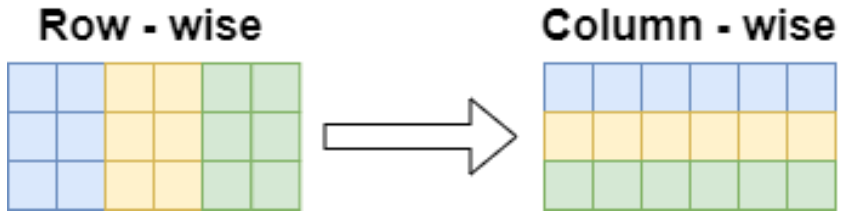
- Распределенный датасет, состоящий из партиций, каждой из которой можно задать предпочтительную локацию (preferred location).
- С RDD ассоциирована функция, которая строит его из родительского RDD (на который тоже есть ссылка).
- Для каждого RDD можно пройти весь путь до внешнего источника. Такой путь называется *Lineage*.
- RDD – надежный, т.к. утраченные партиции можно восстановить путем пересчета из источника до текущего RDD.



От RDD к DataFrame

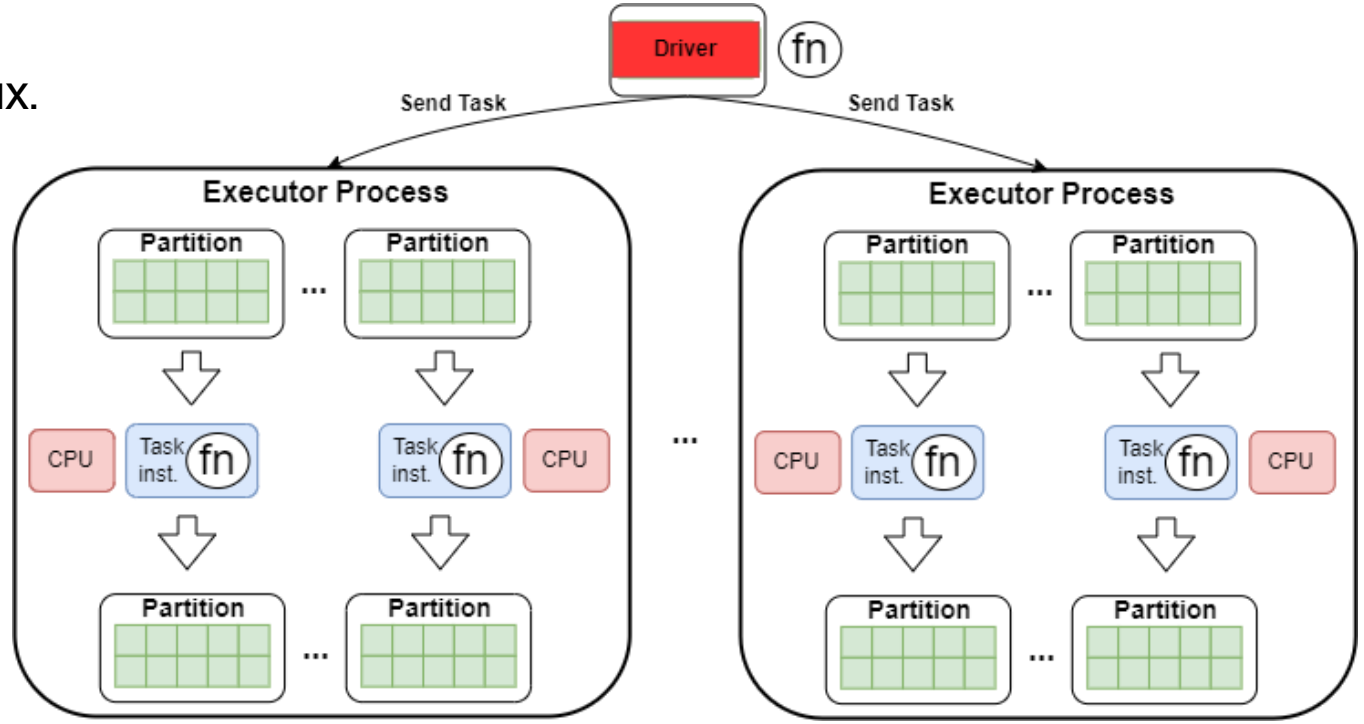
- DataFrame – особый тип представления данных в Spark.
- DataFrame построен на основе RDD, к которому *добавили метаданные* и подключили *колоночное хранение*.
- Метаданные позволяют Spark знать, что он хранит (название колонок, их тип) и как *каждая* колонка была получена и в каком порядке в них хранятся данные. Это позволяет *реализовать оптимизации* подобные тем, что есть в SQL базах данных
- Данные в таких RDD хранятся особым образом – в виде массивов байт в поколоночном представлении.





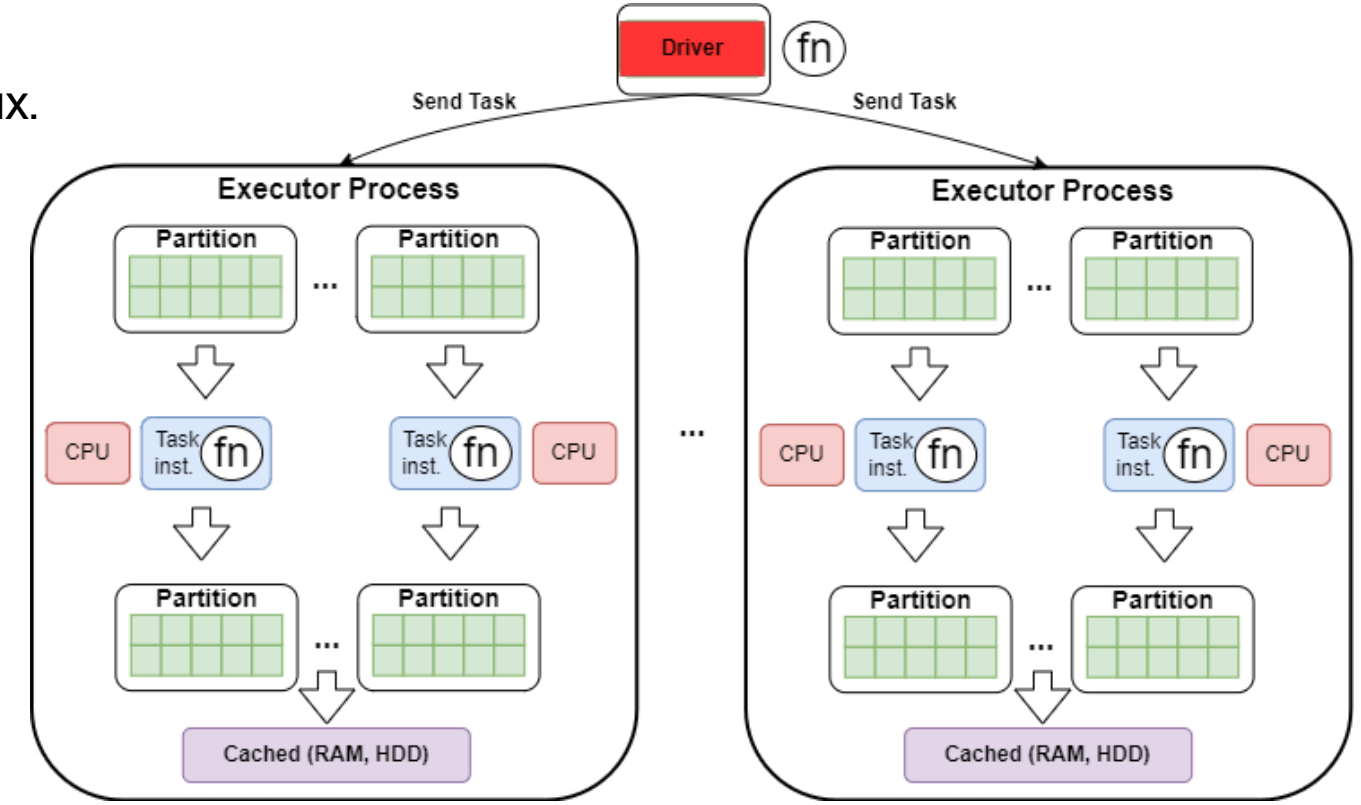
Как Spark выполняет вычисления?

- Driver выполняет скрипт пользователя и формирует задачи (Task) для обработки данных.
- Driver рассылает задачи на экзекьютеры, где каждая задачи производит обработку одной конкретной партиции *запись за записью*.
- В результате выполнения задачи получается новая партиция.



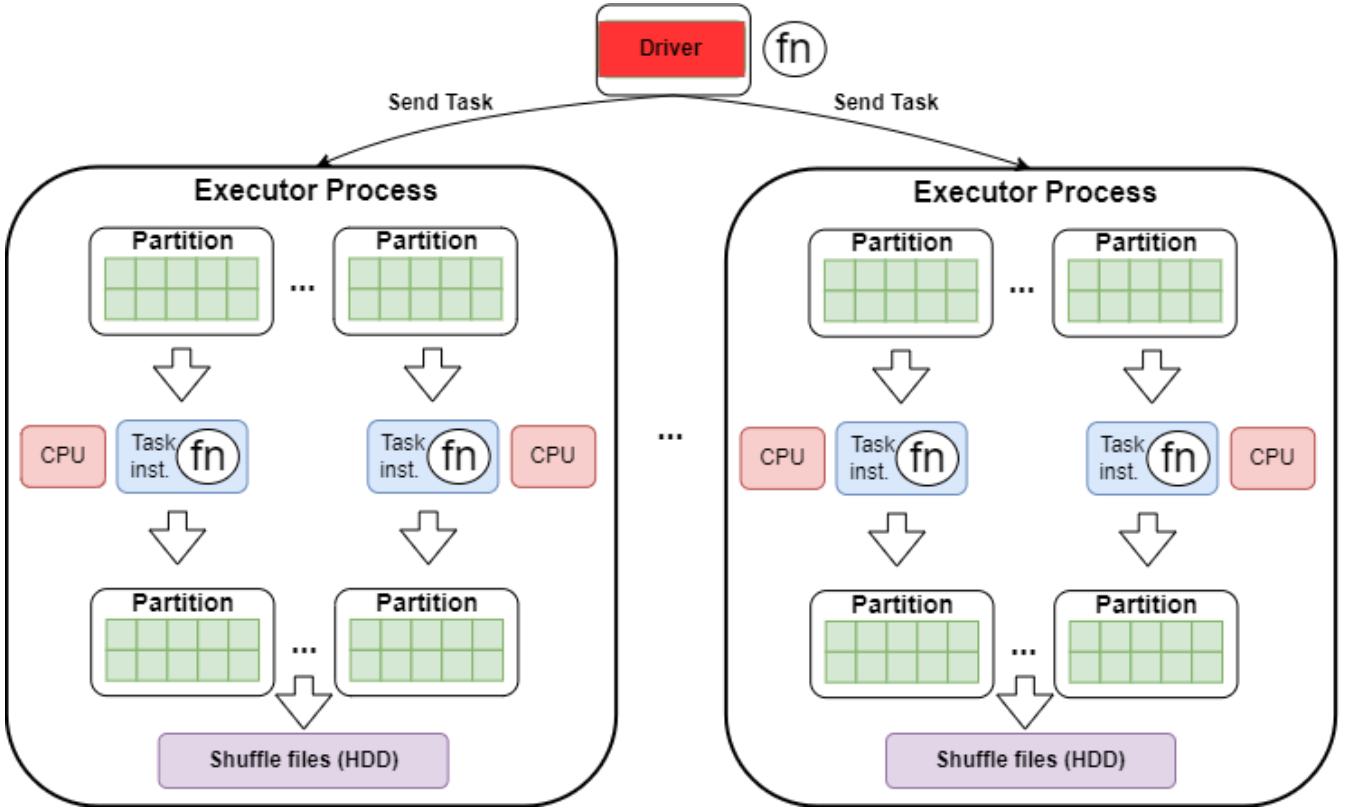
Как Spark выполняет вычисления?

- Driver выполняет скрипт пользователя и формирует задачи (Task) для обработки данных.
- Driver рассылает задачи на экзекьютеры, где каждая задачи производит обработку одной конкретной партиции *запись за записью*.
- В результате выполнения задачи получается новая партиция.
- Новая партиция может быть записана в кэш…



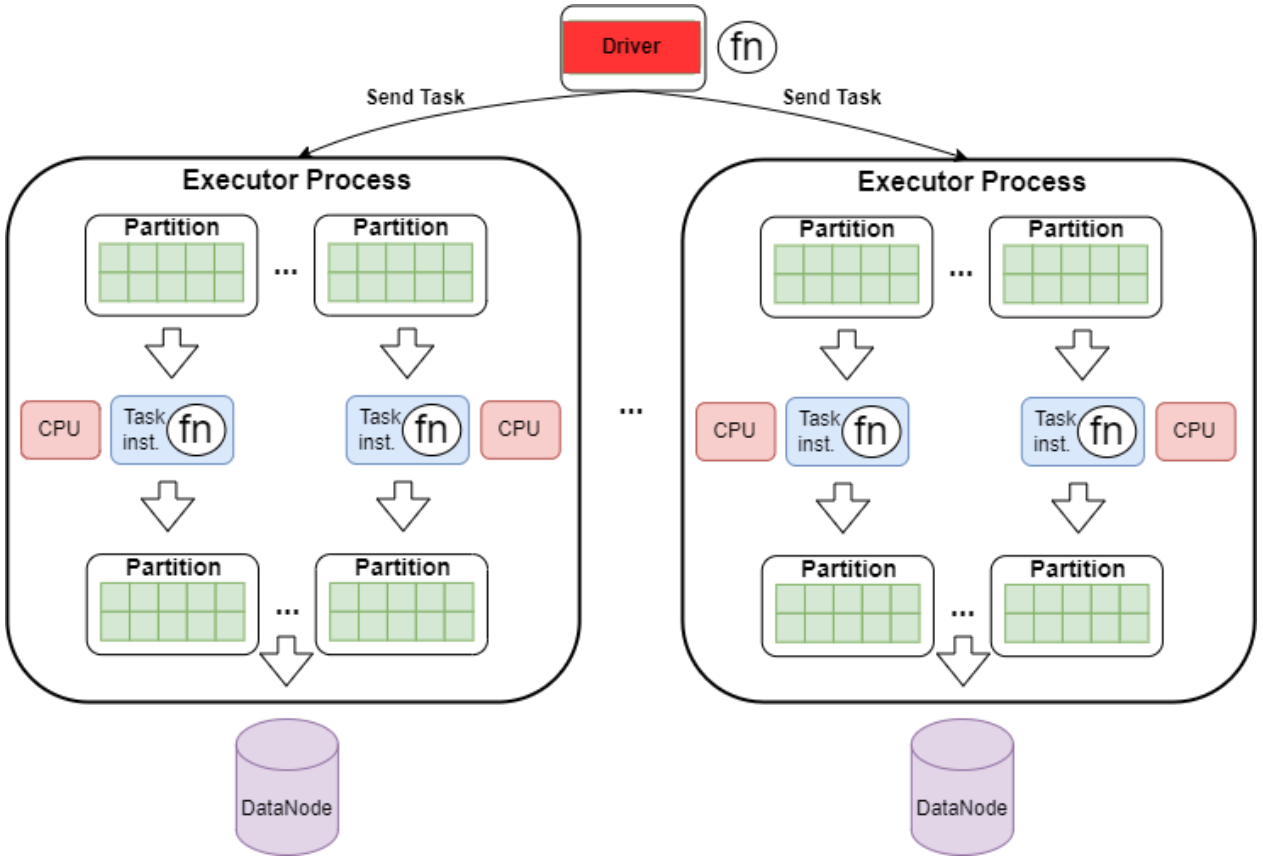
Как Spark выполняет вычисления?

- Driver выполняет скрипт пользователя и формирует задачи (Task) для обработки данных.
- Driver рассылает задачи на экзекьютеры, где каждая задачи производит обработку одной конкретной партиции *запись за записью*.
- В результате выполнения задачи получается новая партиция.
- Новая партиция может быть записана в кэш…
- …Или же стать shuffle file с помощью которых производиться обмен с другими экзекьютерами.



