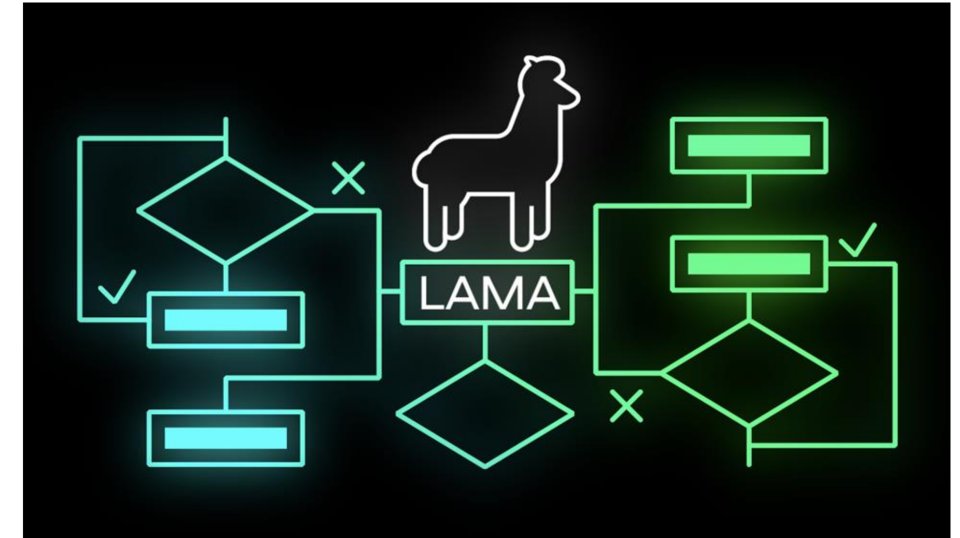
# SLAMA: тонкости масштабируемости AutoML решения на Spark.

Бутаков Николай, Университет ИТМО

Май 2023

# 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)
- Добавляется новый гиюридный data/compute parallel режим для повышения эффективности обработки средних по размеру датасетов



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

# SLAMA: пример типичного AutoML сценария

```
task = SparkTask(task_type)
train_data, test_data = prepare_test_and_train(spark, path, seed)

automl = SparkTabularAutoML(
    spark=spark,
    task=task,
    timeout=3600 * 3,
    general_params={"use_algos": [["lgb", "linear_l2"], ["lgb"]]},
    tuning_params={'fit_on_holdout': True, 'max_tuning_iter': 101,
                   'max_tuning_time': 3600}
)

oof_predictions = automl.fit_predict(train_data, roles=roles)

score = task.get_dataset_metric()
metric_value = score(oof_predictions)
```

# Масштабируемый AutoML: мотивация

Когда может быть полезна масштабируемость в AutoML?

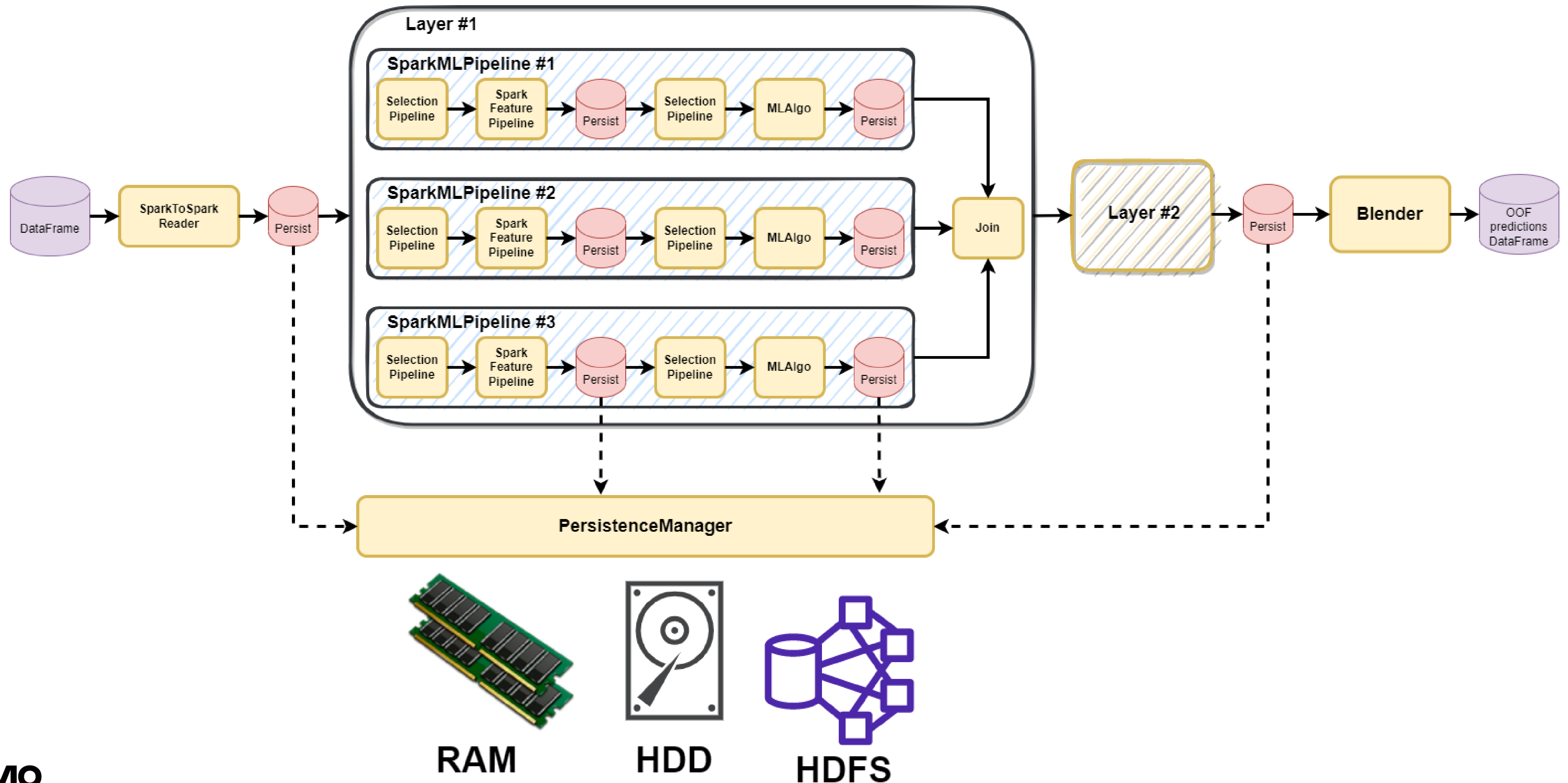
- Перенести решение на больший объем данных без особых усилий
- Больше железа – быстрее получение результата
- Избежать недостатка памяти: время обучения устраивает и на одной машине, а вот OOM – нет.
- Больше доступности ресурсов: можно не стоять в очереди на “жирный” сервер, заменив его набором легкодоступных меньших машин

# Особенности AutoML решения

Чем AutoML пайплайны отличаются от ETL и др.?

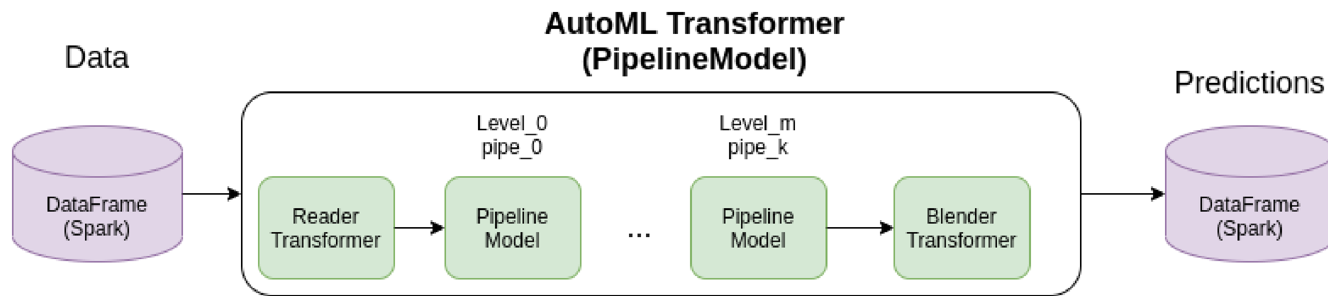
- часто много колонок (широкие фреймы)
- длинные преобразования
- сложная структура преобразований, с ветвлениями и объединениями
- сочетание как типичной ETL нагрузки (feature processing), так и многократного обучения моделей ML (cross-fold, hyperparams tuning, iterative feature importance selection)
- разные подходы к реализации обучения ML моделей – часто MPI-like
- некоторые методы обработки фич склонны к тяжелым агрегациям
- таймеры и дедлайны, динамически влияющие на структуру пайплайнов

# Структура AutoML пайплана в SLAMA




Все основные “строительные блоки” LAMA реализованы в SLAMA.

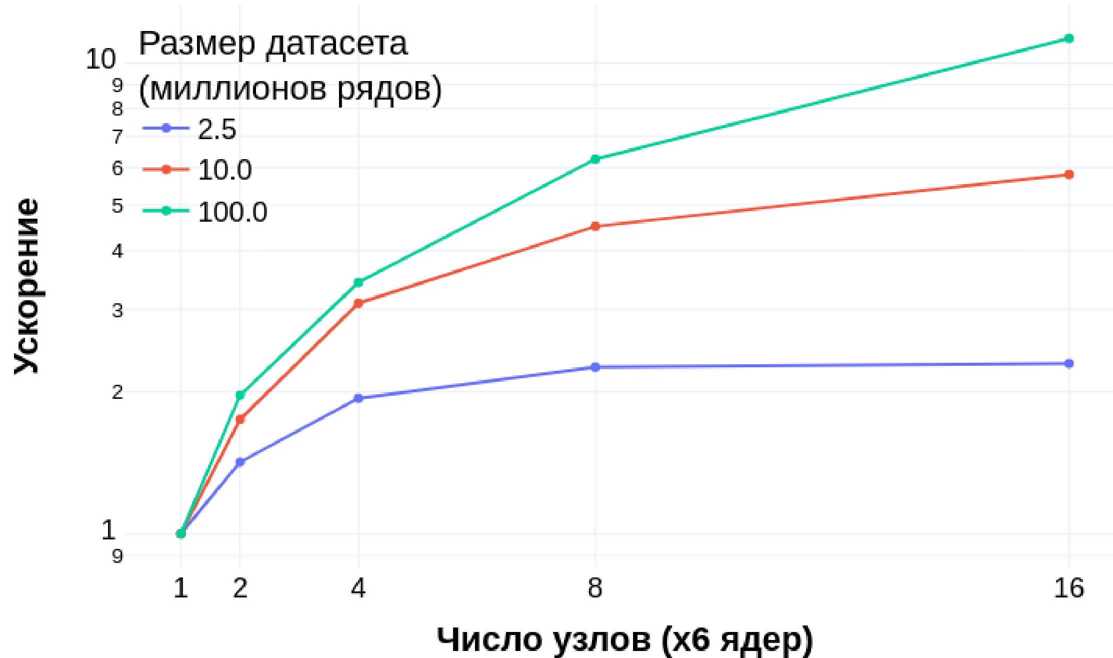
# Результирующий трансформер




- Только линейное преобразование на инференсе
- Нет кэширования или чекпоинтинга
- Для LightGBM возможна конверсия в ONNX модель (ускорение в 2-5 раз)
- Сохранение / загрузка пайплайна на локальный диск, HDFS и т.п.