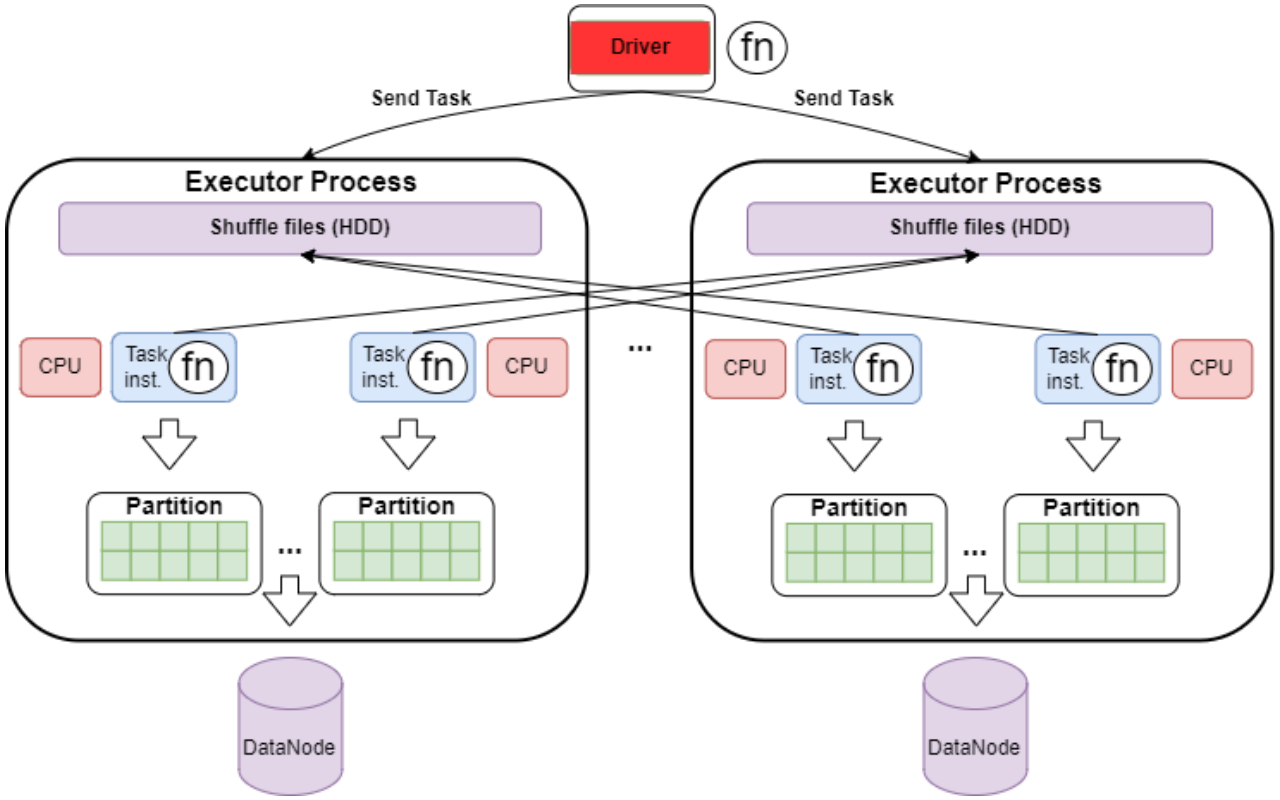
Как Spark выполняет вычисления?

- Driver выполняет скрипт пользователя и формирует задачи (Task) для обработки данных.
- Driver рассылает задачи на экзекьютеры, где каждая задачи производит обработку одной конкретной партиции *запись за записью*.
- В результате выполнения задачи получается новая партиция.
- Новая партиция может быть записана в кэш…
- …Или же стать shuffle file с помощью которых производиться обмен с другими экзекьютерами.
- … Или же быть записана во внешнее хранилище.



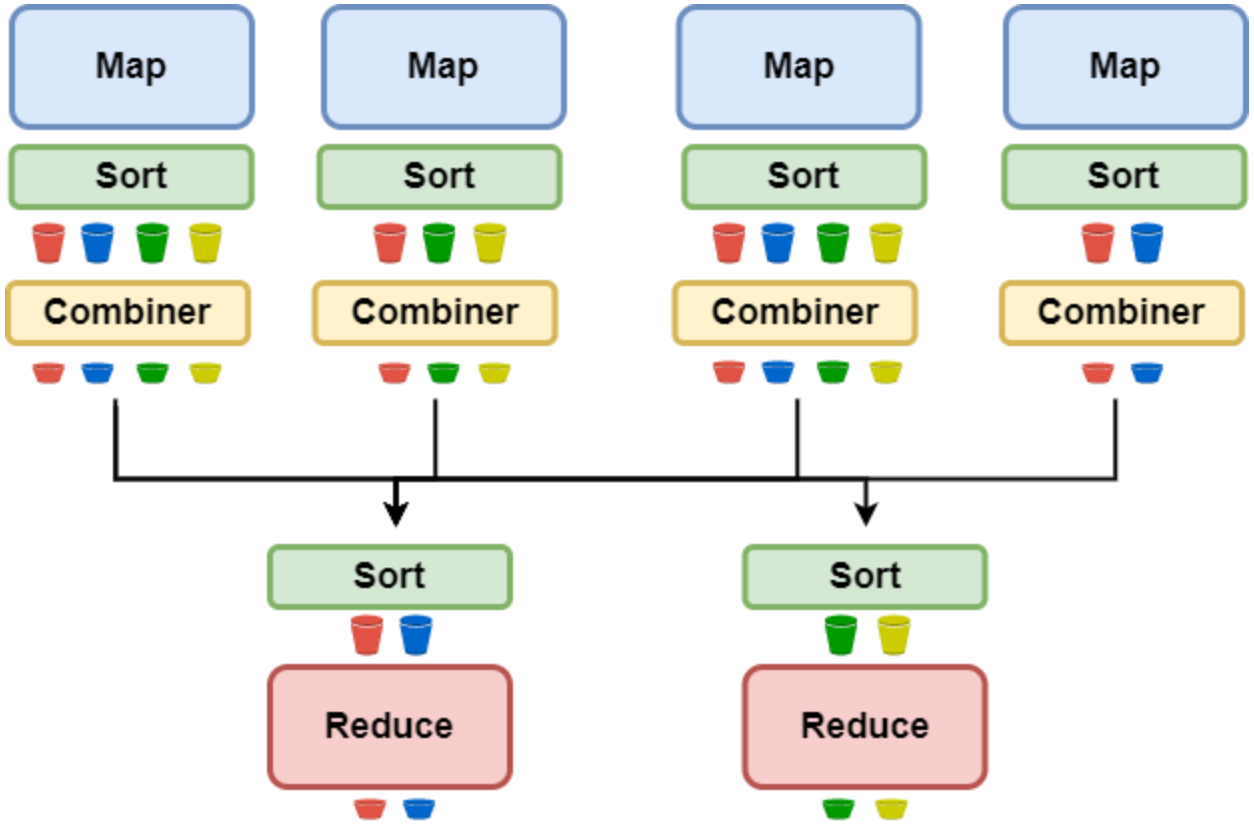
Как Spark выполняет вычисления?

- Задачи в Spark могут не только читать уже существующую партицию на экзекьютере, где они запускаются.
- Задача может читать кусок данных из внешнего хранилища, т.е. создавать начальную партицию на экзекьютере.
- Задача может читать shuffle files с *<u>другого экзекьютера</u>*.



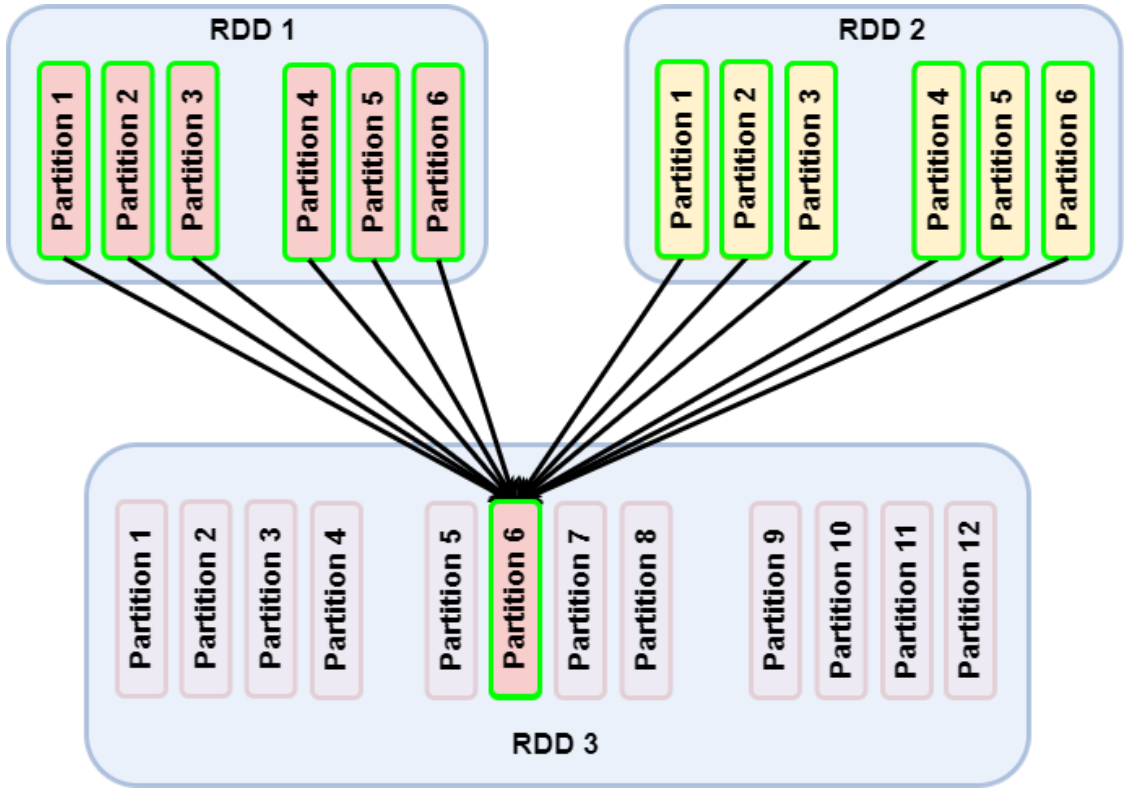
MapReduce паттерн

- Ключевой паттерн обработки данных на котором строится как обработка SQL подобных запросов, так и многих алгоритмов машинного обучения.
- Состоит из двух основных фаз Map и Reduce (но есть еще дополнительные фазы) и обмена данными между ними Shuffle.
- Map – производит записи в виде <key>:<value>
- Shuffle – собирает все записи с одним и тем же значением <key> на одном экзекьютере.
- Reduce – обрабатывает все записи с одним и тем же значением <key> в одном батче (за один вызов функции).

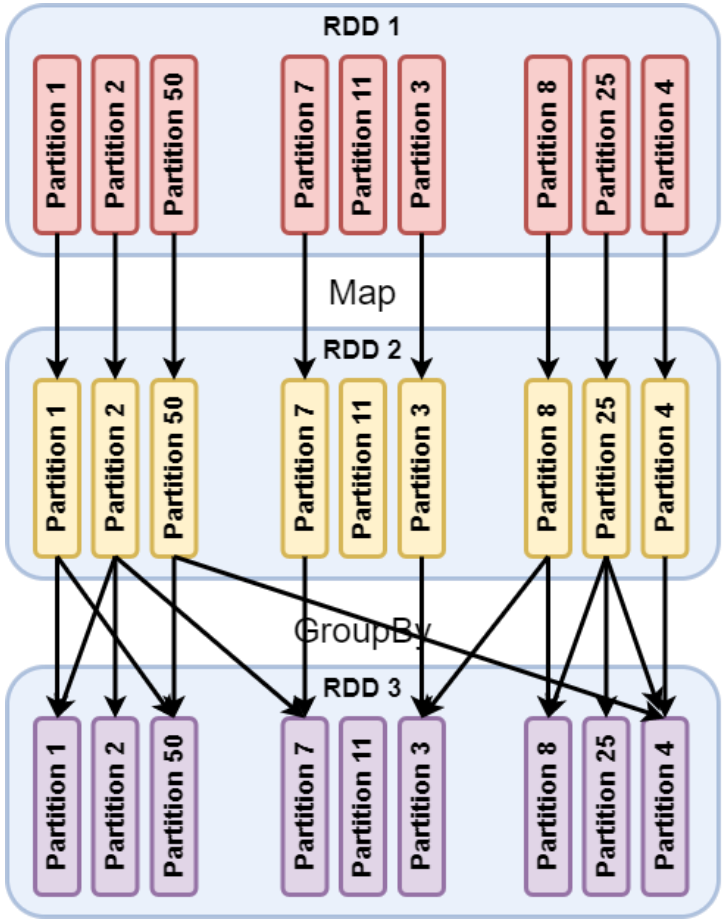


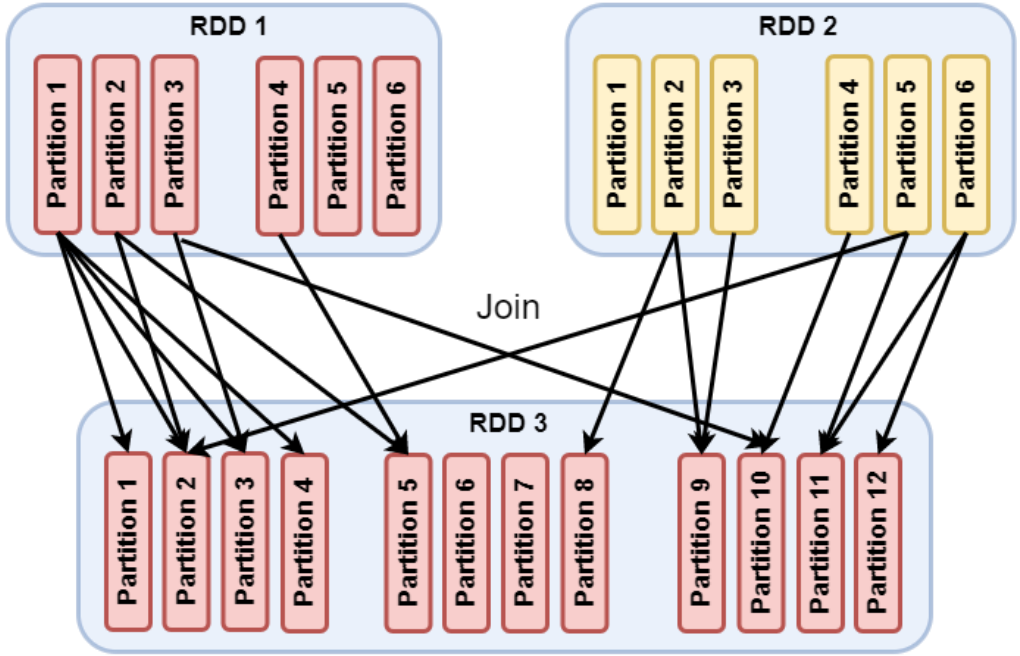
Shuffle: обмен данными

- В Spark во время Shuffle производит новый RDD.
- Каждая партиция этого нового RDD будет содержать данные с одним и тем же ключом из всех партиций родительского датафрейма.
- В одной партиции нового RDD могут собраться несколько ключей, если ключей больше, чем партиций (а обычно оно так и есть).
- Shuffle-задача должна собрать shuffle files с нужными частями *для всех родительских партиций*. Что может быть довольно дорого.



Shuffle: операции

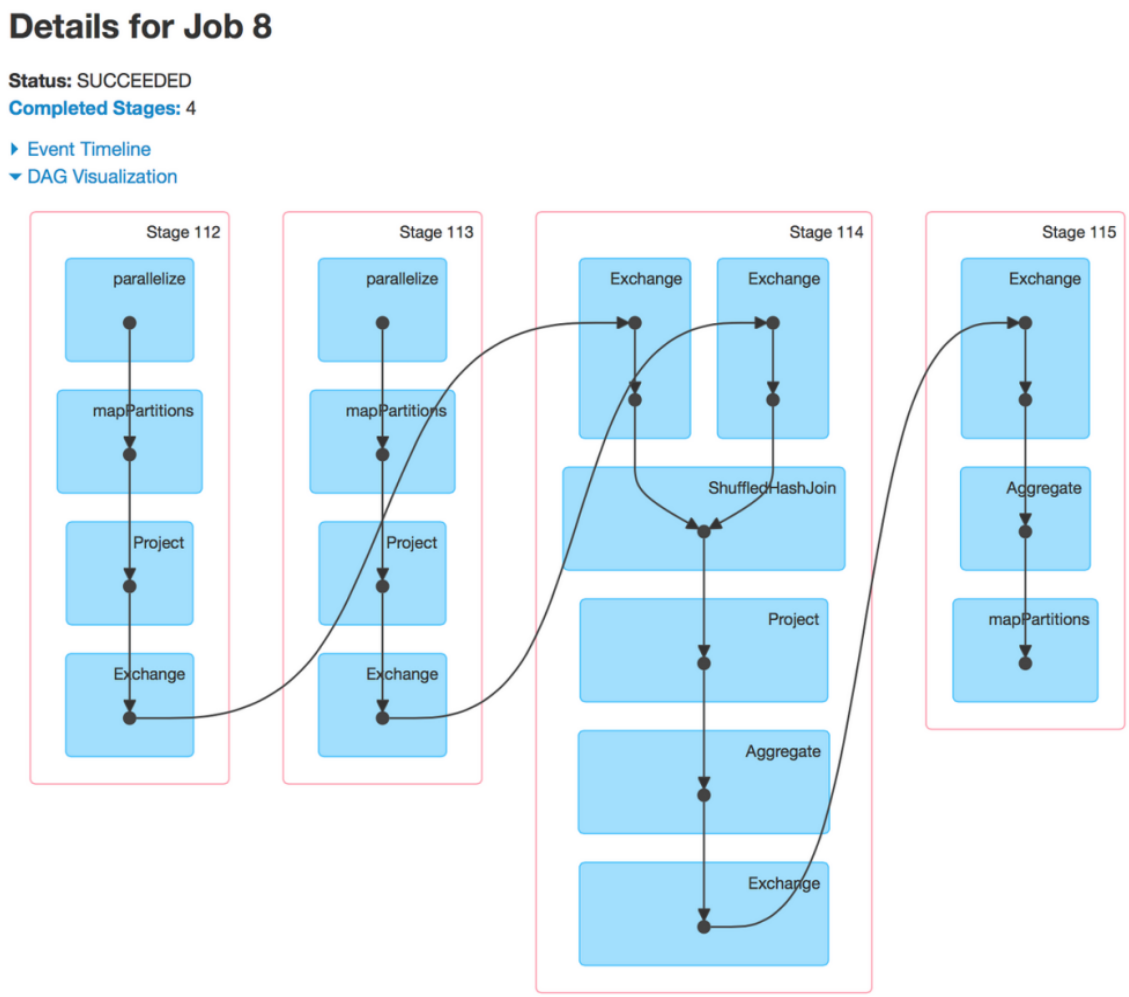




Shuffle – основа таких фундаментальных операций как Group By и Join.

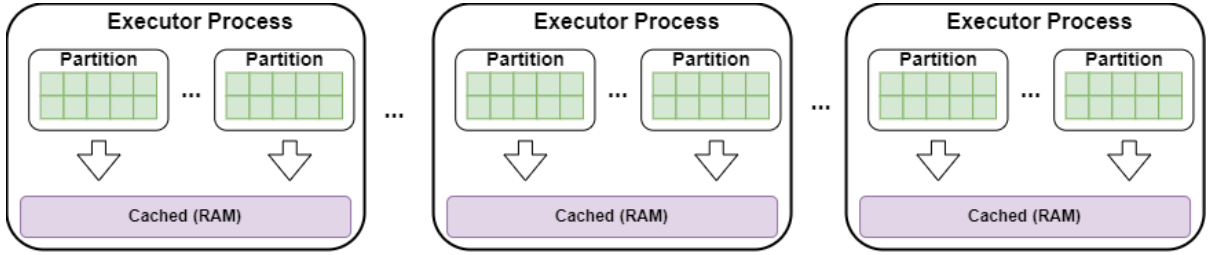
Beyond MapReduce: DAG как основа Spark

- Чистый MapReduce предполагает наличие только графов одного типа (+ кейсы, где есть только Map или только Reduce).
- Модель вычислений Spark гибче – она предполагает выполнение любых направленных ациклических графов (Directed Acyclic Graph, DAG).
- Отдельный *логический процесс* обработки внутри Spark приложения называется Job/
- Ключевое преимущество такой модели – снижение накладных расходов на запись промежуточных результатов во внешнее хранилище.
- В сочетании с возможностью кеширования промежуточных результатов между разными Job, это позволяет Spark быть очень эффективным для итеративной обработки данных.



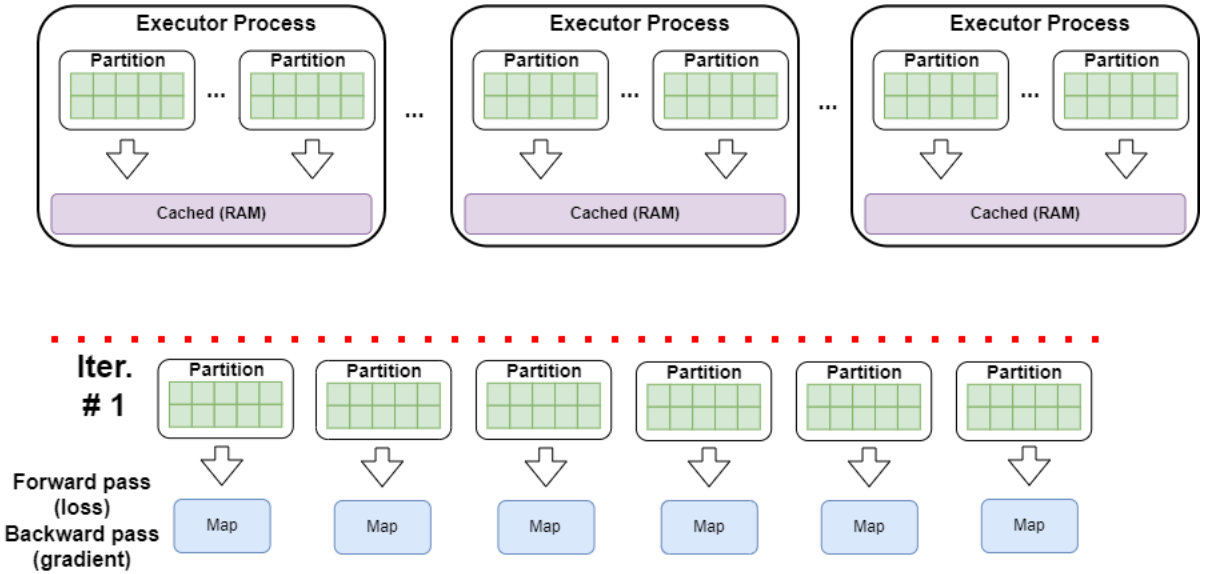
Итеративные вычисления в Spark

- Итеративные вычисления важная часть многих алгоритмов ML и в частности SGD (stochastic gradient descent), который используется для обучения.
- Для итеративных вычислений датасет обычно должен быть закеширован.



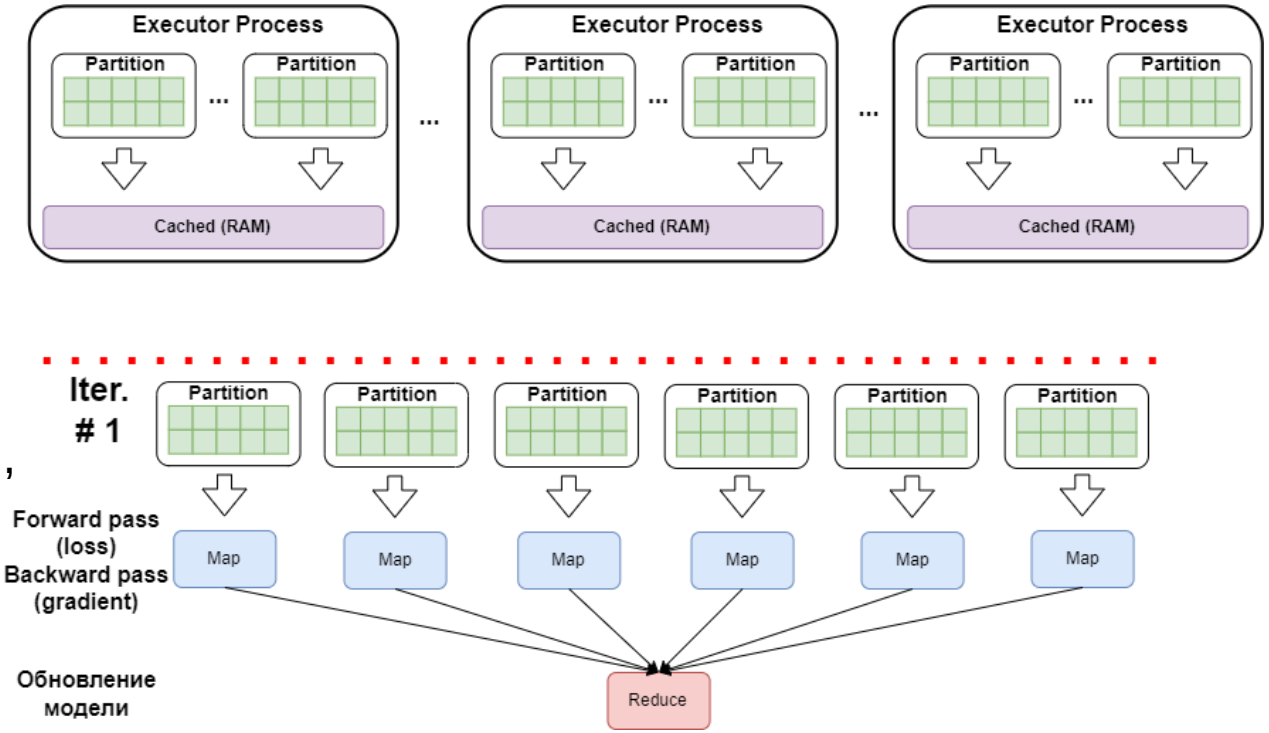
Итеративные вычисления в Spark

- Итеративные вычисления важная часть многих алгоритмов ML и в частности SGD (stochastic gradient descent), который используется для обучения.
- Для итеративных вычислений датасет обычно должен быть закеширован.
- На каждой итерации выполняется MapReduce Job, где Map вычисляет ошибку и градиент…



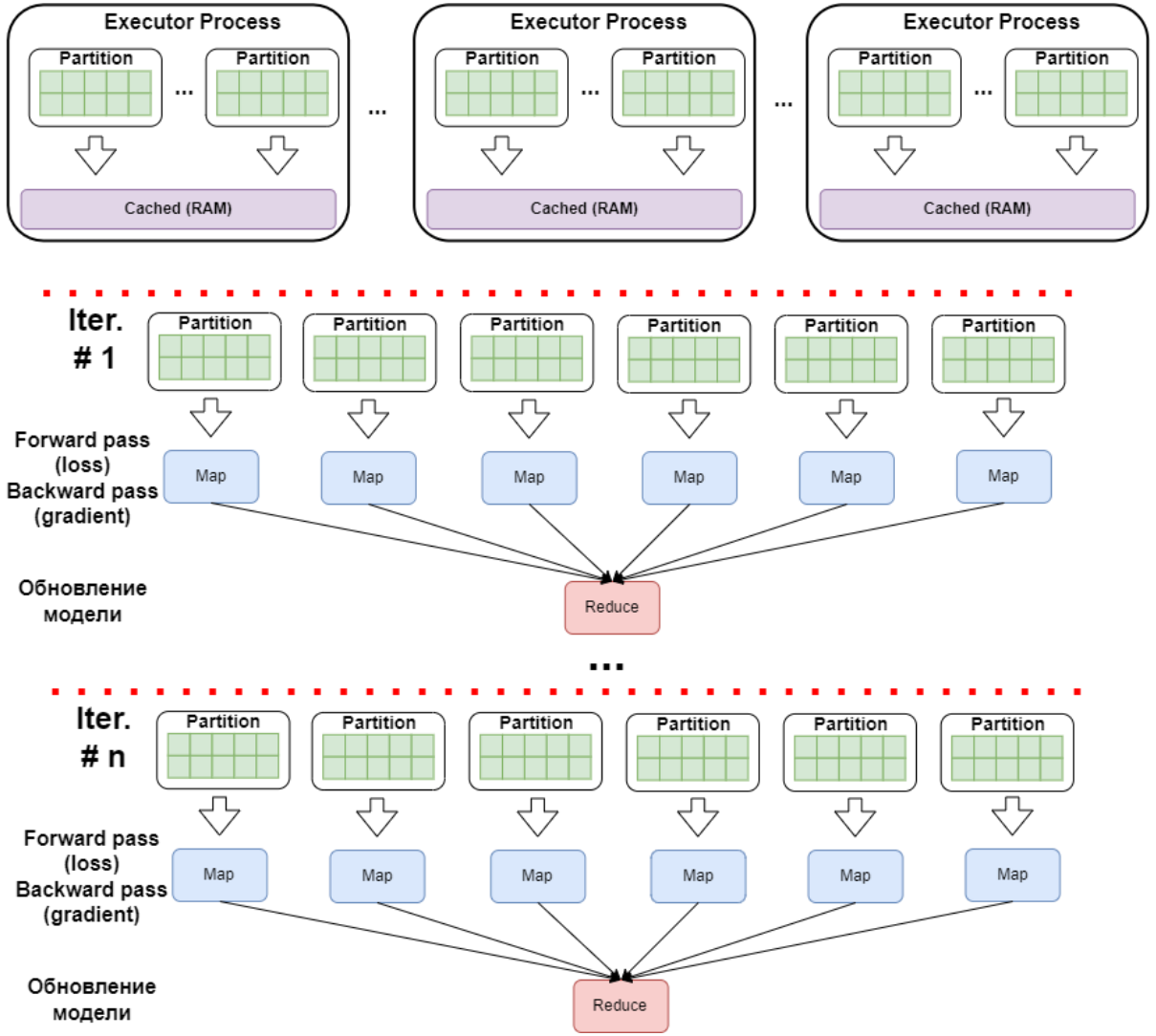
Итеративные вычисления в Spark

- Итеративные вычисления важная часть многих алгоритмов ML и в частности SGD (stochastic gradient descent), который используется для обучения.
- Для итеративных вычислений датасет обычно должен быть закеширован.
- На каждой итерации выполняется MapReduce Job, где Map вычисляет ошибку и градиент…
- … A reduce собирает градиенты и обновляет параметры модели.



Итеративные вычисления в Spark

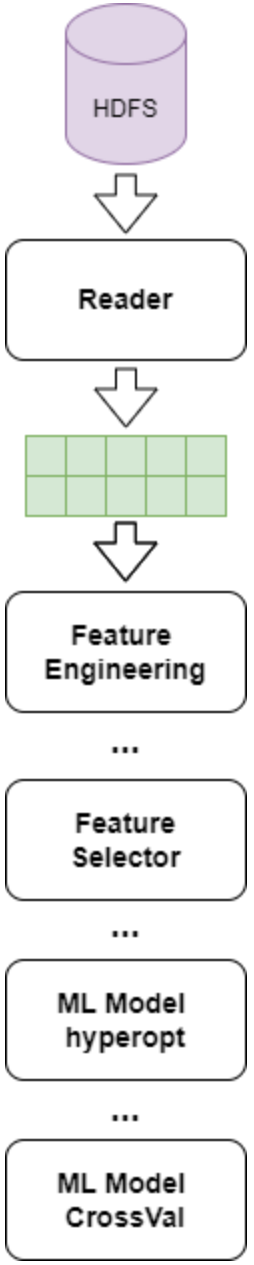
- Итеративные вычисления важная часть многих алгоритмов ML и в частности SGD (stochastic gradient descent), который используется для обучения.
- Для итеративных вычислений датасет обычно должен быть закеширован.
- На каждой итерации выполняется MapReduce Job, где Map вычисляет ошибку и градиент…
- … A reduce собирает градиенты и обновляет параметры модели.
- Такая обработка выполняется в течении N повторений – итераций.



Пайплайны AutoML

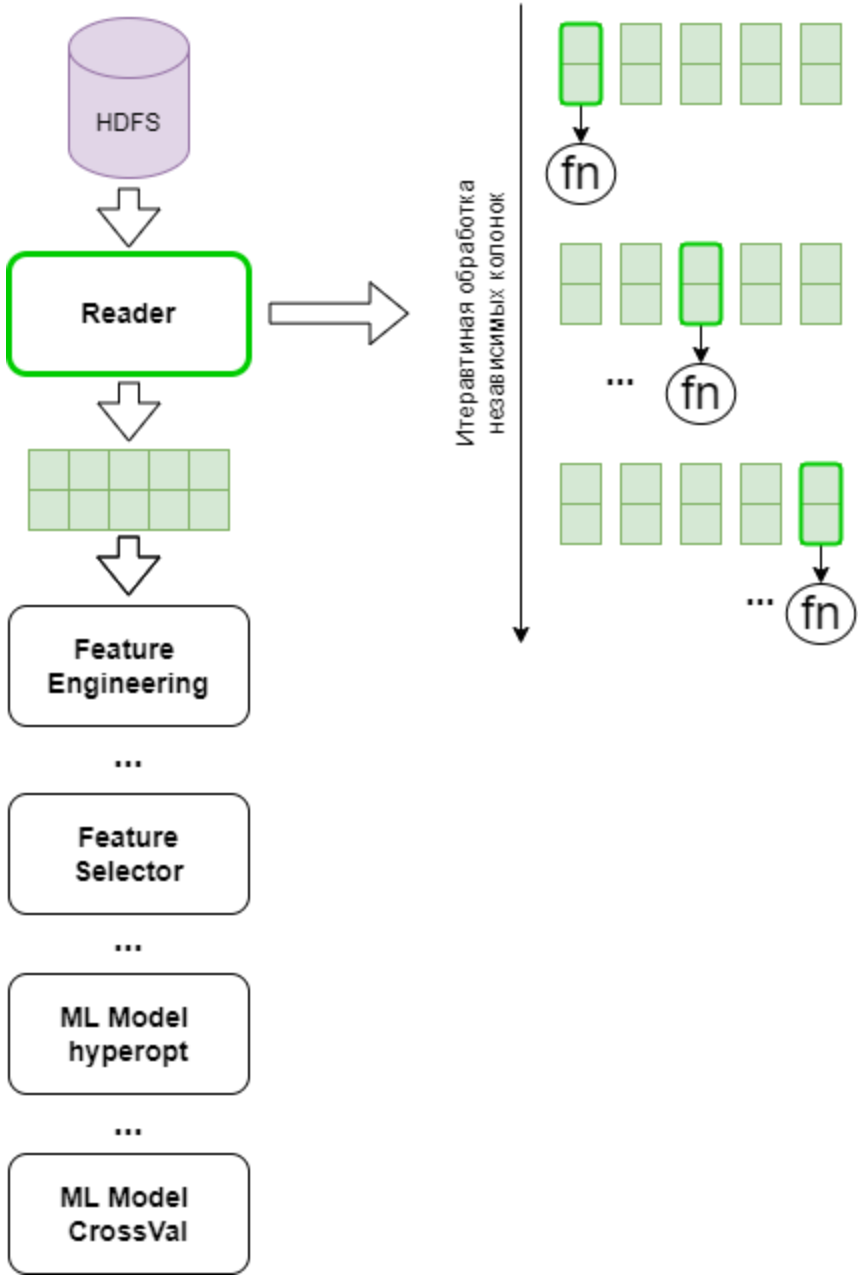
Простой пайплайн состоит из этапов:

- идентификации типов колонок – numeric, categorical (Reader)
- создания фич (Feature Engineering) – колонок только numeric типов, очищенных и пригодных для обучения модели
- отсеивания “бесполезных” фич (Feature Selector)
- поиска гиперпараметров для обучения (hyperopt)
- обучения модели (crossval).