Только последовательность линейных преобразований в конечном AutoML трансформере.

# 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

С ростом данных масштабируемость растет.

ИТМО

# Особенности обработки категориальных переменных

# 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

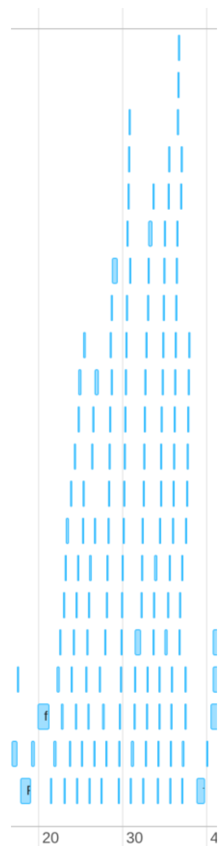
... для OrdinalEncoder нам нужен ранк

... 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.

# Обработка категориальных фич: “наивный”подход

Проблема: при реализации “в лоб” при поколоночной обработке (через groupby и agg) кластер используется не эффективно из-за не догруженности.



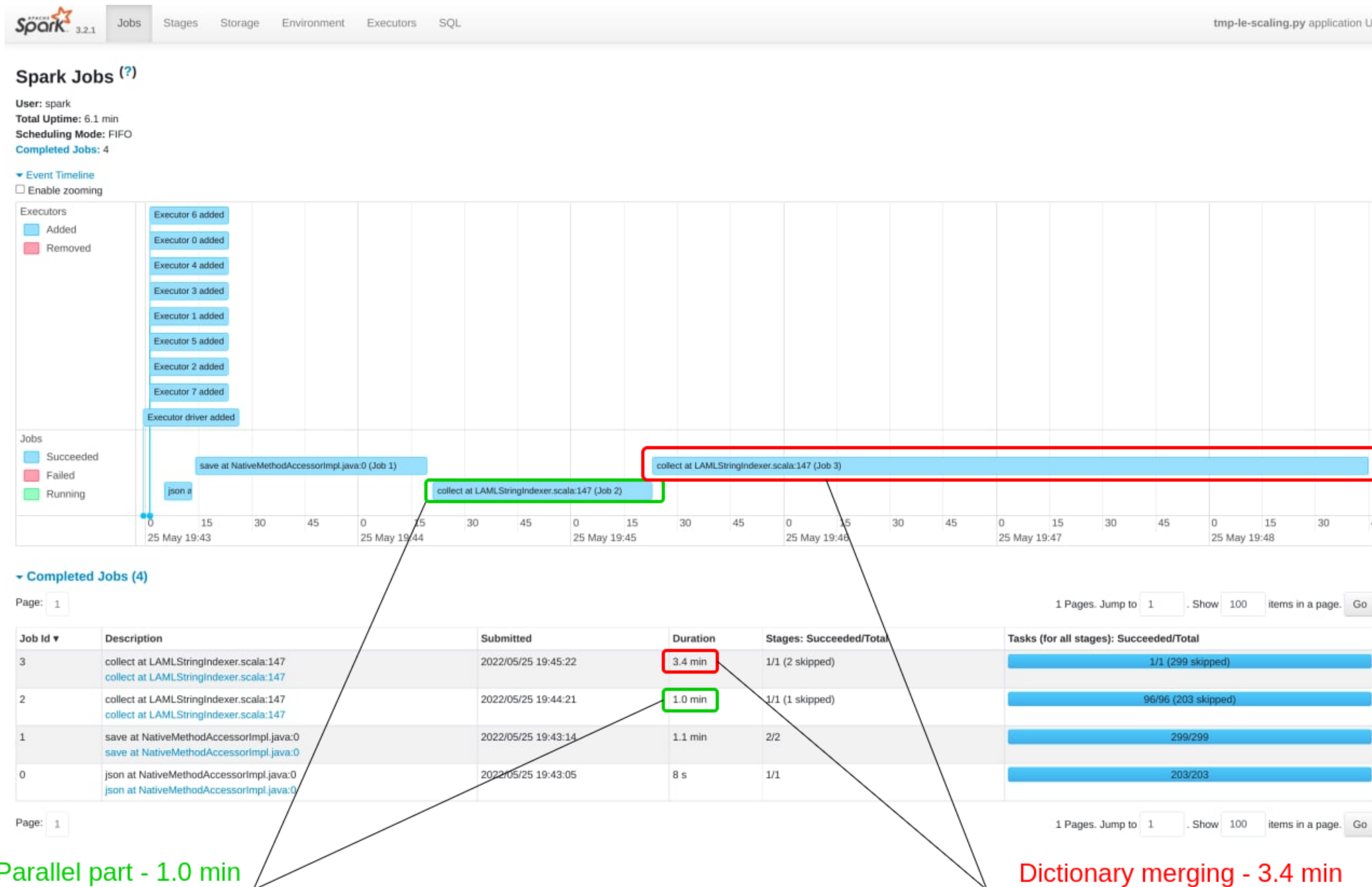

Много независимых job: одна job - одна колонка

```
abstract class Aggregator[-IN, BUF, OUT] extends Serializable {

  A zero value for this aggregation. Should satisfy the property that any b + zero = b.
  Since: 1.6.0
  def zero: BUF

  Combine two values to produce a new value. For performance, the function may modify b and return it
  instead of constructing new object for b.
  Since: 1.6.0
  def reduce(b: BUF, a: IN): BUF

  Merge two intermediate values.
  Since: 1.6.0
  def merge(b1: BUF, b2: BUF): BUF

  Transform the output of the reduction.
  Since: 1.6.0
  def finish(reduction: BUF): OUT

  Specifies the Encoder for the intermediate value type.
  Since: 2.0.0
  def bufferEncoder: Encoder[BUF]

  Specifies the Encoder for the final output value type.
  Since: 2.0.0
  def outputEncoder: Encoder[OUT]
```

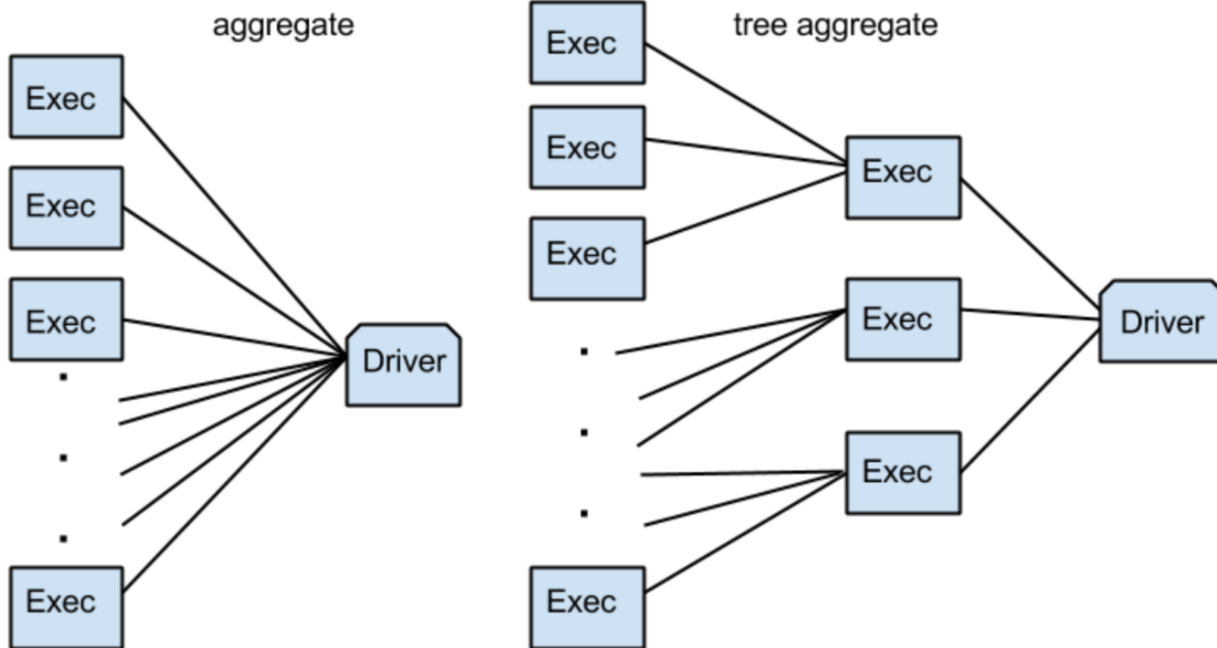
Интерфейс агрегатора, который может обработать все колонки за раз

# 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.



# 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

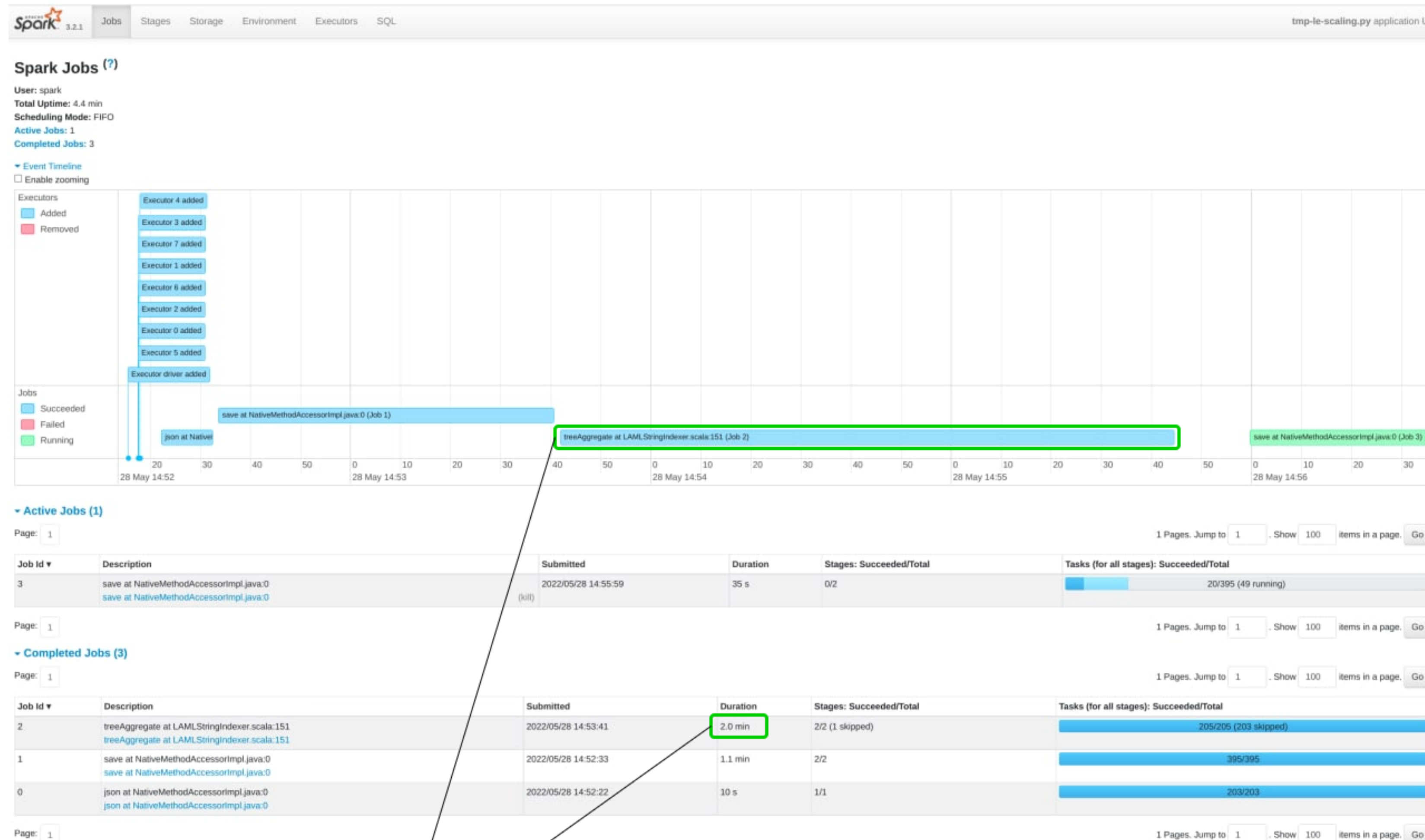
В treeAggregate возможны несколько раундов передач данных

ИТМО

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

# Обработка категориальных фич: Aggregatot + Tree Reduce

На нашем примере время работы сократилось в 2+ раза.