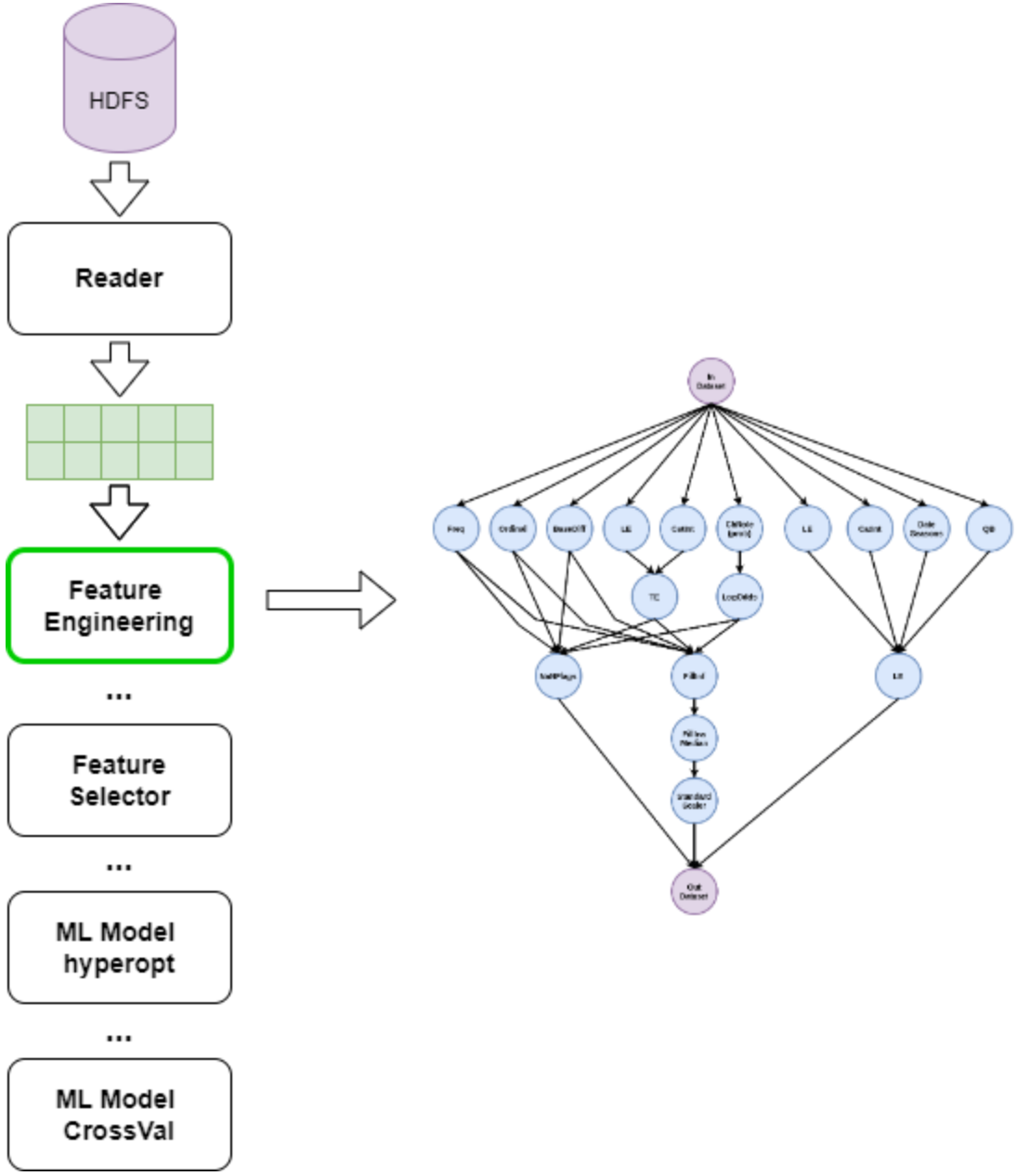
Пайплайны AutoML

- Reader обрабатывает итеративно колонку за колонкой. *Датасет можно не кешировать, если источник поддерживает поколоночное чтение.*



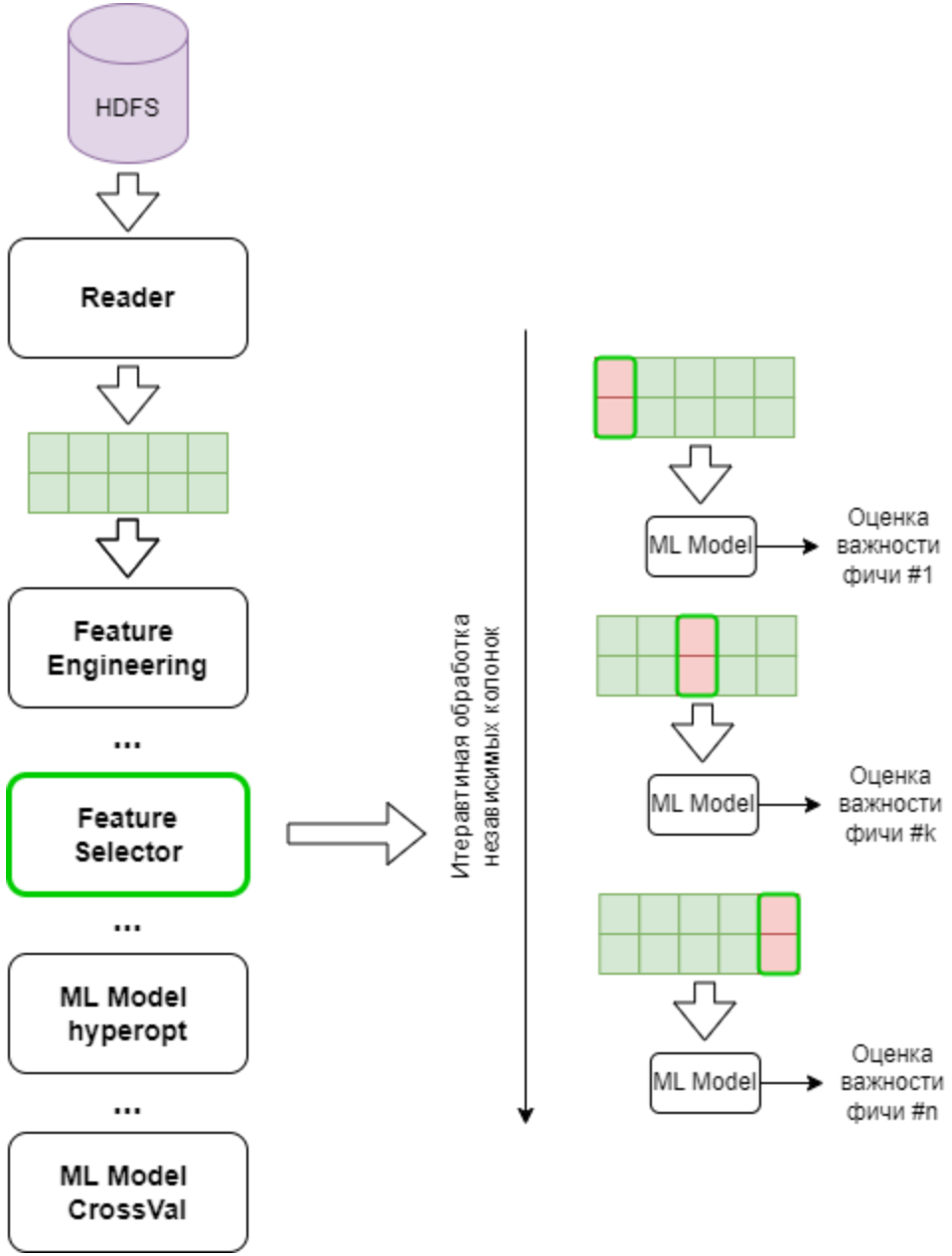
Пайплайны AutoML

- Reader обрабатывает итеративно колонку за колонкой. *Датасет можно не кешировать, если источник поддерживает поколоночное чтение.*
- В Feature Engineering мы последовательно преобразуем датасет через набор трансформаций. *По каждому промежуточному датасету нам нужно сделать два прохода.*



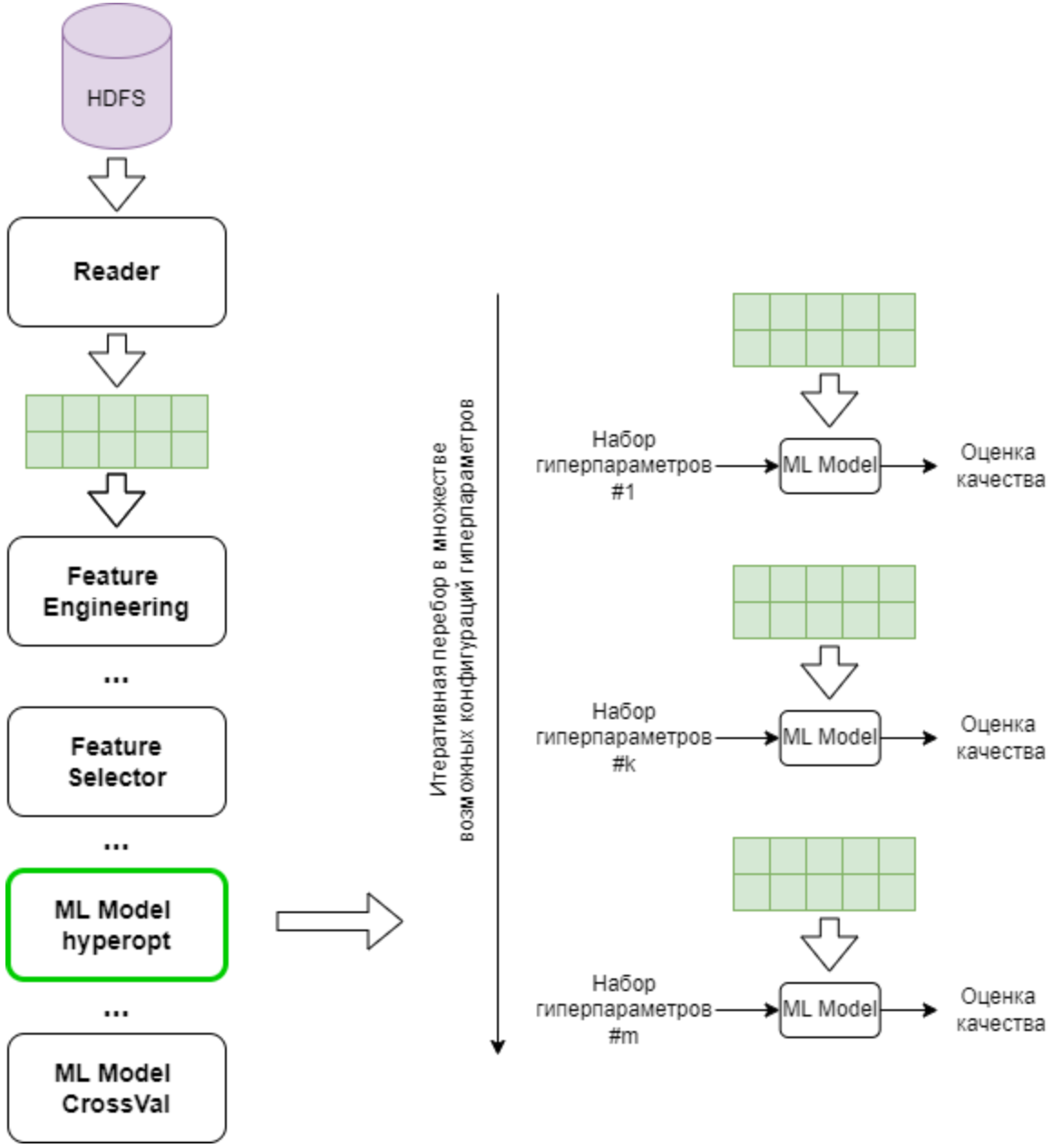
Пайплайны AutoML

- Reader обрабатывает итеративно колонку за колонкой. *Датасет можно не кешировать, если источник поддерживает поколоночное чтение.*
- В Feature Engineering мы последовательно преобразуем датасет через набор трансформаций. *По каждому промежуточному датасету нам нужно сделать два прохода.*
- В Feature Selector происходит итеративная оценка качества при манипуляции с каждой колонкой. *Датасет нужно кешировать.*



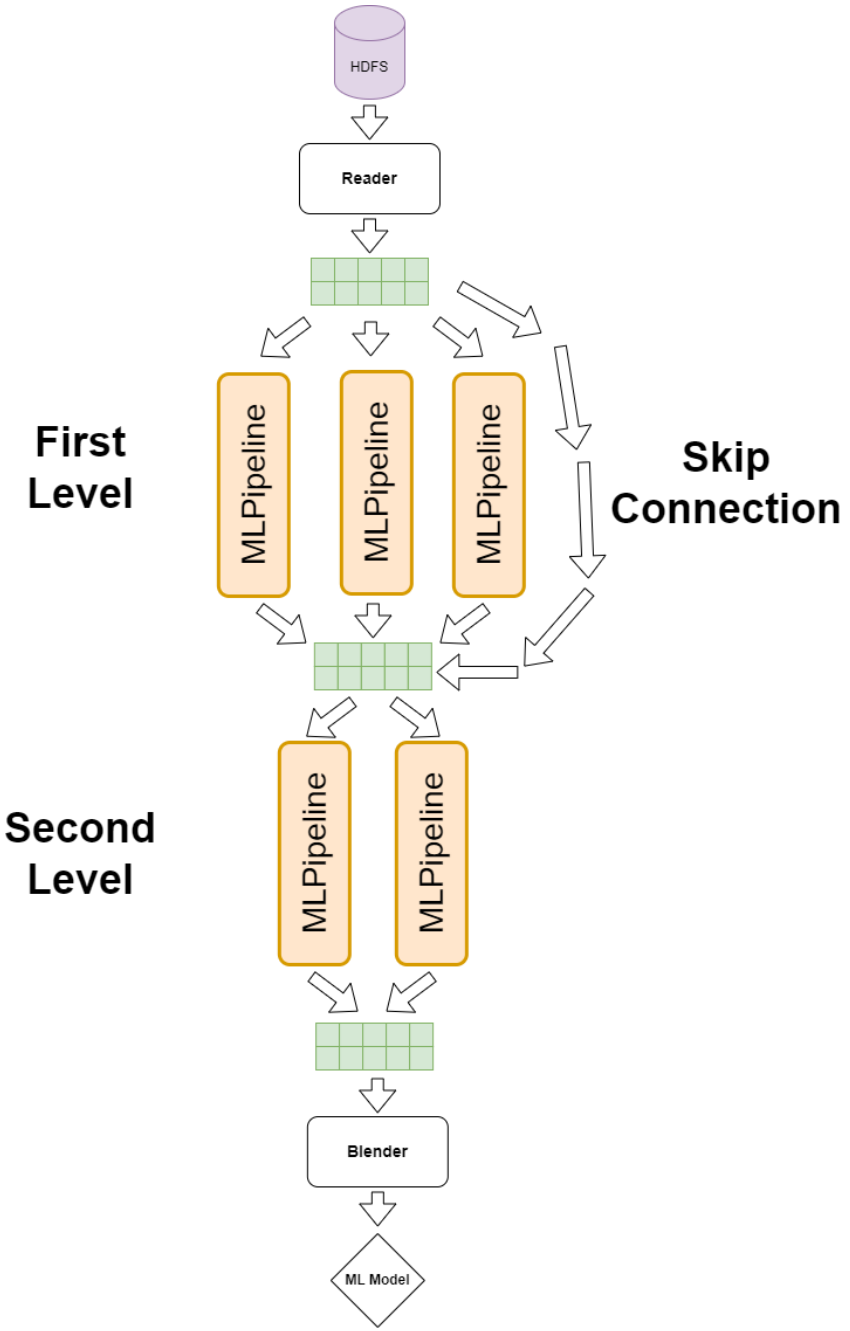
Пайплайны AutoML

- Reader обрабатывает итеративно колонку за колонкой. *Датасет можно не кешировать, если источник поддерживает поколоночное чтение.*
- В Feature Engineering мы последовательно преобразуем датасет через набор трансформаций. *По каждому промежуточному датасету нам нужно сделать два прохода.*
- В Feature Selector происходит итеративная оценка качества при манипуляции с каждой колонкой. *Датасет нужно кешировать.*
- В hyperopt обучаем множество моделей с разными конфигурациями. *Датасет нужно кешировать.*