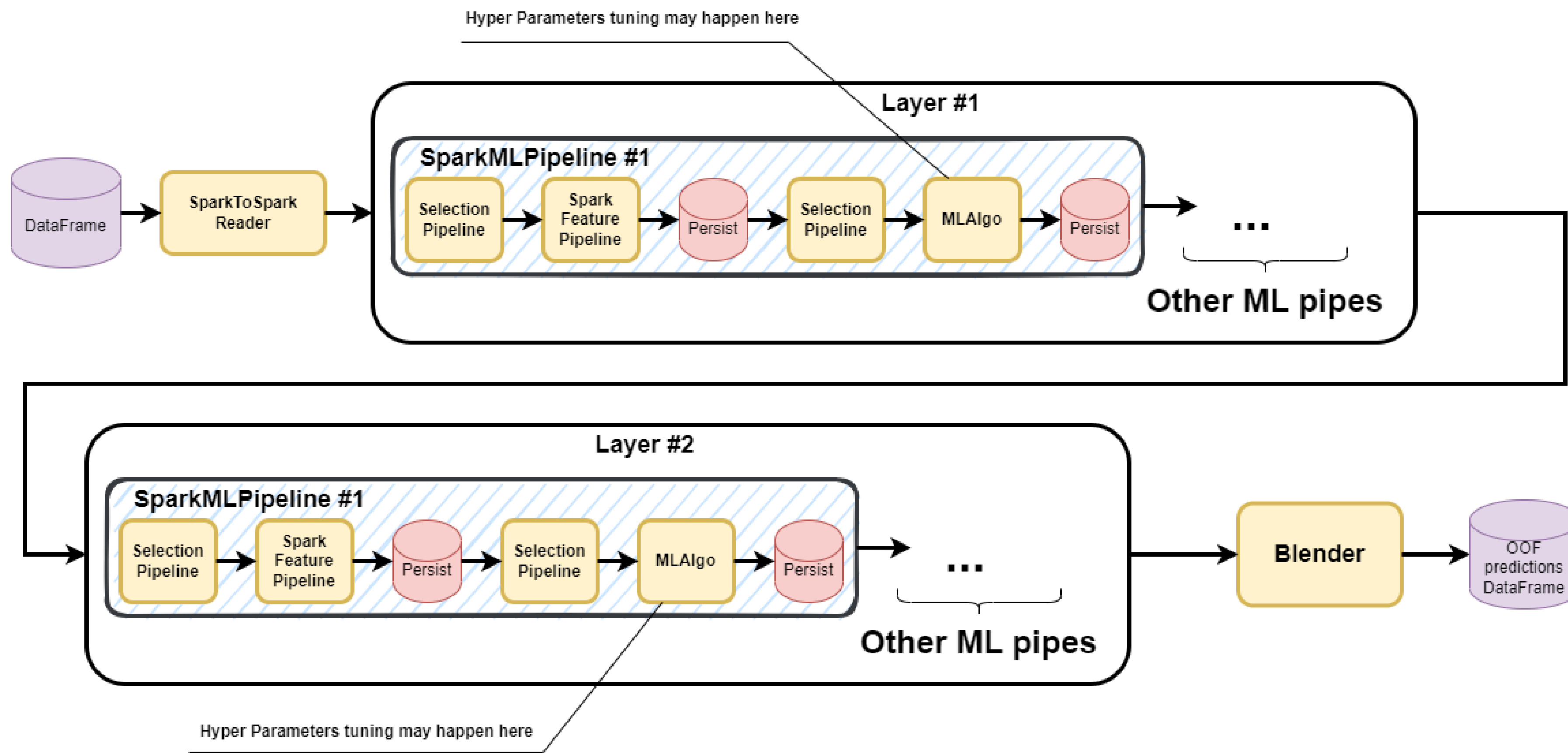



Full aggregation - 2.0 min only

ITMO

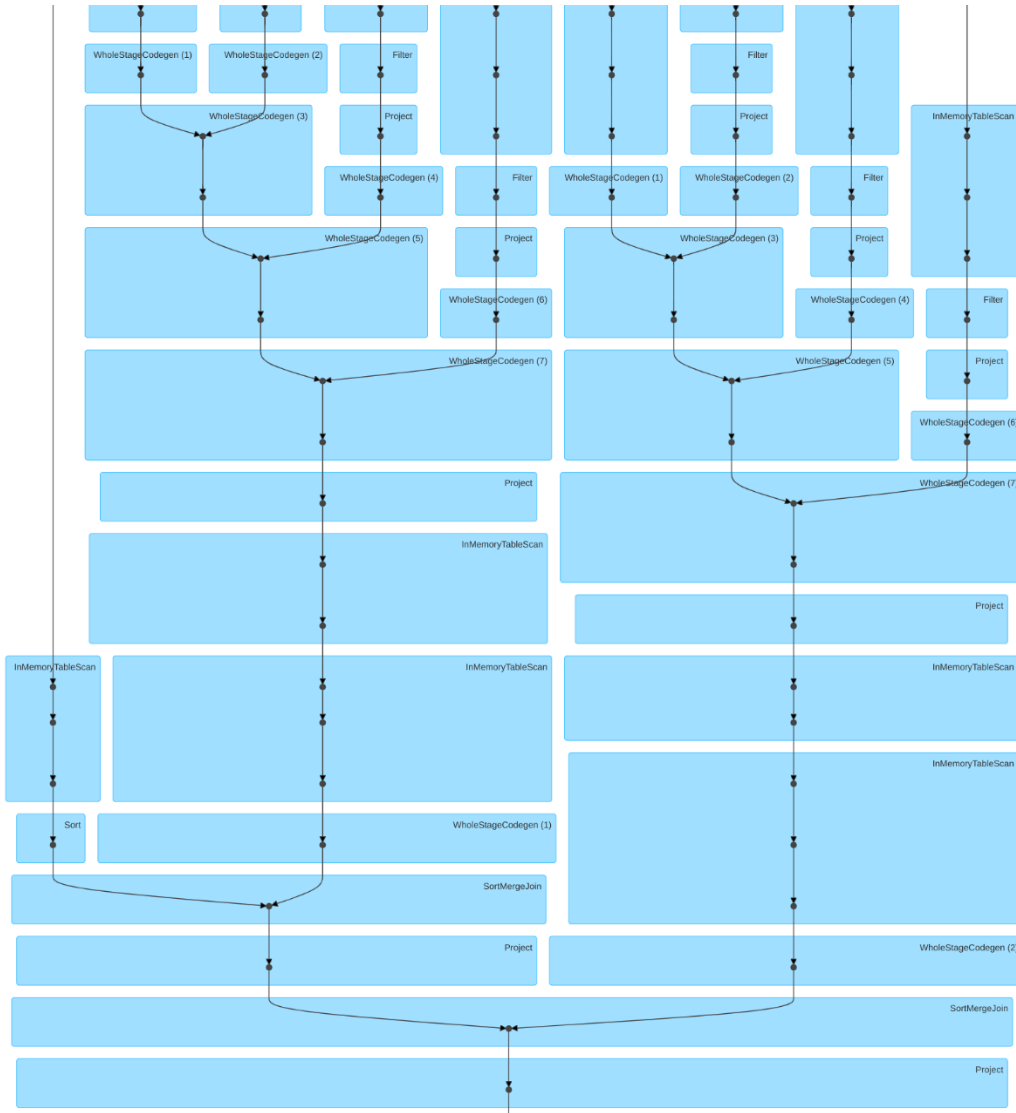
# Особенности кэширования и сохранения промежуточных данных

# Модель выполнения в Data Parallel режиме




- Во время обучения основной датасет проходит серию ПОСЛЕДОВАТЕЛЬНЫХ преобразований, разделяемых кэшированием или чекпоинтами, без шаффлов или джоинов (в иделае).
- Обмен данными присутствует только для агрегаций статистики, нужной для настройки энкодеров и на итерациях во время обучения моделей для их синхронизации.

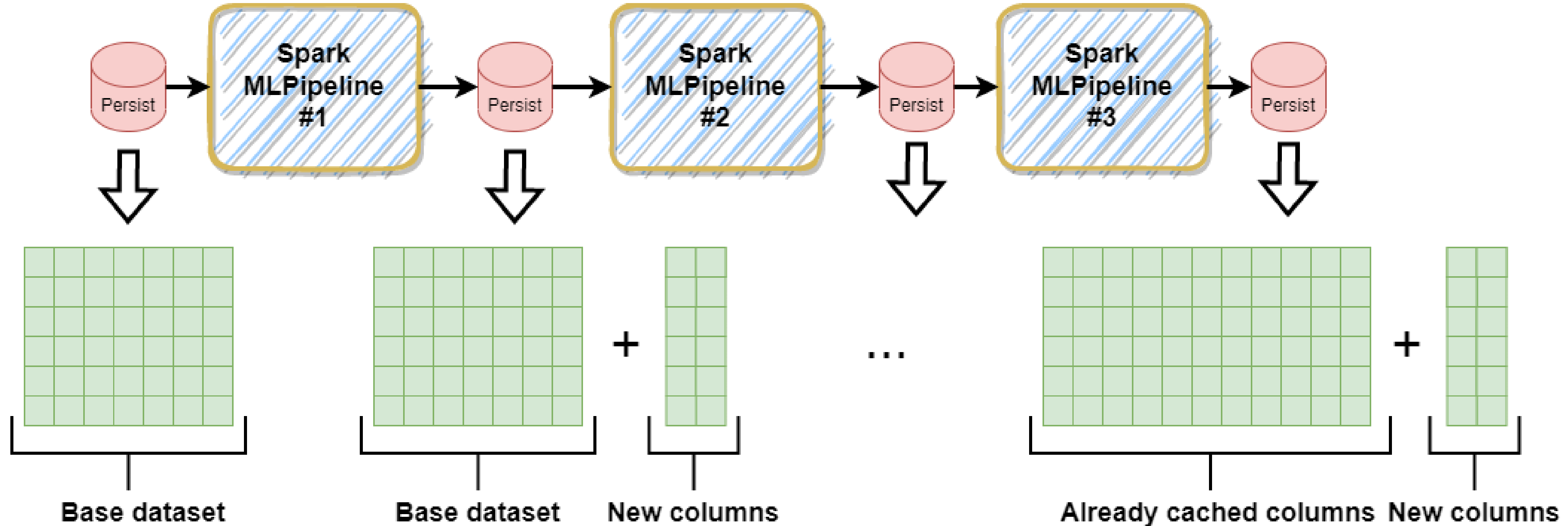
# Объединение пайплайнов в слое




- Собираем новый датасет из Out-of-Folds predictions каждого пайплайна для второго слоя
- Выходной датафрейм каждого пайплайна небольшой по размеру (относительно начального датасета) и очень “узкий” .
- Количество колонок = количество MLPipeline x на количество моделей
- Skip connections: колонки начального датасета тоже могут использоваться на втором слое.
- Можно обойтись только кэширование в памяти без выхода в диск.
- План разрастается и Spark SQL операции второго уровня могут начать тормозить – long plan problem.
- Каждый фолд тоже дает вклад в разрастание плана.