Полный пайплайн AutoML

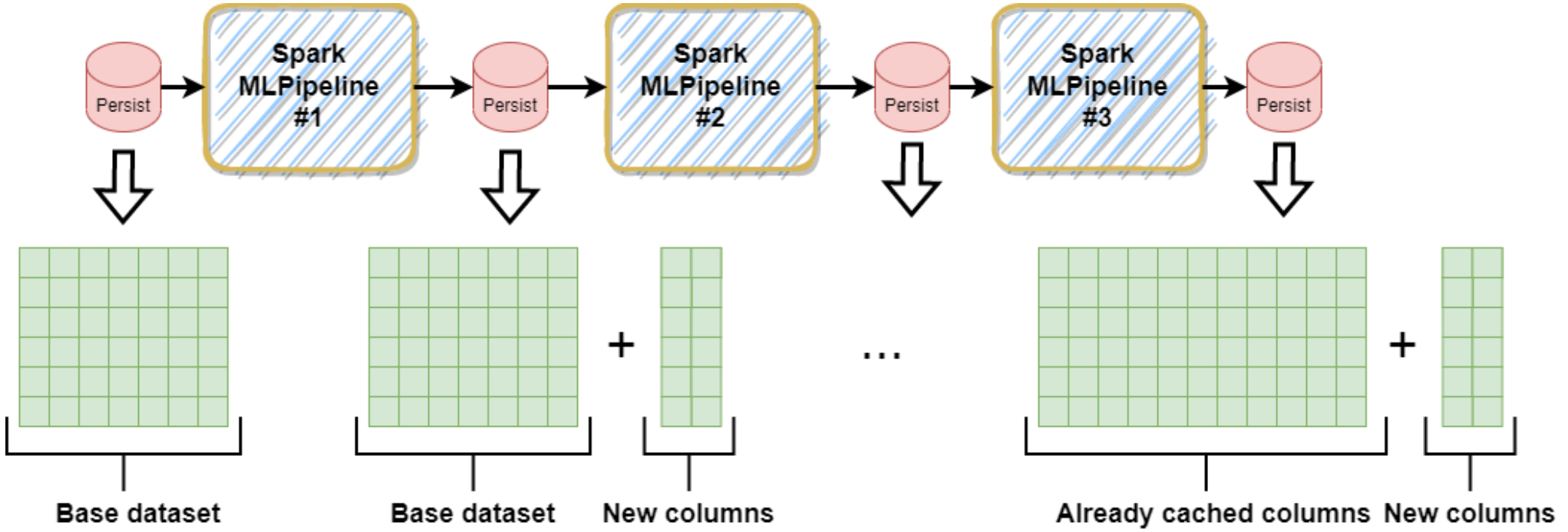
- Полный пайплайн может содержать несколько уровней
- В каждом уровне может быть несколько ML моделей и каждая может быть со своей подготовкой фич.
- При “переходе” на следующей уровень требуется объединение результатов (предсказаний) от всех моделей в слое.
- … А может еще потребоваться совмещение их с начальным датасетом, если включен флаг **Skip Connection.**



Особенности переноса решения на Spark.

Объединение (Join) пайплайнов в слое

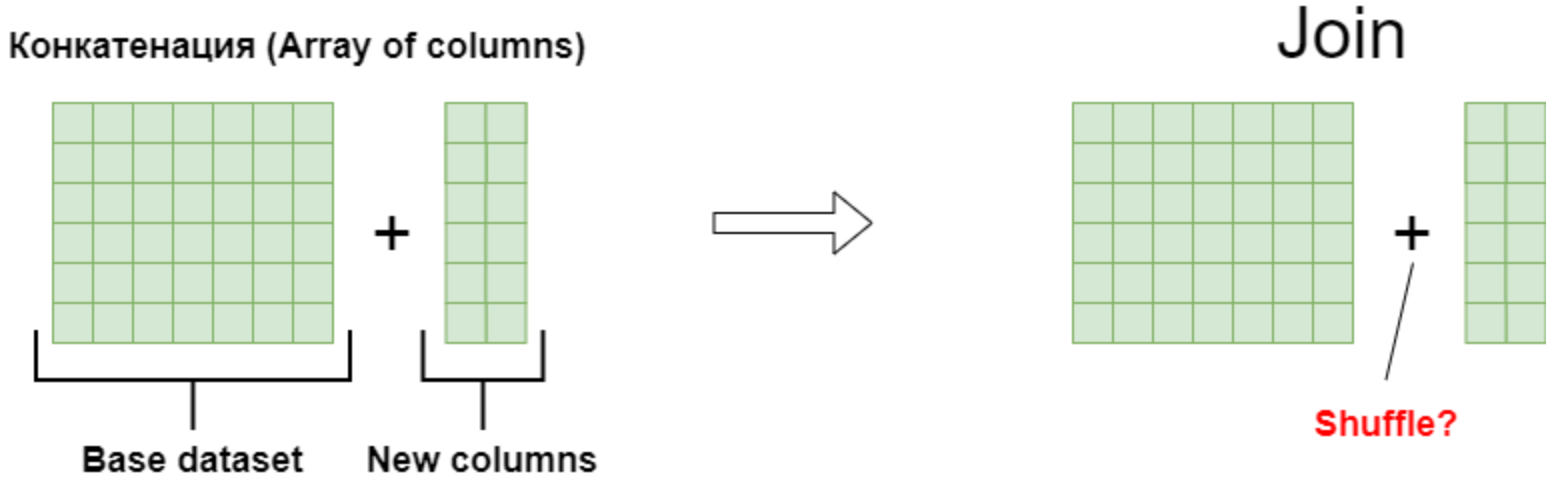
Зачем нам вообще объединять пайплайны через Join, если мы выполняем их все последовательно?



Возможны лишние многократные кэширования начального “широкого” датафрейма, т.к. они нужны каждому следующему пайплайну, и вновь добавляемых колонок.

Join все равно потребуется, если реализовывать параллельную обработку нескольких MLPipeline.

Операция объединения фреймов (row-wise)



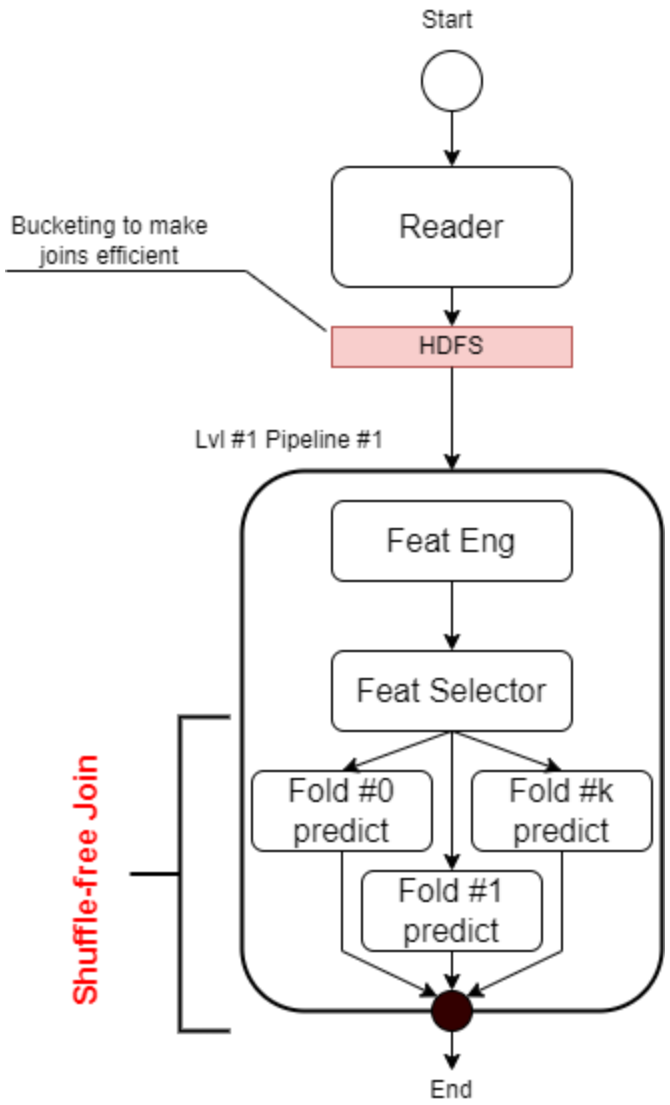
Простая конкатенация массивом в LAMA

- Нет скидывания частей фрейма на диск
- Нет сетевых передач данных
- Нет сортировки

Join в Spark, который без должной оптимизации может окончиться shuffle-ом.

- Может быть запись частей фрейма на диск
- Могут быть передачи данных
- Могут быть сортировки

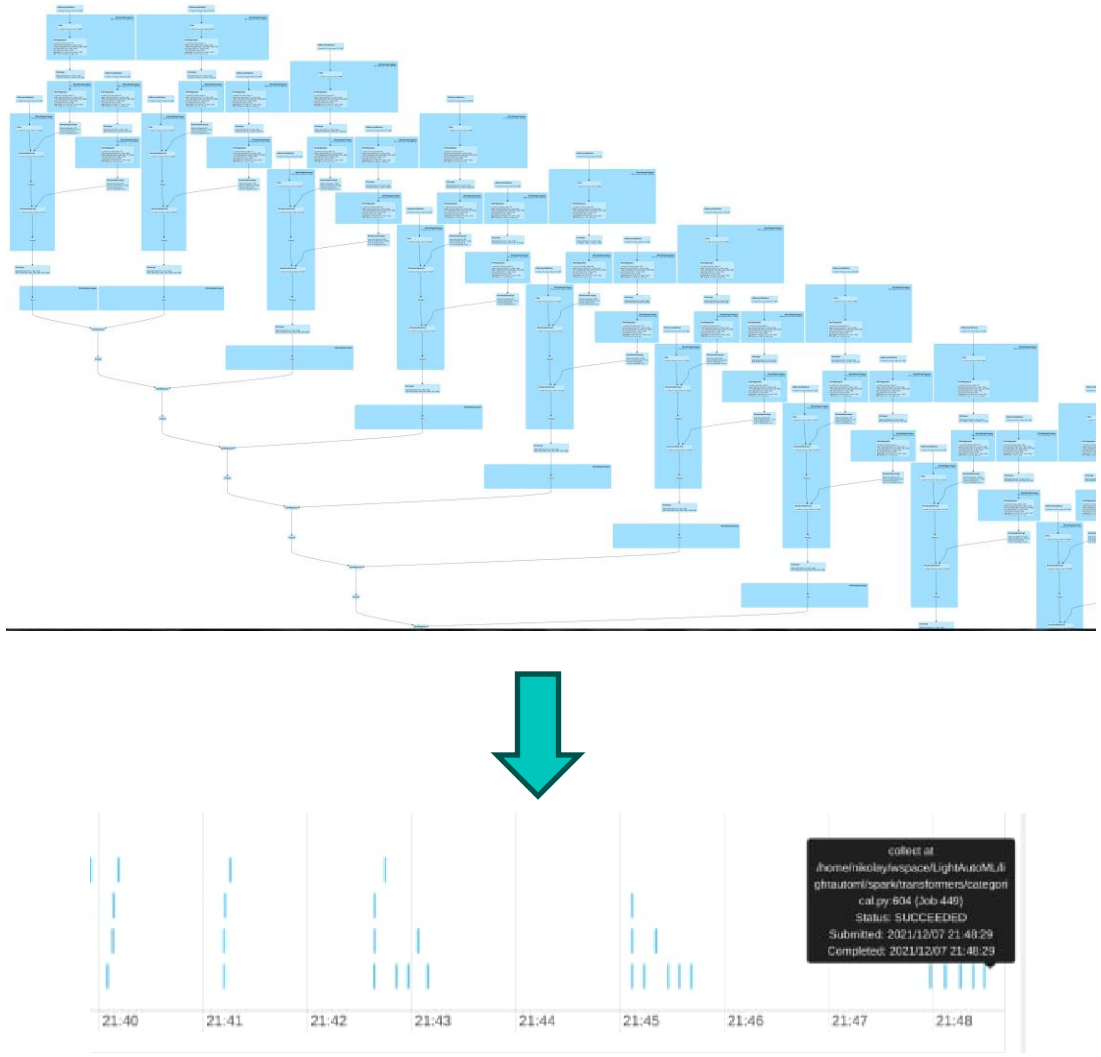
Операция объединения фреймов: бакетирование



- Бакетирование разделяет датасет на части по ключу так, что Spark знает где находятся в HDFS и в каком порядке сохранены ряды с конкретными значениями ключа.
- Если два датафрейма бакетированы по одному и тому же ключу, то записи с одинаковыми ключами гарантированно окажутся на одном и том же экзекьютере при вычислениях и кешировании.
- Все потомк бакетированного датафрейма, не изменяющие ключ и сортировку остаются бакетированными.
- Ко-локация позволяет оптимизировать Join, убрав shuffle и сортировку, что делает его близким к конкатенации колонок в обычной LAMA.
- Если датасет не бакетирован изначально, бакетирование требует создание полной копии датасета на HDFS, что может быть довольно долго.

Длинные пайплайны

С разрастанием истории (граф обработки), обработка может замедляться из-за оптимизатора



- План вычислений становится очень большим и планировщик (Catalyst) тратит все больше времени на планирование – получаем Long-lineage problem / bottleneck.
- Кроме того, замедление может наступать из-за необходимости каждый раз доставлять все вычислительные зависимости вместе с новой job – доп. расходы на трансфер тасок / десериализацию [1].
- Таким проблемам наиболее подвержены много этапные итеративные вычисления.
- Кэширование не решает, т.к.оптимизатор не учитывает закешированные промежуточные данныепри планировании

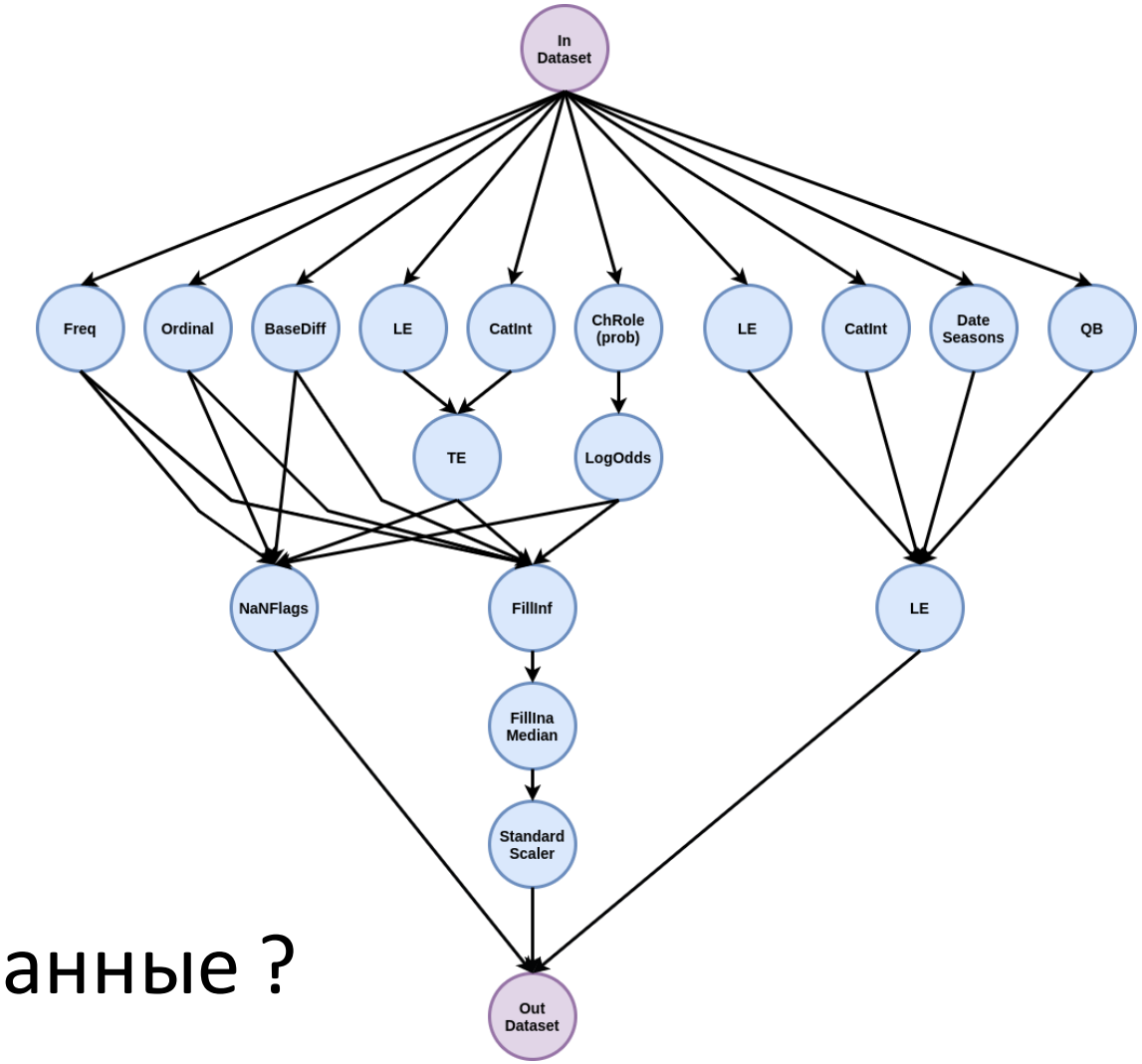
ИТМО

Длинные пайпы и join (даже бакетированные) увеличвают сложность планирования.

Оптимизация кеширования в пайплайнах обработки

Особенности

- FeaturePipeline подготавливает данные для использование в ML алгоритме.
- Подготовка состоит из применения множества энкодеров и трансформеров данных.
- Какие энкодеры будут применяться зависит от имеющихся в данных колонок и их типа.
- А у некоторых энкодеров есть строгий порядок следования, т.е. зависимости по данным. Например TargetEncoder требует *уже обработанные* LabelEncoder-ом колонки/

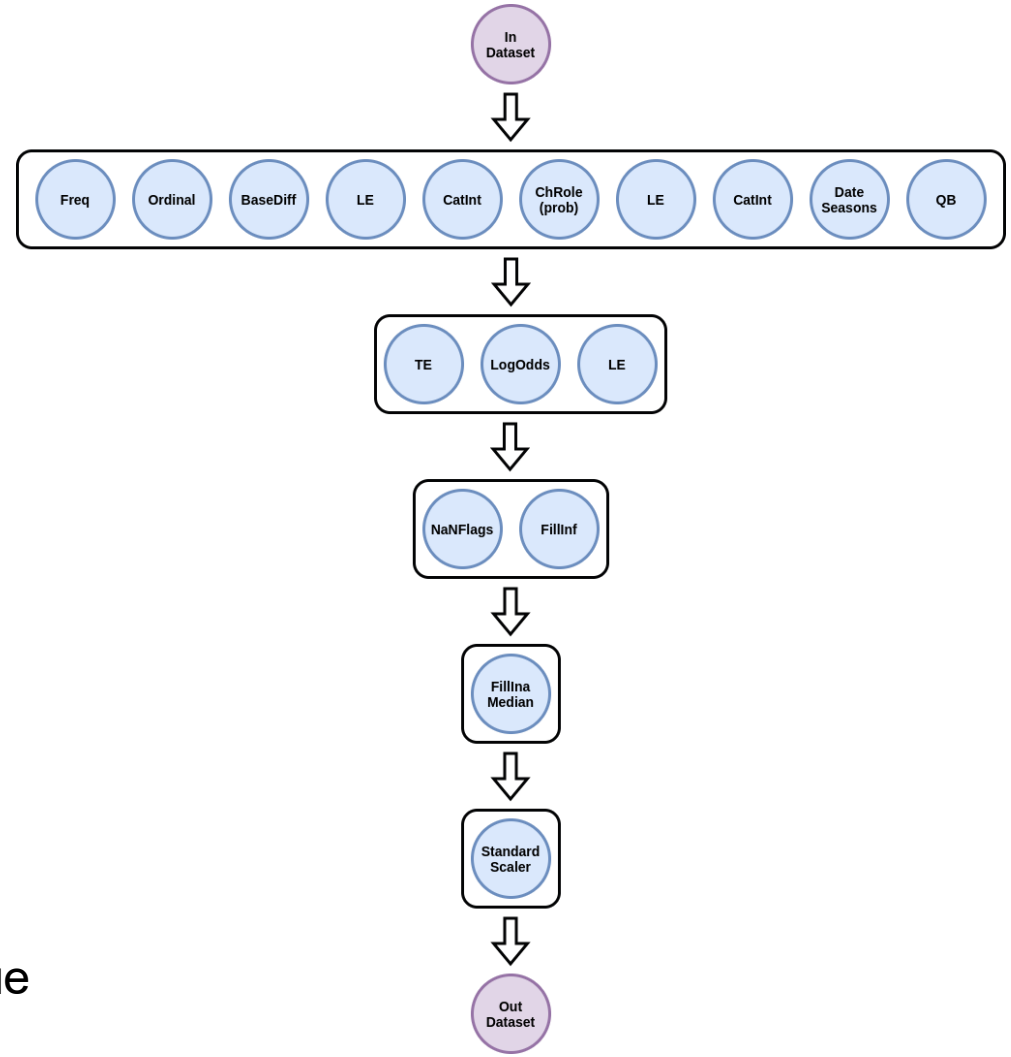


Когда лучше всего кэшировать данные ?

Оптимизация кеширования в пайплайнах обработки: слои

Варианты

- Не кэшировать? Возможно повторение дорогостоящих вычислений.
- Кэшировать после каждого энкодера / трансформера? Слишком накладно.
- Кэшировать через регулярные промежутки? Можно нарваться на дорогие повторные вычисления.
- Выделить слои независимых энкодеров и кэшировать только после них?

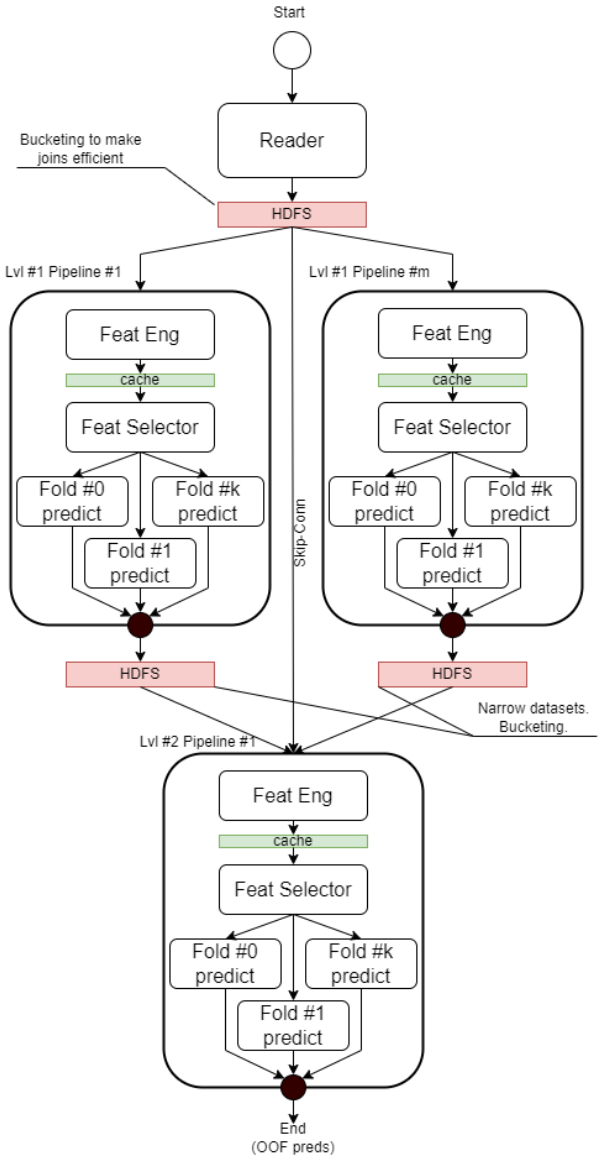


Решение

- Строим граф энкодеров используя зависимости
- Применяем топологическую сортировку и получаем слои
- Вставляем проекцию в конце слоя, убирая колонки, которые не будут далее нужны
- Вставляем кэшер после каждого такого слоя

Нет повторений и объем кэширования сокращен (но можно лучше)

Сокращение сложности планирования

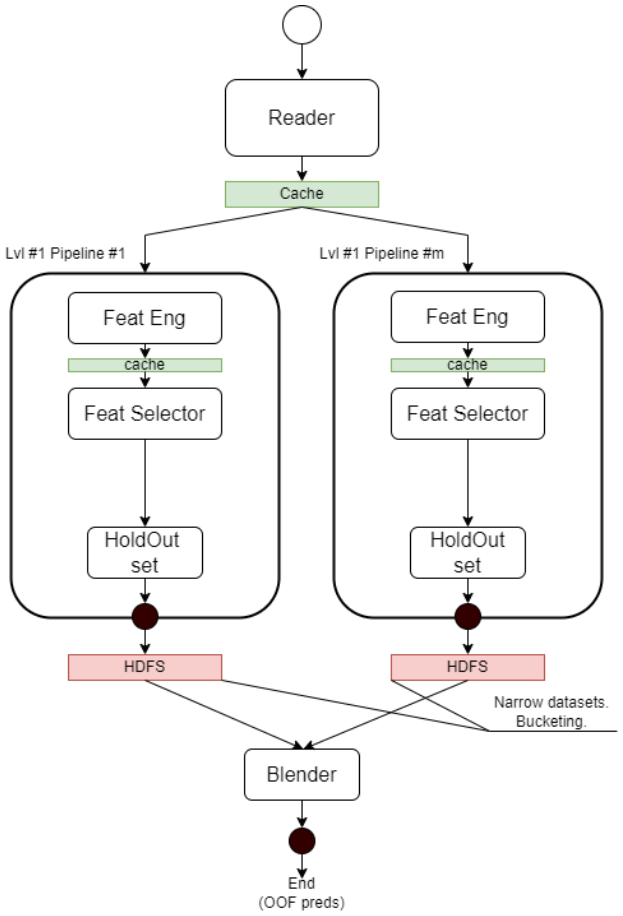


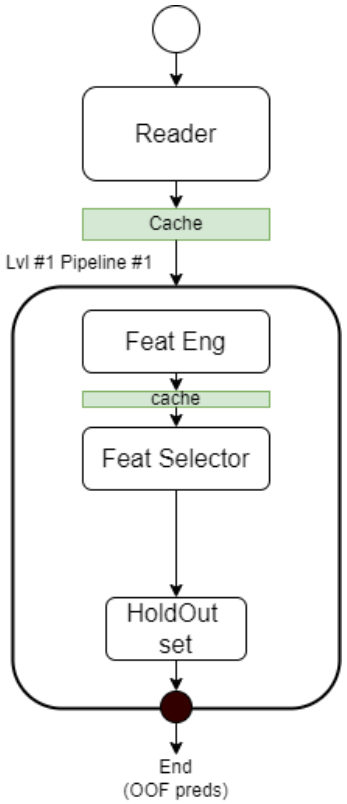
- Long-lineage problem решается с помощью пруннинга плана.
- Пруннинг можно сделать разными способами, но наиболее оптимальное для наших условий- бакетирование промежуточного датафрейма, что подразумевает запись/чтение в/с HDFS:
  - длинный план сокращается
  - длинные зависимости в плане тоже прунятся
  - сохраняется информация о бакетировании
  - отказоустойчивость тоже сохраняется (т.к. новый датасет становится “корневым”)
- Но не все места в сценарии AutoML одинаково полезны для пруннига: по возможности, надо пруннить там, где запишется меньше всего данных, но при этом план еще достаточно короткий.
- Предикты отдельных пайплайнов в каждом слое наиболее “узкие”, а после них план существенно разростается из-за join-ов в конце слоя.

Решение

- Вводим сущность (PersistenceManager), которая определяет способ кэширования промежуточного датасета в зависимости от того, в каком месте в пайплайне мы находимся (а в потенциале и от конфигурации самих пайплайнов)
- Трекаем зависимости между датасетами (parent-child отношения).
- Когда промежуточный датасет кешируется, проверяем нет ли закешированного родительского датасета, который можно было бы удалить.
- В местах потенциального бренчивания ставим блок на удаление родительского датасета. Производим удаление датасета при выходе из соотвествующего контекста.

Сокращение сложности планирования





- Оптимальное кеширование и пруннинг зависит от конфигурации сценария AutoML.
- Более простые сценарии (например, нет второго уровня) или “узкие” датасеты с малым количеством фичей могут не требовать пруннига в части мест в сценарии или вообще обходиться кешированием.
- SLAMA предоставляет опции для конфигурирования способов кеширования промежуточных датафреймах в ключевых местах.
- В зависимости от настроек сценария и датасета AutoML мог бы конфигурировать не только содержательную составляющую своих преобразований, но и способ как он их выполняет, составляющую производительности (Work-In-Progress)

Распределенные компоненты AutoML.

Подготовка данных: обработка категориальных фич

```
{
    "contracted": 1,
    "rejected": 2,
    "checking": 3,
    ...
    "some random status": 35
}
```

Для базового кодирования категориальных фич есть LabelEncoder, FreqEncoder, OrdinalEncoder и TargetEncoder

В Spark есть StringIndexer, который возвращает порядковый номер в качестве лейбла при сортировке по частоте или по алфавиту, но ...

... для FreqEncoder нам нужны сами частоты

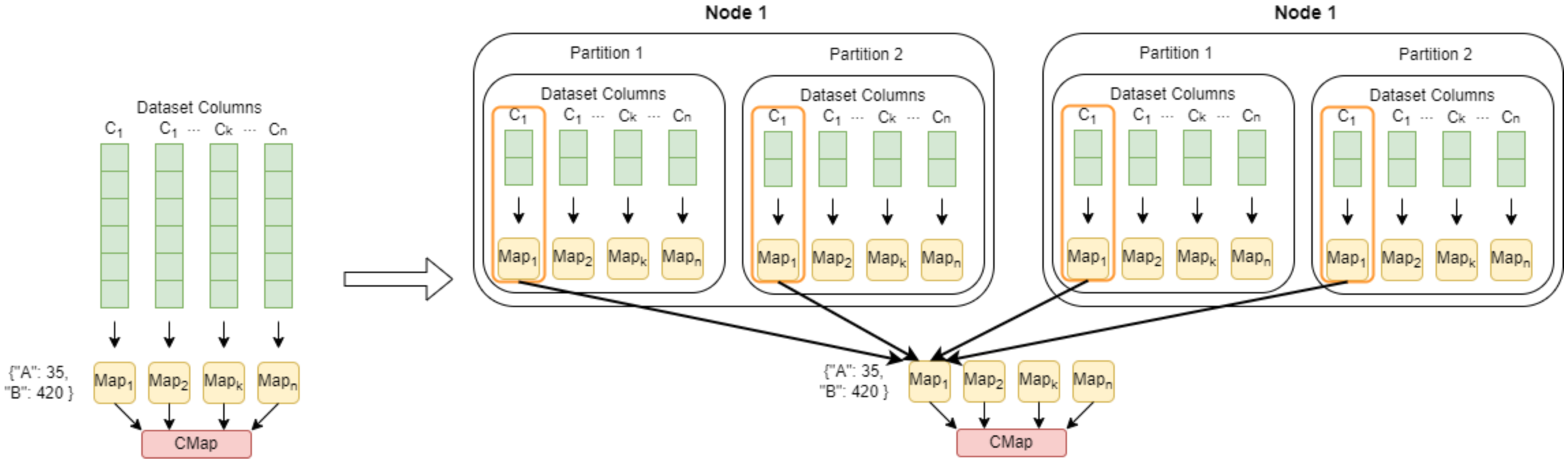
... для OrdinalEncoder нам нужен ранк

... нам нужно отсекать категории со слишком низкой или слишком высокой частотой встречаемости

Можно расширить StringIndexer до нужной реализации, наследовавшись и добавив нужную функциональность. И добавив нужную обертку для PySpark.

…но есть особенности.

Подготовка данных: обработка категориальных фич



Агрегация кат. фич в случае LAMA:
один словарь на колонку

Агрегация кат. фич в случае SLAMA:
один словарь на колонку *на партицию*

20 байт X 1 000 – 10 000 категорий X 100 – 1 000 колонок X Nпартиций

Размер Map1 зачастую равен размеру частичной Map1 из-за равномерности распределения категорий.