# Объединение (Join) пайплайнов в слое

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



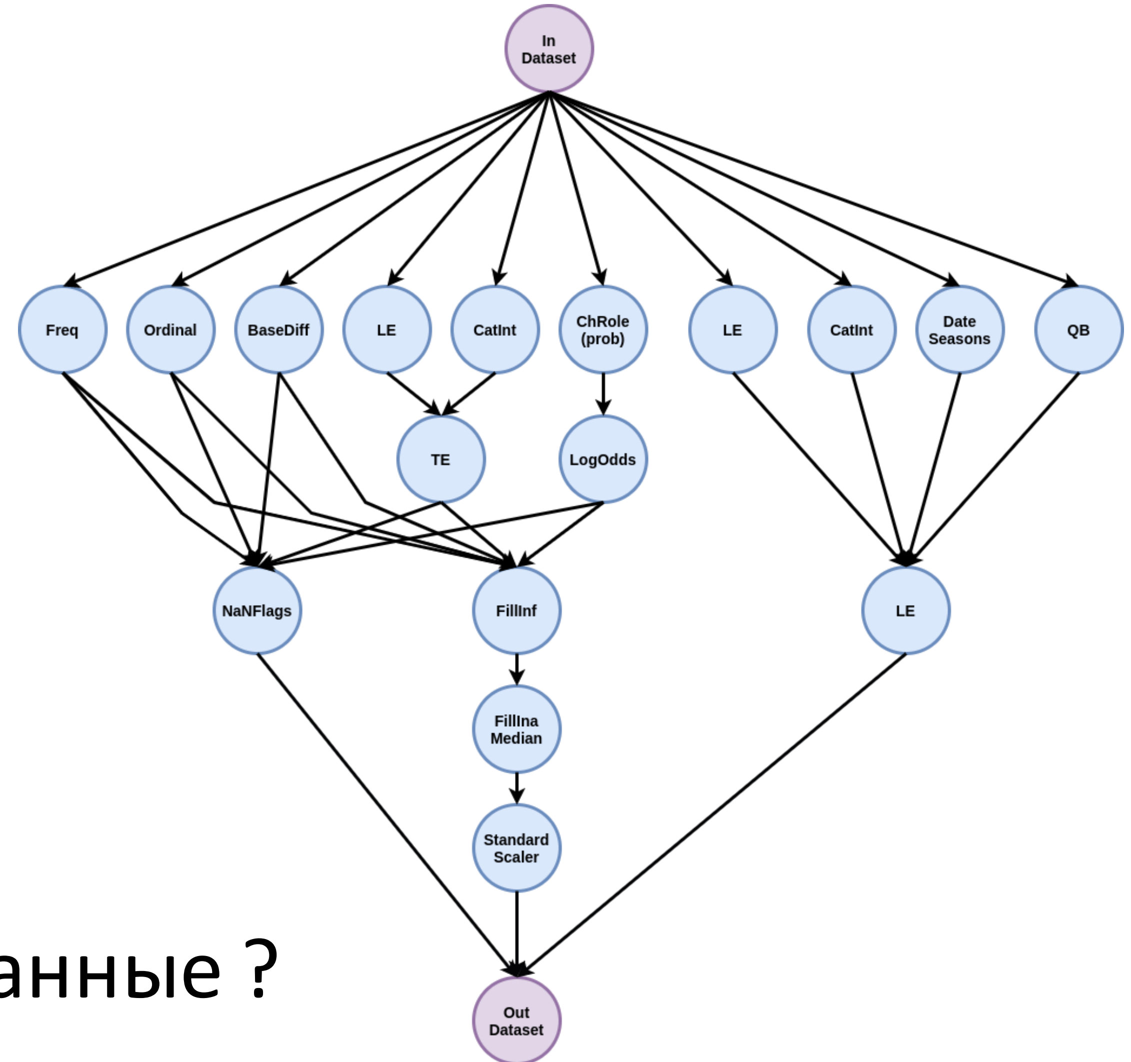

Возможны лишние многократные кэширования начального “широкого” датафрейма, т.к. они нужны каждому следующему пайплайну, и вновь добавляемых колонок.

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

# Построение обработки в SparkFeaturePipeline

## Особенности

- SparkFeaturePipeline 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 ?

ИТМО

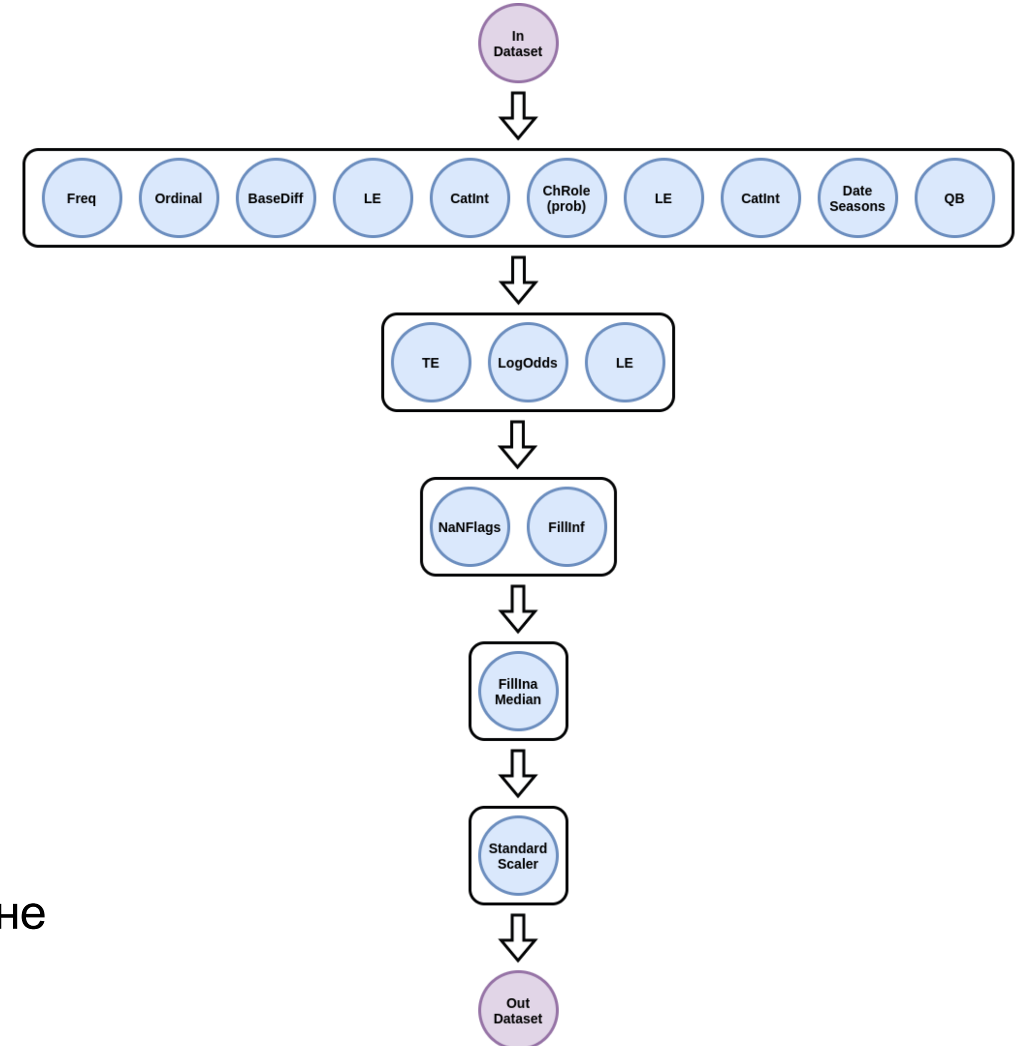
# Оптимизация кэширования в SparkFeaturePipeline

## Варианты

- Не кэшировать? Возможно повторение дорогостоящих вычислений.
- Кэшировать после каждого энкодера / трансформера? Слишком накладно.
- Кэшировать через регулярные промежутки? Снова можем прийти к повторным вычислениям.
- 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

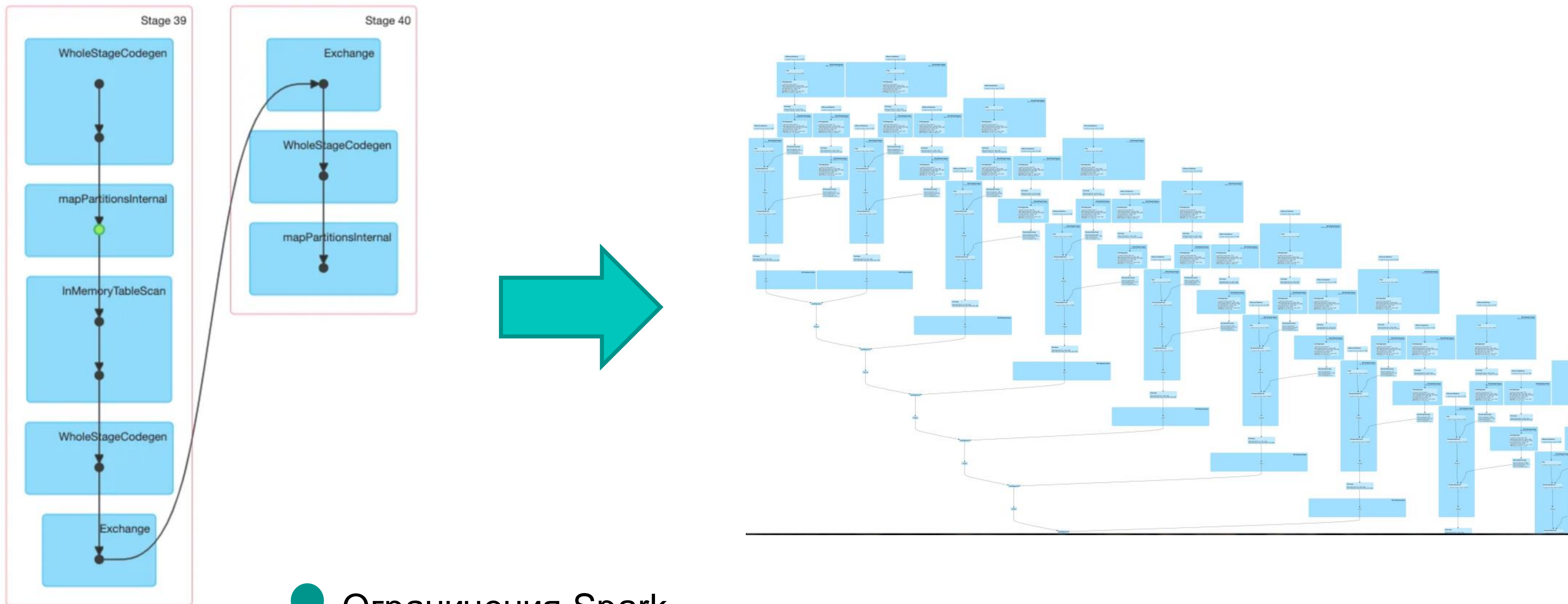



ИТМО

**Нет повторений и объем кэширования сокращен (но можно лучше)**

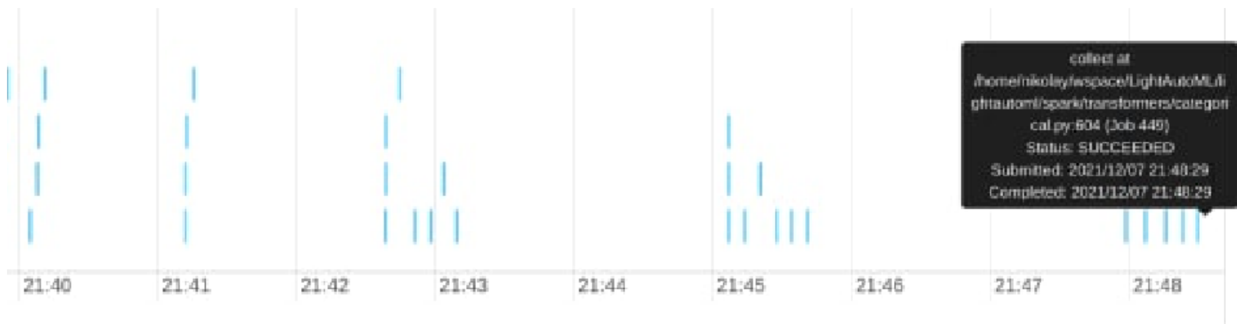
# Проблема №2

С ростом количества стадий в DAG Spark, в т.ч. в линейном DAG, обработка замедляется.