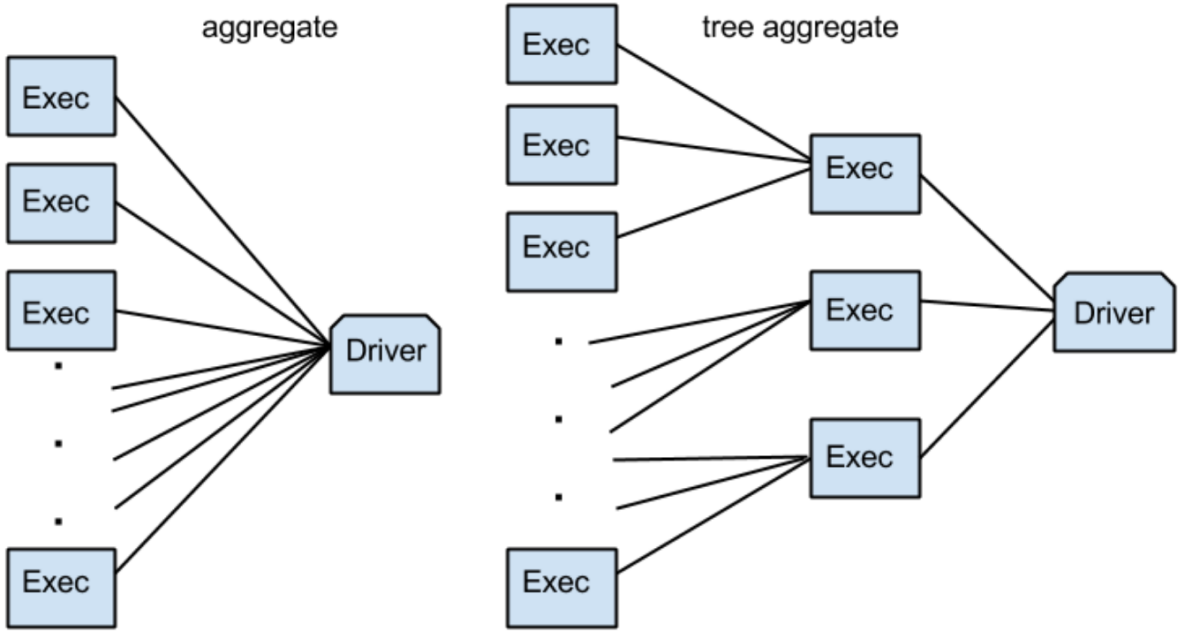
Чем больше размер распределенного приложения, тем больше словарей нужно агрегировать:
больше данных передать, больше вычислений сделать

ИТМО

Обработка категориальных фич: как сделать эффективно?

- Не обязательно использовать само значение категории, т.к. оно может быть очень длинным и занимать много байт, вместо это прибегаем к Hashing Trick, что сведет размер потребляемой памяти до 8 или 4 байт на запись (long / int). Но нужна поддержка на стороне трансформера.
- При реализации “в лоб” при поколоночной обработке (через groupby и agg) из PySpark кластер используется не эффективно из-за не догруженности и добавляются оверхеды на запуск job (планирвощик, инициализация, запуск), что аккумулируется при большом числе колонок. *Если спуститься на уровне Scala можно сделать лучше.*
- На уровне Scala доступен специальный класс Aggregator (не доступен в PySpark), делаем кастомную UDAF которая позволит обработать все колонки за раз в одной job-е. Кроме того, задаем cryo ser/deser, что существенно сокращает объем передаваемых данных.
- Вместо HashMap для подсчета можно использовать OpenHashMap, что займет больше RAM, но будет работать в *2-3 раза* быстрее при частых вставках. Есть еще реализация LongHashMap, которая может быть еще быстрее.
- Вместо стандартного aggregate можно использовать TreeAggregate, что может сократит время выполнения в 1.5 - 2 раза (зависит от глубины дерева и объема передаваемых данных)

Обработка категориальных фич: Tree Reduce pattern



StringIndexer использует Aggregator для получения конечного словаря категорий.

Конечный словарь требует сбора всех словарей на одном узле и обработка происходит в один поток

Если словарей много и/или много категорий – это длительный процесс.

Можно обойти применив treeAggregate вместо просто aggregate: итеративный сбор словарей в один с уменьшающимся числом партиций

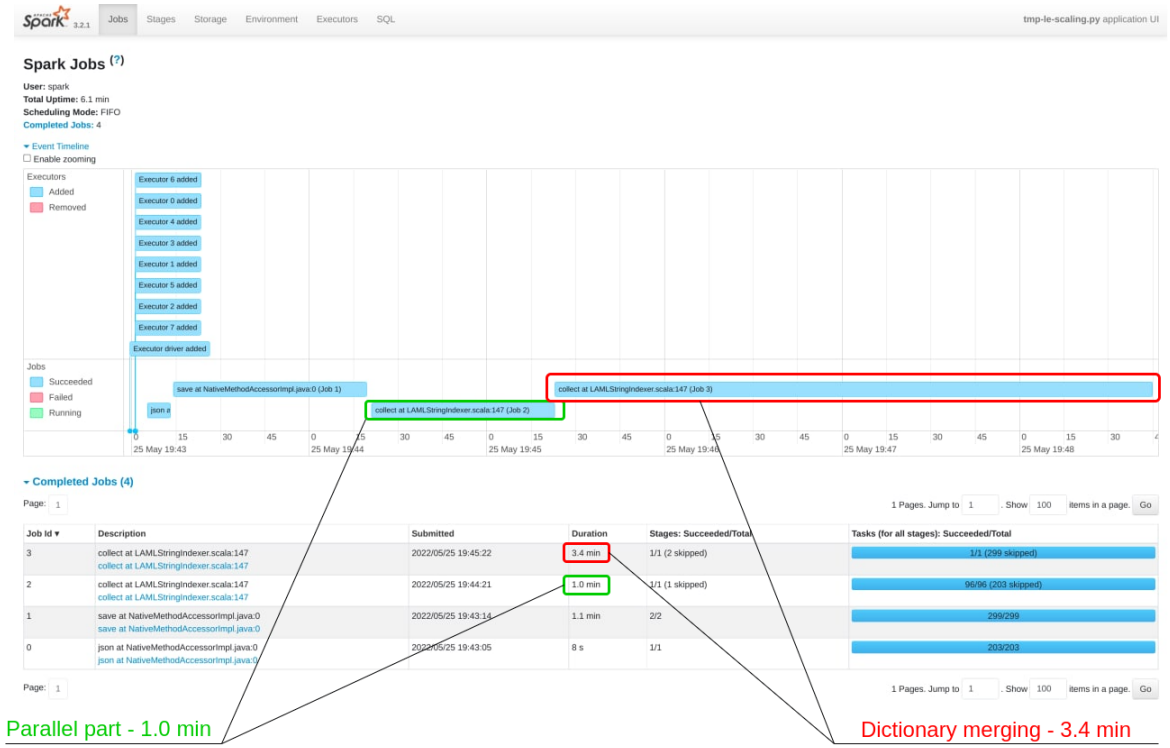
Полезно также для CatIntersectionsEncoder

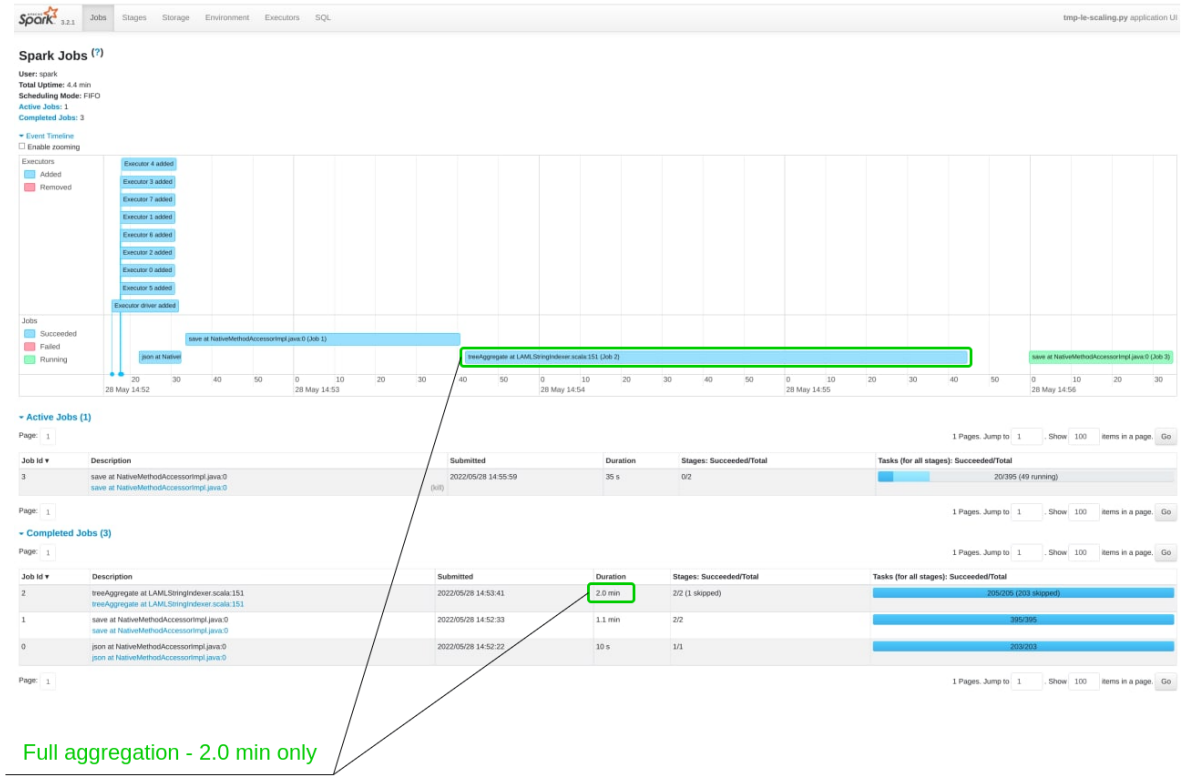
В treeAggregate возможны несколько раундов передач данных

https://umbertogriffo.gitbook.io/apache-spark-best-practices-and-tuning/rdd/treereduce_and_treeaggregate_demystified

Обработка категориальных фич: Aggregator

Проблема: при большом количестве категорий (и/или партиций), базовая реализация StringIndexer может тратить больше времени на НЕ параллельную агрегацию словарей, чем на обработку самих данных.

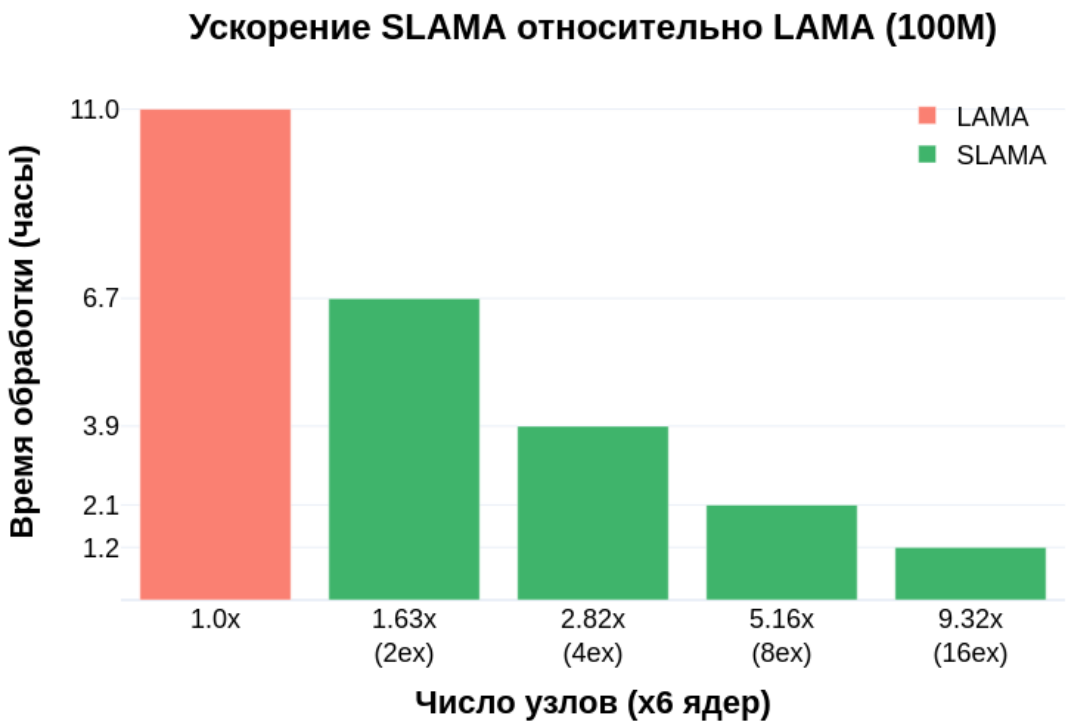




Время работы сократилось в 2+ раза.

Улучшение масштабируемости и 3D parallelism.

Ограничение масштабируемости



- Изначальный вариант SLAMA опирается на классический data-parallel подход.
- Но при избытке ресурсов и не достаточно большом размере датасета не получается эффективно использовать эти ресурсы.
- Алгоритмы ML обычно не масштабируются линейно с ростом данных, эффективность снижается с ростом числа доступных ресурсов.
- Возможное решение – введение возможности использования **гибридного подхода**, сочетающего возможности data-parallel и compute-parallel подходов.

За счет горизонтального масштабирования на больших датасетах SLAMA работает быстрее LAMA, но на средних датасетах эффективность масштабирования быстро снижается.

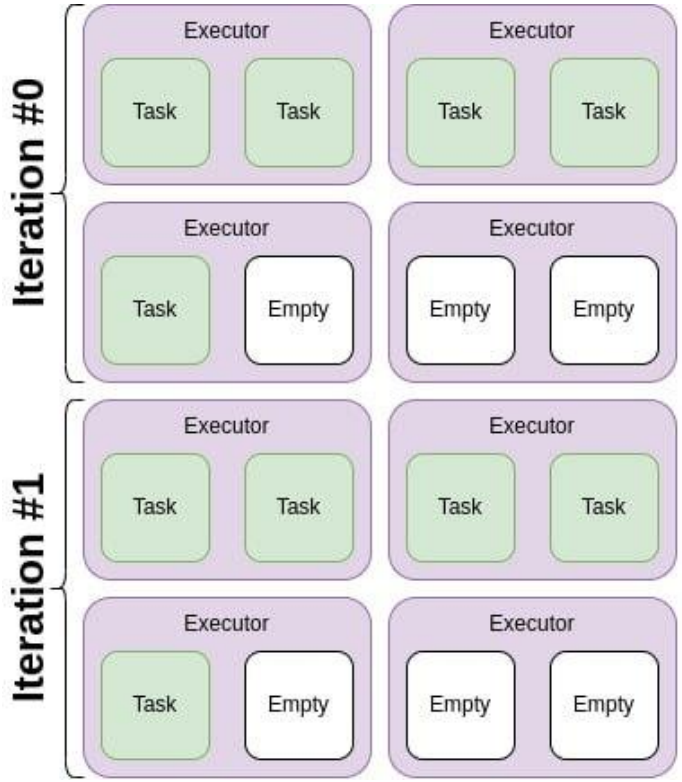
Data- vs compute- parallel режимы: 3D parallelism

- Внутри пайплайнов можно запускать несколько параллельных job с обучением моделей на разных фолдах или для подбора оптимальных параметров –> нужен shuffle, не поглотит ли он все бенефиты?
- Параллельность по данным, но с меньшей и более эффективной степенью, сохраняется внутри каждой из job -> нужно подобрать оптимальное сочетание
- Параллелить можно и по пайпам при достаточности ресурсов и времени -> нужно учесть уложимся ли мы дедлайн

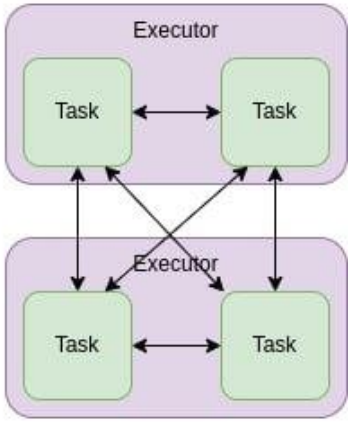
Все вместе – более эффективный 3D parallelism…
…но требующий грамотной настройки параметров в зависимости от данных и доступного времени

Обмен данными в ML алгоритмах

Некоторые ML алгоритмы работают по MPI-подобной схеме обмена данными на итерациях. Например: lightgbm, CatBoost, Neural Networks на PyTorch / Keras.



Итеративный алгоритм, разбитый на job-ы.