- Ограничения Spark
  - проблема длинного плана (Long Plan problem)
  - проблема длинной истории (Long-lineage problem)

# Ограничения Spark: Long-plan and long-lineage problem




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.

Но есть решение.

ИТМО

[1] Wang, Haoran, Jinliang Wei, and Garth Gibson. “Addressing the Long-Lineage Bottleneck in Apache Spark.” (2018).

# Ограничения Spark: возможные решения

- Пересоздание DataFrame из RDD
  – надежно, быстро, но не решается проблема с вычислительными зависимостями, может быть потеряна важная информация для оптимизатора

```
ds = SparkSession.getActiveSession()\
    .createDataFrame(dataset.rdd, schema=dataset.schema).cache()
ds.write.mode('overwrite').format('noop').save()
```

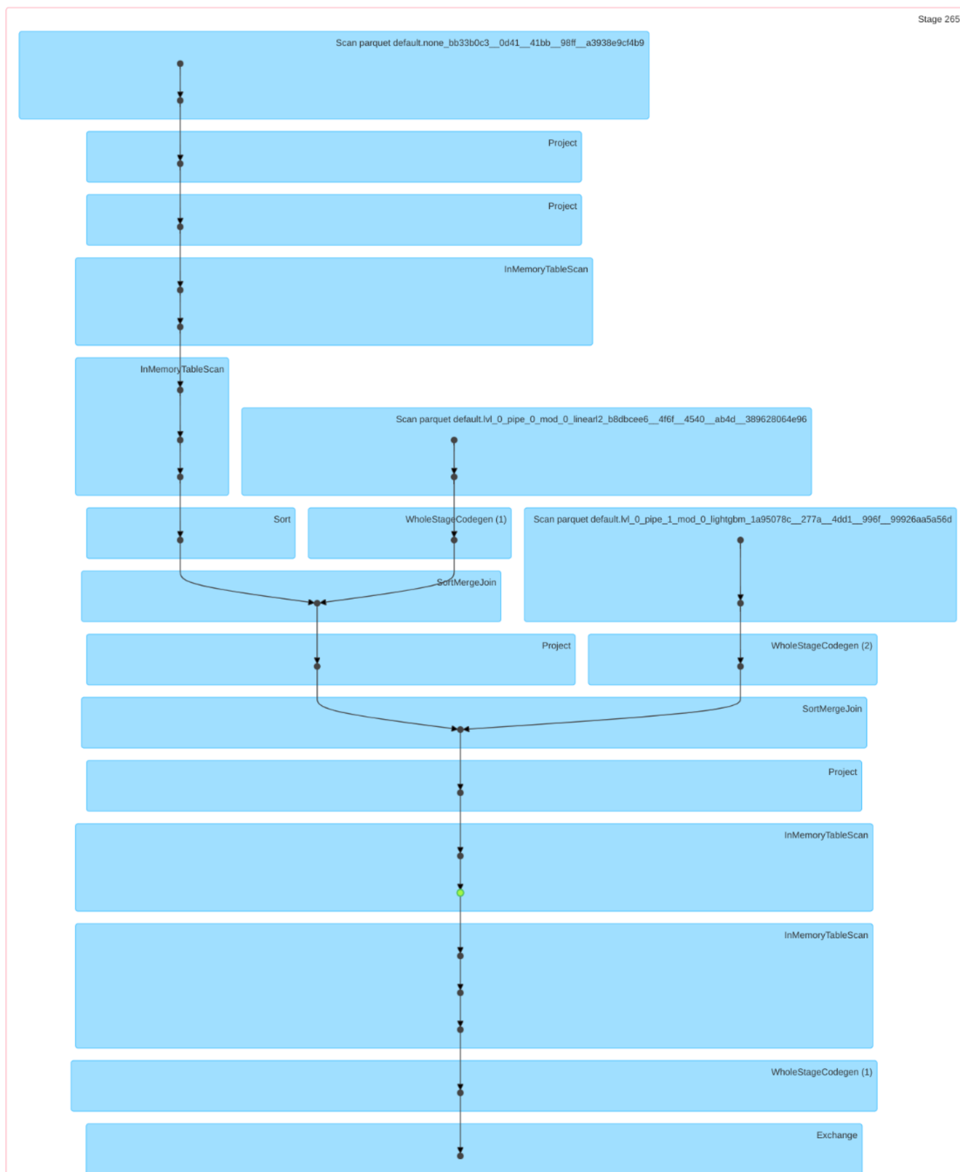
- Checkpoint
  – надежно, может быть долго, некоторая важная для оптимизатора информация о партицировании и сортировке может быть потеряна

```
ds = dataset.checkpoint(eager=True)
```

- Local Checkpoint
  – НЕ надежно, но быстро, некоторая важная для оптимизатора информация о партицировании и сортировке может быть потеряна

```
ds = dataset.localCheckpoint(eager=True)
```

# Ограничения Spark: бакетирование

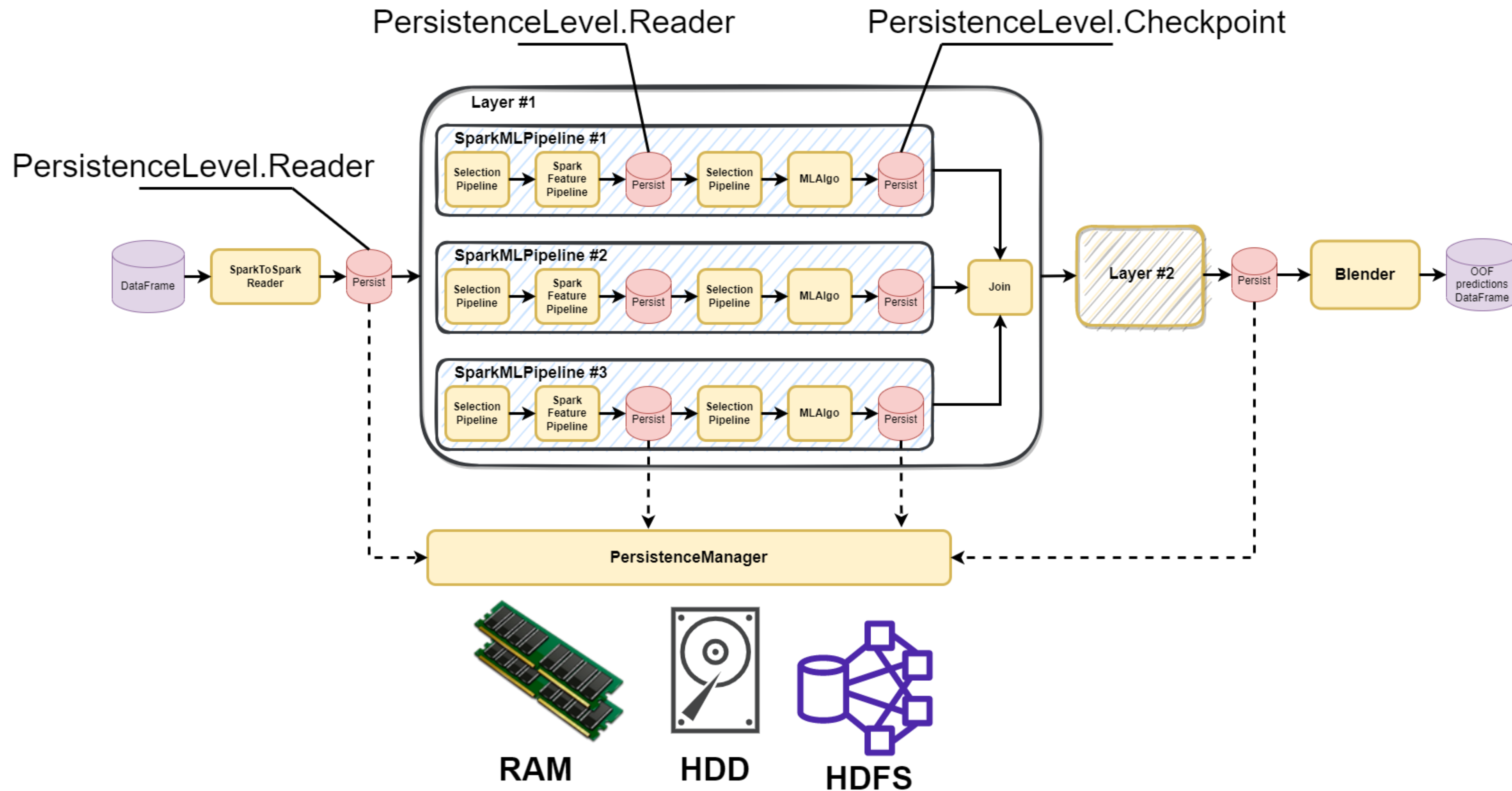



- Аналогично checkpoint в механизме сохранения
- Записывает датафрейм во внешнее хранилище
- Надежно к отказам
- Позволяет оптимизировать операцию Join, избавившись от shuffle.

Реализуется как отдельный persistence manager.

# Persistence Manager

- Сохранение датасета происходит в разных точках пресета, отличающихся по своей важности и поэтому обозначенными своим уровнем сохранения (PersistenceLevel): Reader, Regular, Checkpoint



# Persistence Manager: разновидности (2/2)

- Сохранение датасета происходит в разных точках пресета, отличающихся по своей важности и поэтому обозначенными своим уровнем сохранения (PersistenceLevel): Reader, Regular, Checkpoint
- Доступны 3 базовых реализации PersistenceManager: PlainCache, LocalCheckpoint, Bucketed
- Доступна реализация комбинатор CompositePersistenceManager. Позволяет скомбинировать разные менеджеры сохранения для применения в точках с разным уровнем.
- Доступны предопределенные комбинаторы CompositePlainCachePersistenceManager и CompositeBucketedPersistenceManager
- CompositePlainCachePersistenceManager: Reader – Bucketed, остальное - PlainCache
- CompositeBucketedPersistenceManage: Reader, Checkpoint – Bucketed; Regular - PlainCache

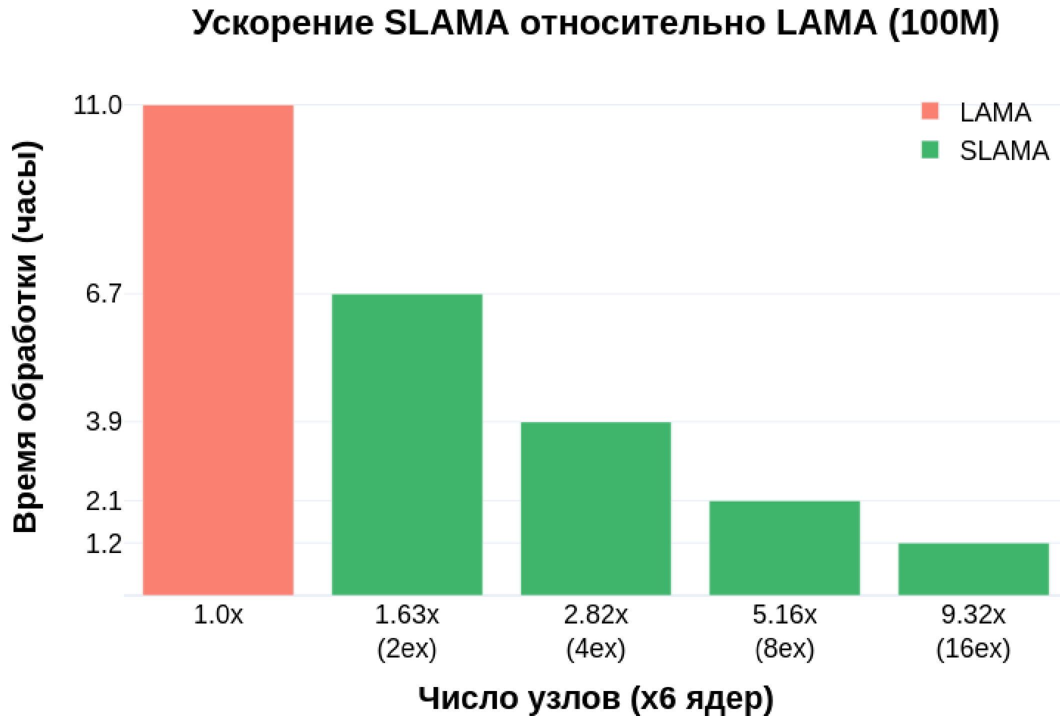
# Persistence Manager: что выбрать?

Выбор зависит от условий применения и датасета:

- Простой пресет с одним слоем, состоящей из одной модели - PlainCachePersistenceManager
- В пресете только один слой? Тогда можно не бакетировать датасет после ридера, а только закешировать (в фолдах потребуются union-ы) – подойдет CompositePersistenceManager с PlainCache для Reader и Regular и Bucketed для Checkpoint
- В пресете два слоя, но нет Skip? Тогда тоже подойдет CompositePersistenceManager с PlainCache для Reader и Regular и Bucketed для Checkpoint
- Датасет большой, но узкий? Тогда подойдет CompositePlainCachePersistenceManager
- В остальных случаях или не знаешь, что выбрать, универсальное решение - CompositeBucketedPersistenceManager

# Особенности масштабирования на датасетах среднего размнера

# Ограничение масштабируемости



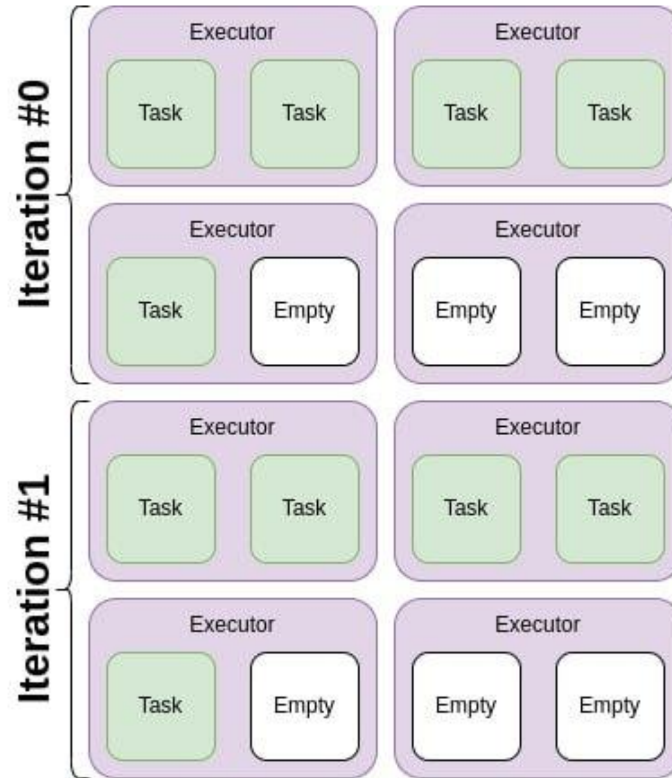

- Изначальный вариант SLAMA опирается на классический data-parallel подход.
- Но при избытке ресурсов и не достаточно большом размере датасета не получается эффективно использовать эти ресурсы.
- Алгоритмы ML обычно не масштабируются линейно с ростом данных, эффективность снижается с ростом числа доступных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

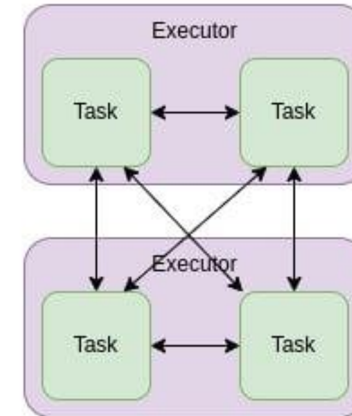
За счет горизонтального масштабирования на больших датасетах SLAMA работает быстрее LAMA, но на средних датасетах эффективность масштабирования быстро снижается.

ИTMO

# 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.

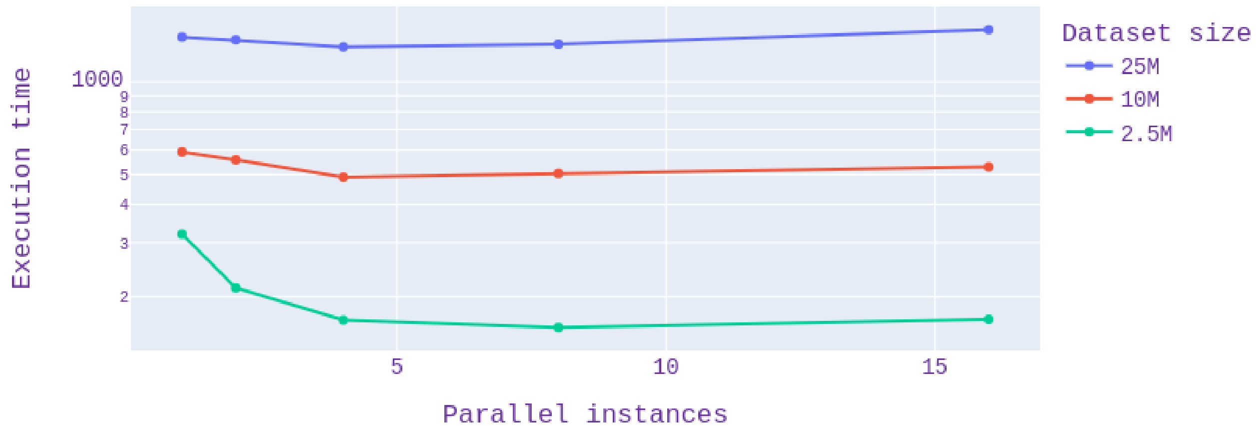
Алгоритм с MPI-подобным обменом данны

ИТМО

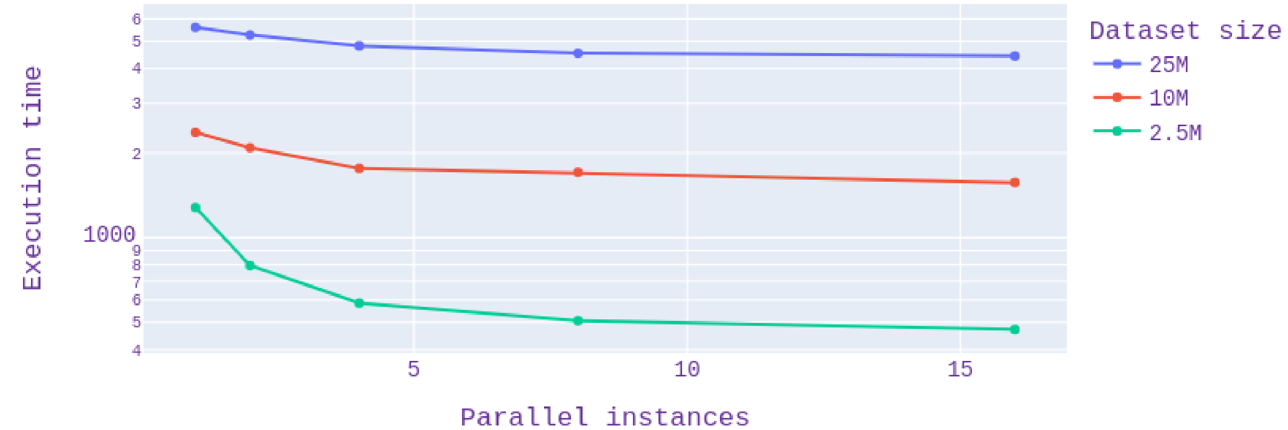
В случае с MPI-like может быть: чем больше размер “мира”, тем больше оверхедов на синхронизации.

# Гибридный data/compute parallel режим

16 инстансов




64 инстанса



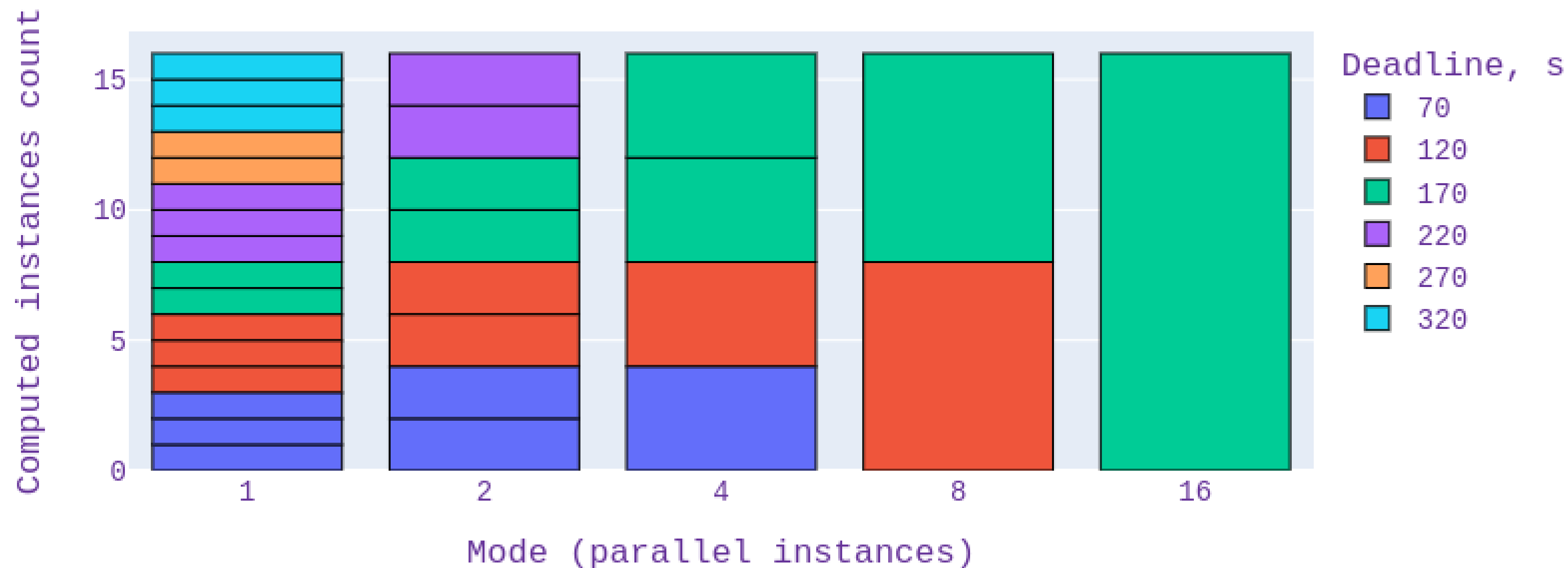

Зачем нам вообще гибридный режим, если compute-parallel режим позволяет рассчитать максимально быстро весь набор вычислений?