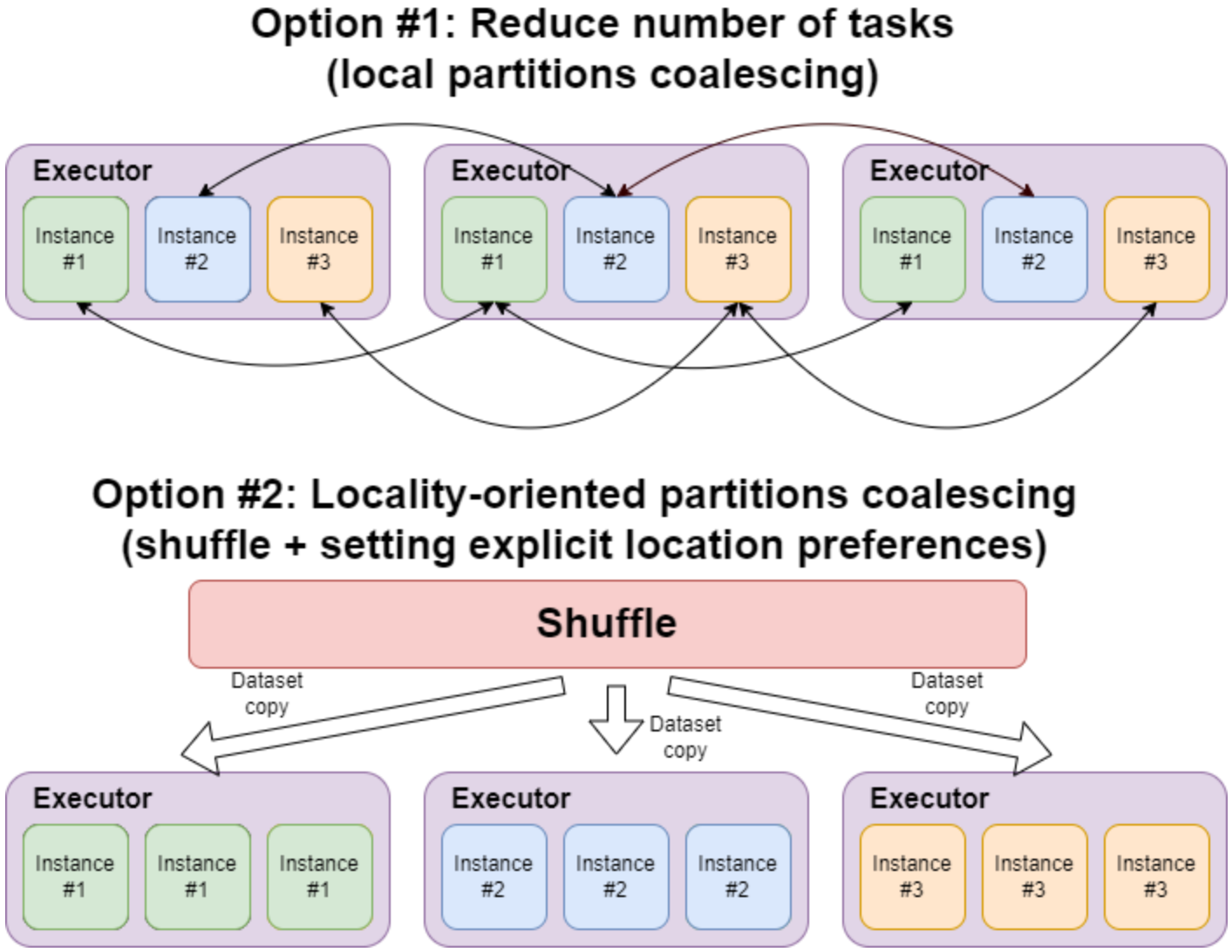


Алгоритм с MPI-подобным обменом данных.

В случае с MPI-like может быть: чем больше размер “мира”, тем больше оверхедов на синхронизации.

Гибридный data/compute parallel для MPI-like

Как можно реализовать data/compute parallel режим в случае MPI-like алгоритмов?

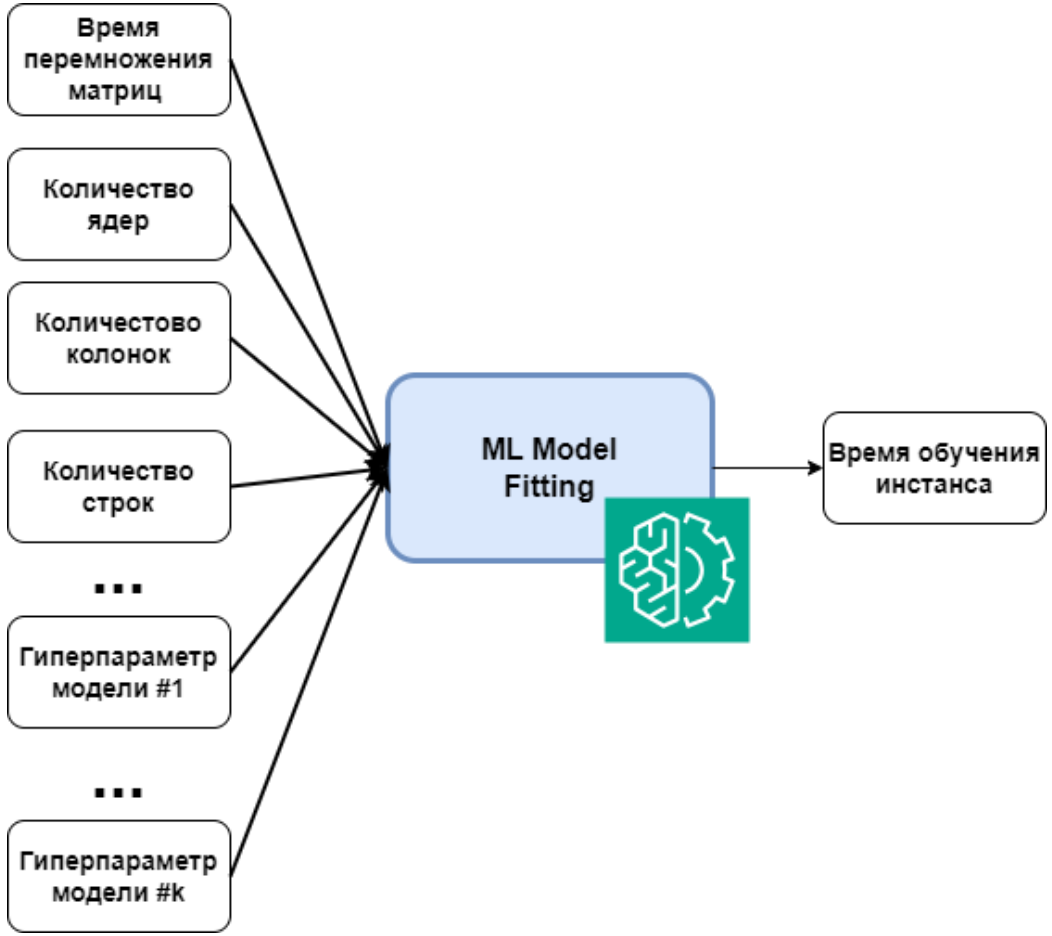


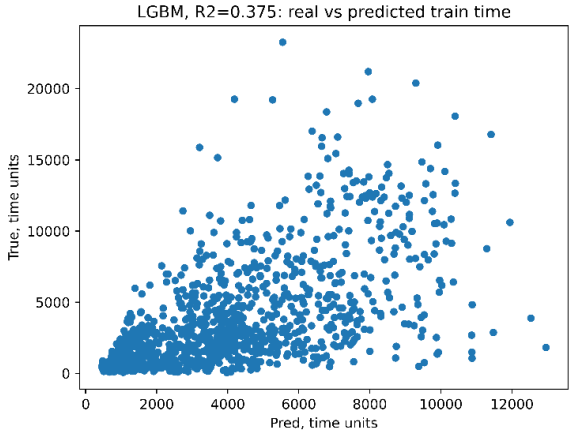
Простое уменьшение числа тасок.

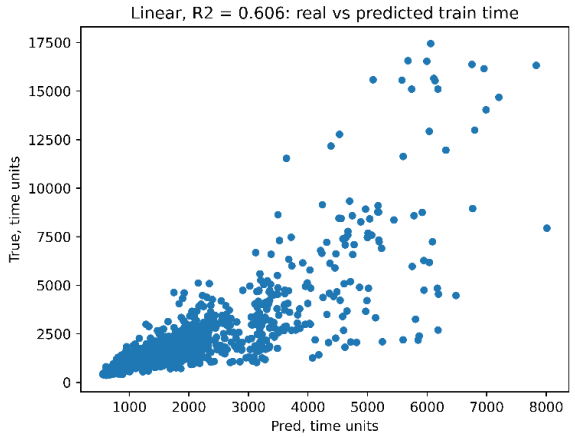
“Умный” coalescing.

POpt: прогнозируем время выполнения конфигураций и выбираем лучшую

- Строим датасет путем сбора статистики о реальных запусках и расширяем его за счет запусков на синтетических данных.
- Обучаем регрессионную модель, которая предсказывает время выполнения.
- Точный прогноз времени не особо интересен, важнее масштаб.

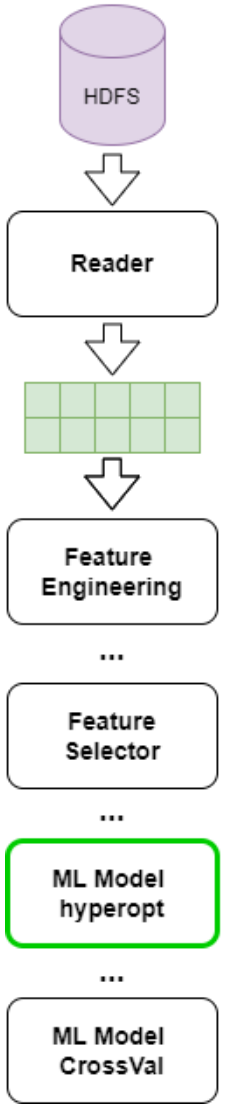


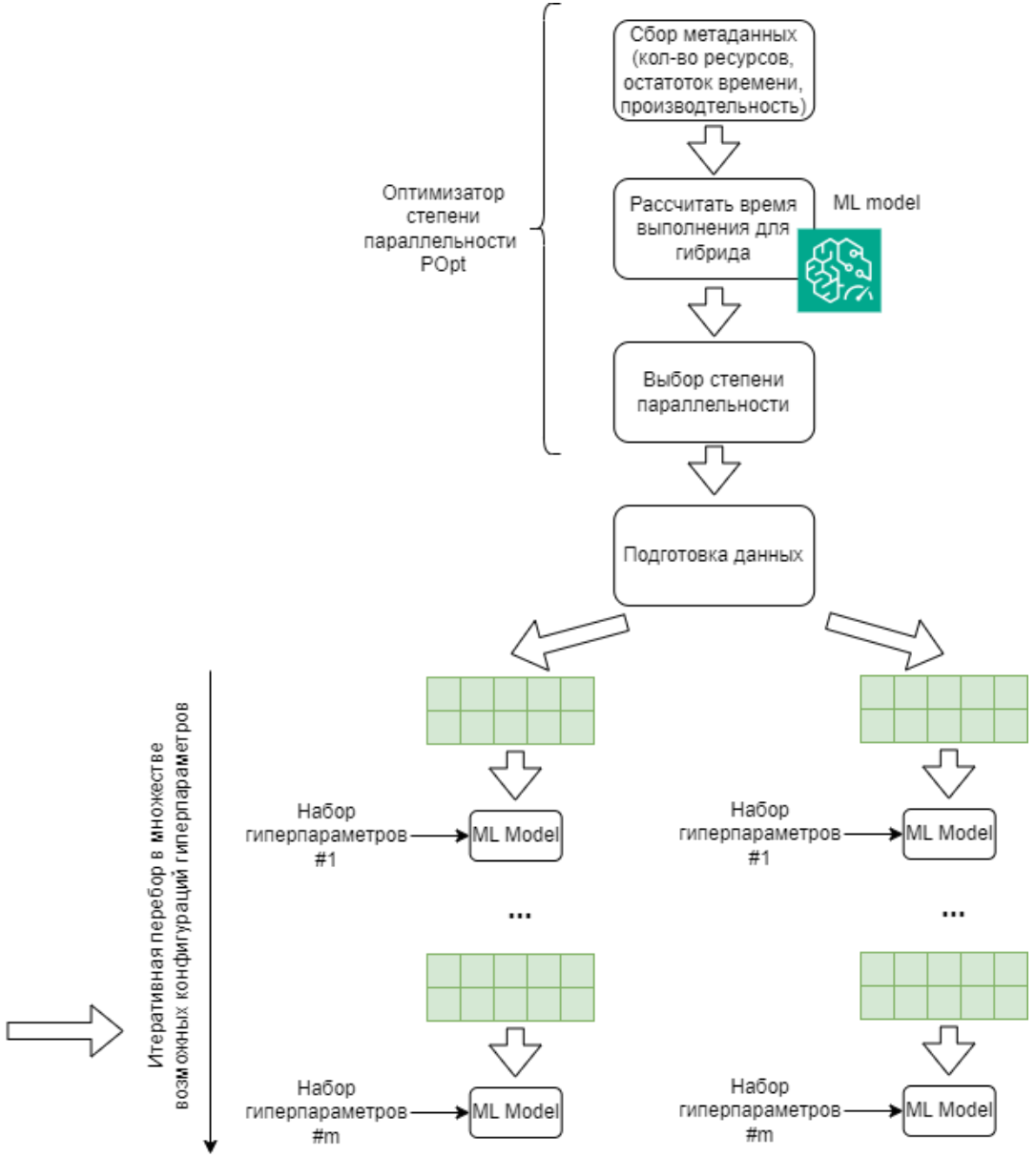




Popt: применение

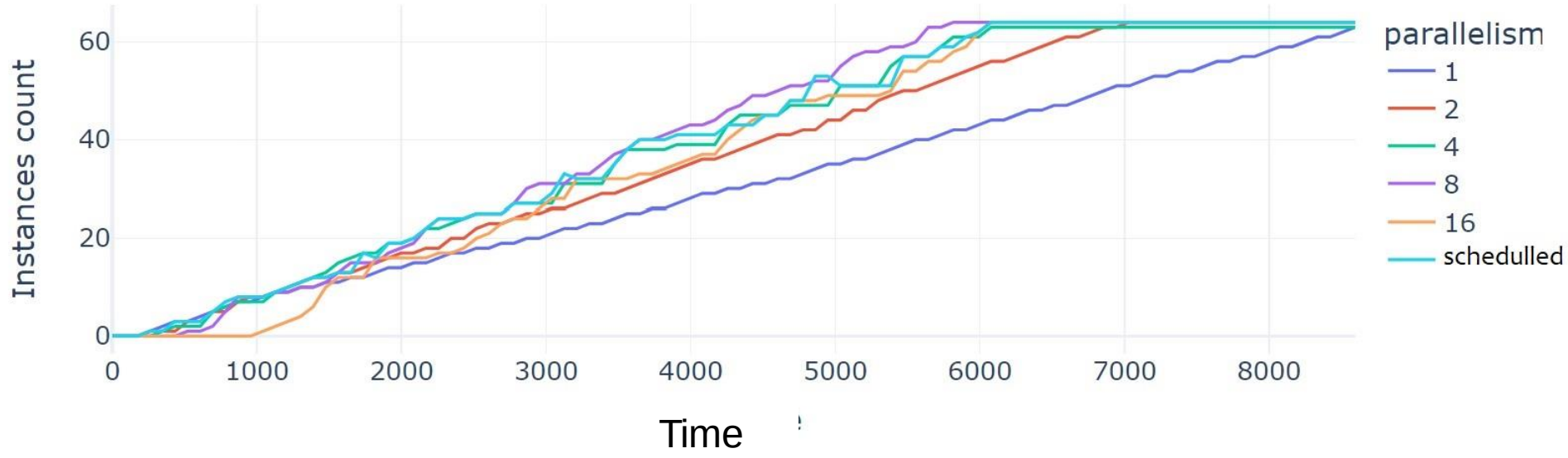
- Этапы пайплайна, где возможно параллельность могут применять Popt, чтобы подобрать оптимальную степень параллельности.
- Применяем ML модель, чтобы спрогнозировать время выполнения при том или ином количестве ресурсов на инстанс обучения.
- Учитываем время шаффла (подготовки данных) и потенциальную латентность.
- Выбираем конечный вариант и отправляем на расчет.





POpt: выигрыш

Датасет на **25** млн



В зависимости от доступного времени / количества инстансов к расчету:

- сначала эффективнее считать полностью в data-parallel режиме
- при 8-10 инстансах параллелизма 2 эффективнее
- при 20+ инстансах параллелизм 8 эффективнее

Коэффициент масштабируемости улучшается с ~9x до ~13x для 16 экзекьютеров.

Выводы

Выводы

- Спускаться до уровня Java/Scala можно и нужно, чтобы написать эффективный код для PySpark. Эффективные агрегации и тонкое управление процессами вычислений доступно только там.
- Современные реалии требуют адаптации фреймворков и большей гибкости от них, предоставления большей поддержки гибридным режимам вычислений, что в свою очередь создает больше параметров для настройки.
- 3D-parallelism одна из таких адаптаций, позволяющая добиться улучшения производительности до 40% (по результатам экспериментов)
- Чтобы добиться большей производительности и/или упростить жизнь пользователю можно прибегать к интеллектуальной настройке параметров за счет профилирования, накопления статистики и **применения методов ML / AutoML уже к данным, генерируемых самим фреймворком и/или приложением на его основе**.
- Развитие в этом направлении может потребовать развития и самой экосистемы JVM: улучшения observability JVM-процессов, сбора данных и сопряжения их с высокоуровневым поведением приложений; развития “нативных” инструментов обучения и сервинга моделей ML; фреймворков AutoML.

По всем вопросам пишите на почту: alipoov.nb@gmail.com

SBER AI LAB

Лаборатория Искусственного Интеллекта

ITMO

Центр “Сильный ИИ в промышленности”