- Можем не влезть в память на одной машине, если весь датасет переедет туда.
- Можем не уложиться в дедлайн: достаточное количество расчетов, выполненное до дедлайна лучше, чем все расчеты, но законченные уже после дедлайна
- Не достаточное количество расчетов, чтобы загрузить все выделяемые ресурсы при одновременно хорошем коэффициенте масштабирования на небольшое число экзекьютеров

ITMO

# Гибридный data/compute parallel режим: 2.5 M датасет

Можем не уложиться в дедлайн: достаточное количество расчетов, выполненное до дедлайна лучше, чем все расчеты, но законченные уже после дедлайна

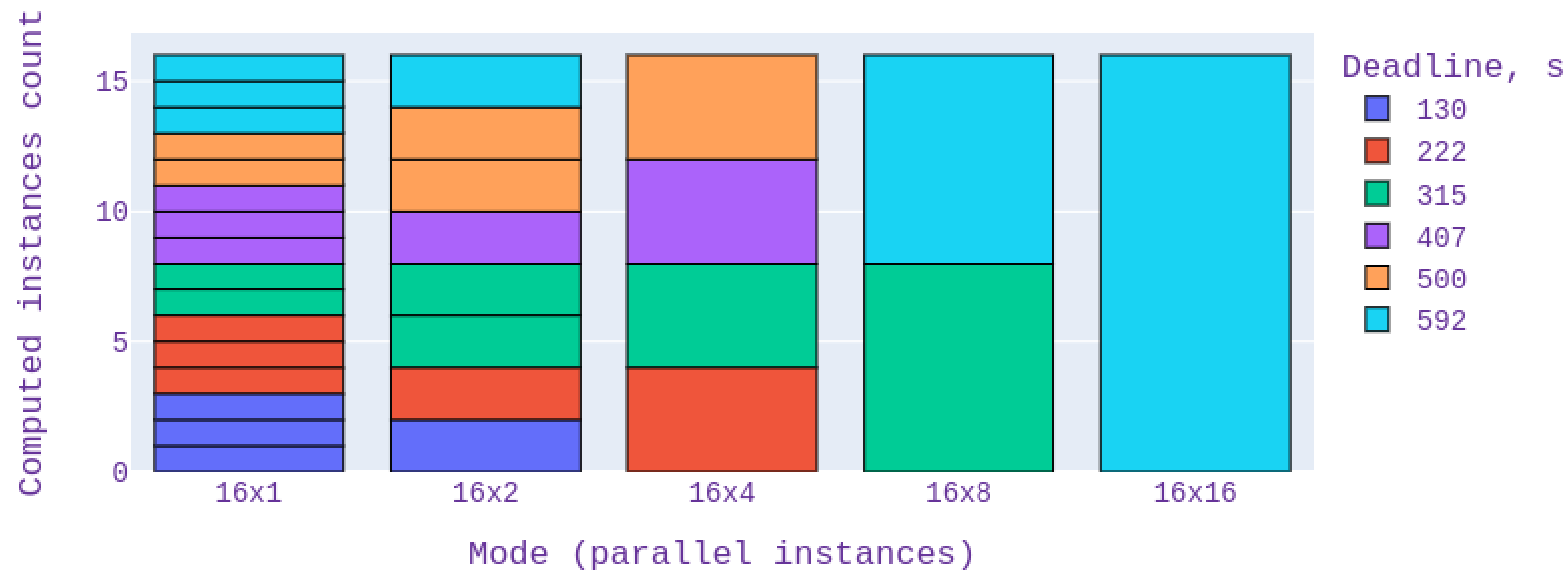



В зависимости от доступного времени: сначала наиболее эффективен data-parallel режим, потом гибридный, а потом гибридный или compute-parallel.

ИТМО

# Гибридный data/compute parallel режим: 10M датасет

Можем не уложиться в дедлайн: достаточное количество расчетов, выполненное до дедлайна лучше, чем все расчеты, но законченные уже после дедлайна

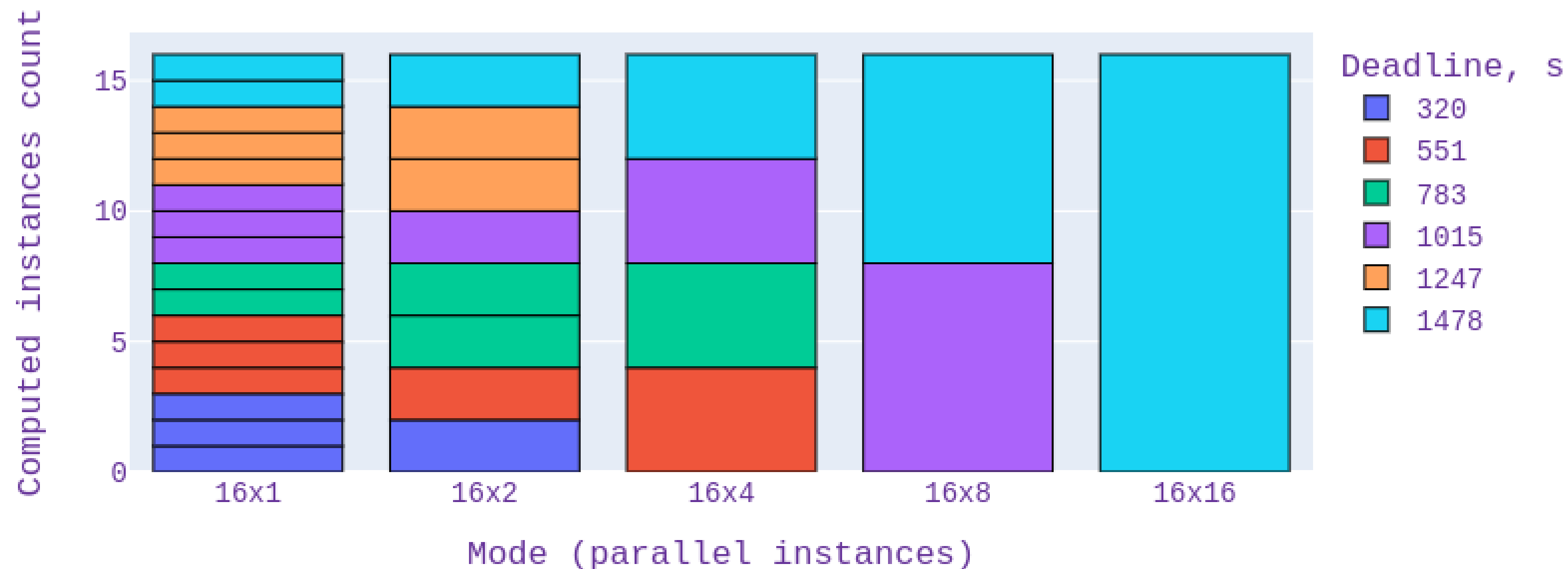



В зависимости от доступного времени: сначала наиболее эффективнее гибридный (2), потом data-parallel, а потом гибридный (4), .

ITMO

# Гибридный data/compute parallel режим: 25M датасет

Можем не уложиться в дедлайн: достаточное количество расчетов, выполненное до дедлайна лучше, чем все расчеты, но законченные уже после дедлайна

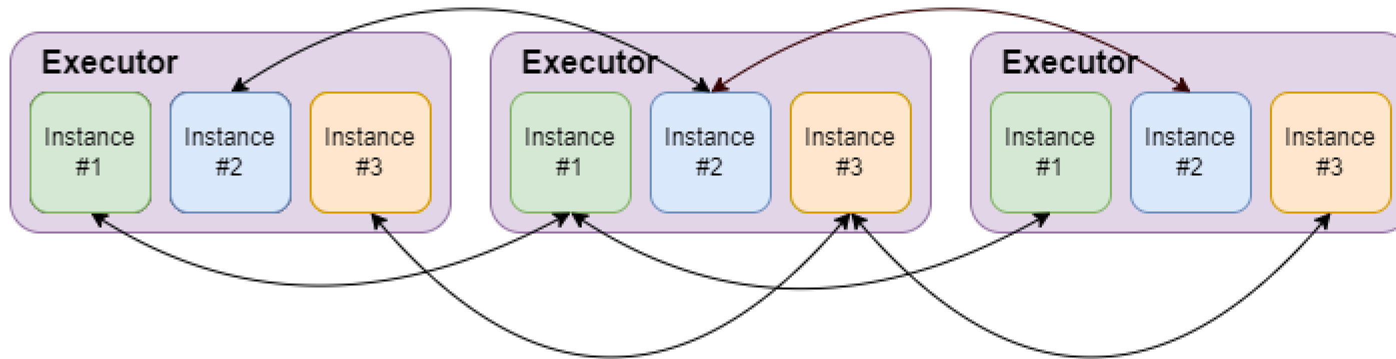



В зависимости от доступного времени: почти все время наиболее эффективен data-parallel, а для дедлайна 1015 секунд – гибридный (4) .

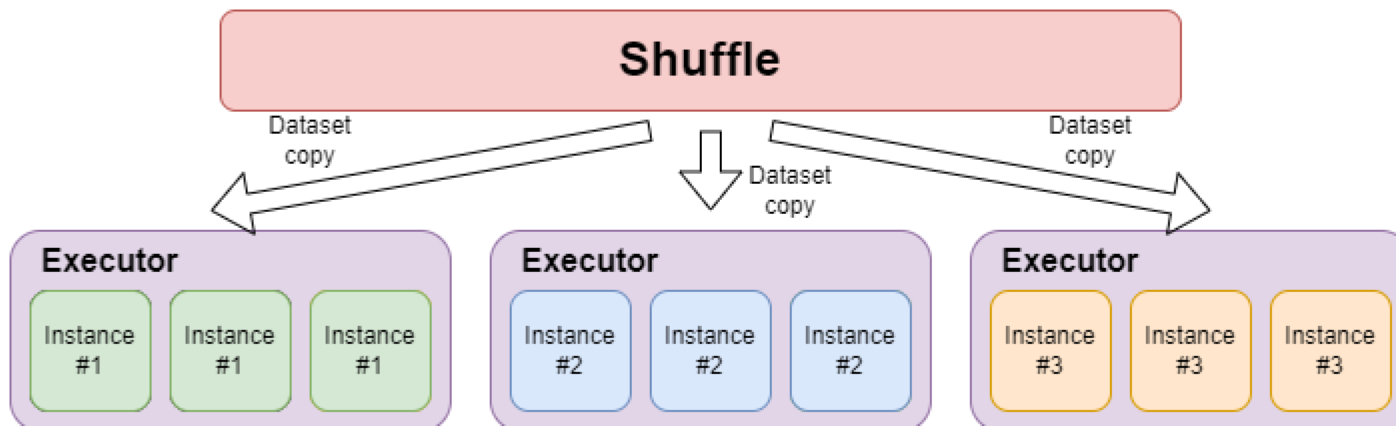
ИТМО

# Гибридный data/compute parallel для MPI-like

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




Простое уменьшение числа тасок.

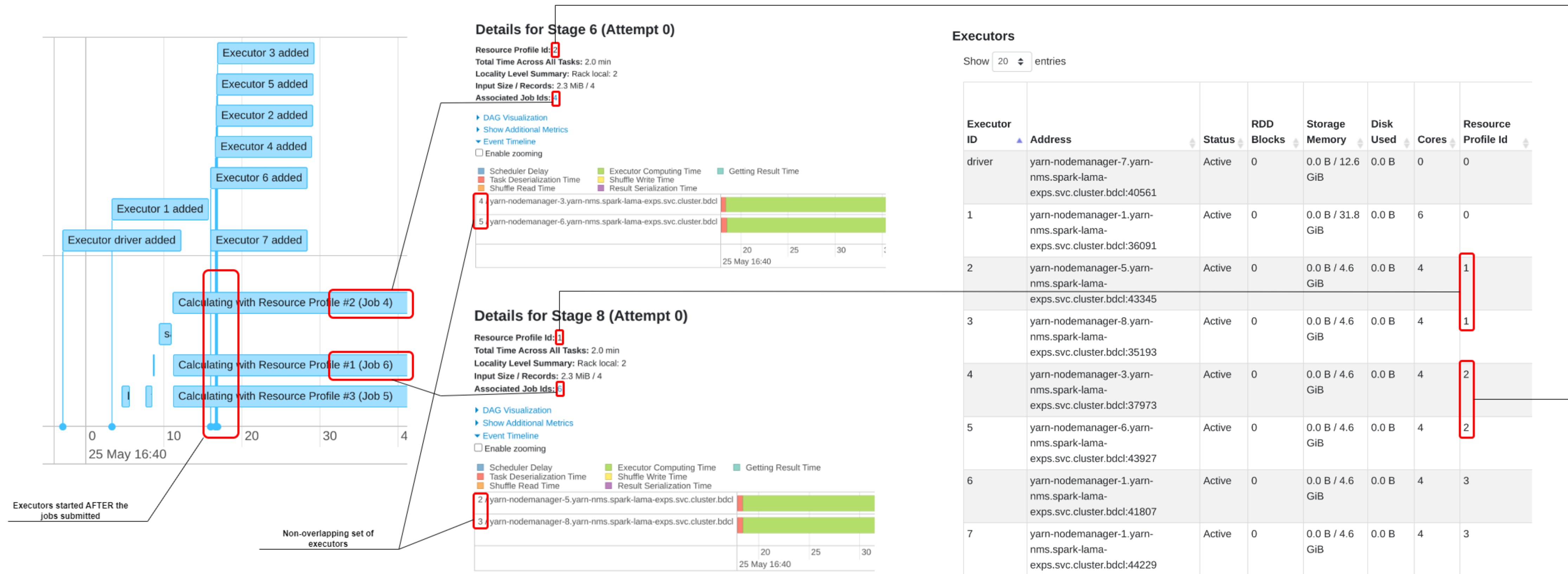



“Умный” coalescing.

ИTMO

# Гибридный data/compute parallel: Stage-Level Scheduling (WIP)

Stage-level scheduling (Spark 3.1.1+) позволяет выделять отдельные наборы экзекьютеров в режиме dynamic allocation “на лету”.



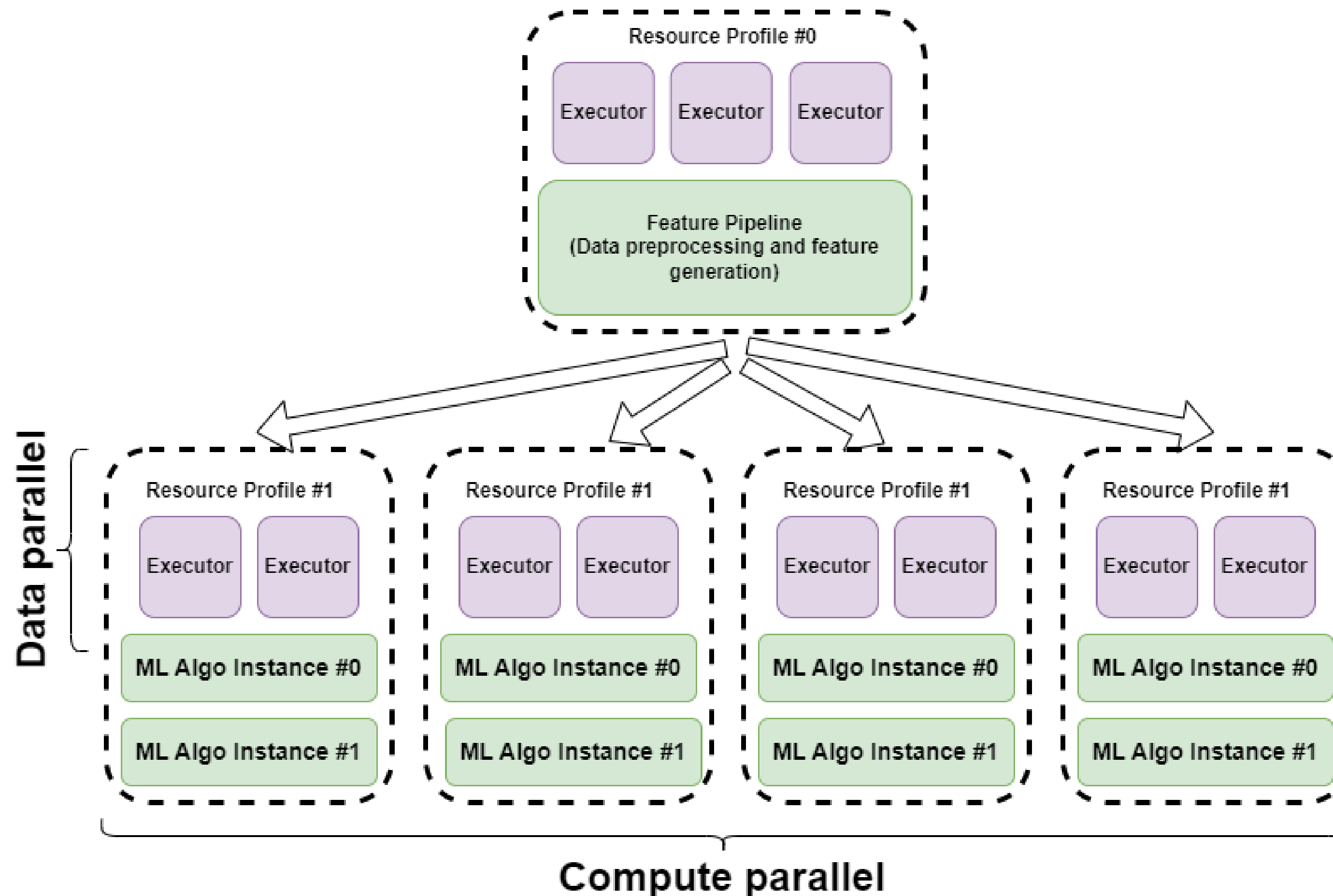

| Executor ID | Address | Status | RDD Blocks | Storage Memory | Disk Used | Cores | Resource Profile Id |
|---|---|---|---|---|---|---|---|
| driver | yarn-nodemanager-7.yarn-nms.spark-lama-exps.svc.cluster.bdcl:40561 | Active | 0 | 0.0 B / 12.6 GiB | 0.0 B | 0 | 0 |
| 1 | yarn-nodemanager-1.yarn-nms.spark-lama-exps.svc.cluster.bdcl:36091 | Active | 0 | 0.0 B / 31.8 GiB | 0.0 B | 6 | 0 |
| 2 | yarn-nodemanager-5.yarn-nms.spark-lama-exps.svc.cluster.bdcl:43345 | Active | 0 | 0.0 B / 4.6 GiB | 0.0 B | 4 | 1 |
| 3 | yarn-nodemanager-8.yarn-nms.spark-lama-exps.svc.cluster.bdcl:35193 | Active | 0 | 0.0 B / 4.6 GiB | 0.0 B | 4 | 1 |
| 4 | yarn-nodemanager-3.yarn-nms.spark-lama-exps.svc.cluster.bdcl:37973 | Active | 0 | 0.0 B / 4.6 GiB | 0.0 B | 4 | 2 |
| 5 | yarn-nodemanager-6.yarn-nms.spark-lama-exps.svc.cluster.bdcl:43927 | Active | 0 | 0.0 B / 4.6 GiB | 0.0 B | 4 | 2 |
| 6 | yarn-nodemanager-1.yarn-nms.spark-lama-exps.svc.cluster.bdcl:41807 | Active | 0 | 0.0 B / 4.6 GiB | 0.0 B | 4 | 3 |
| 7 | yarn-nodemanager-1.yarn-nms.spark-lama-exps.svc.cluster.bdcl:44229 | Active | 0 | 0.0 B / 4.6 GiB | 0.0 B | 4 | 3 |

Отдельный тред может рассчитать последовательность алгоритмов на своем собственном наборе экзекьютеров в желаемой конфигурации data/compute parallel.

ITMO

# Гибридный data/compute parallel: Stage-Level Scheduling (WIP)

Stage-level scheduling (Spark 3.1.1+) позволяет выделять отдельные наборы экзекьютеров в режиме dynamic allocation “на лету”.




ITMO

Отдельный тред может рассчитать последовательность алгоритмов на своем собственном наборе экзекьютеров в желаемой конфигурации data/compute parallel.

# Выводы

# Ссылка на репозиторий Slama

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

# Выводы

- Иногда нужно спускаться до уровня Scala можно и нужно, чтобы написать эффективный код для PySpark. Классы типа Aggregator доступны только там.
- Кэширование и сохранение промежуточных данных важный процесс, которым нужно управлять, включая что и когда должно вытесняться из кэша и в каких моментах стоит обрезать lineage.
- Масштабирование крупных датасетов прекрасно реализуется на основе классического data-parallel подхода, однако если датасет “средний”, то уже нужны другие методы и подходы.
- Гибридный data/compute parallel может показать существенно большую эффективность при большом количестве вычислительных ресурсов и ограниченнорм размере датасета, однако выбор оптимальной степени параллельности требует определенной сноровки.
- Развитие среств планирования и управления вычислениями в Spark создает новые возможности для повышения эффективности, в частности для эксплуатации гибридных режимов вычислений с наиболее эффективными конфигурациями экзекьютеров за счет сепарированного выделения экзекьютеров по отдельным профилям.

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

SBER AI LAB

Лаборатория Искусственного Интеллекта

ITMO

Центр “Сильный ИИ в промышленности”