

# Немного советов и рассказ про GPU

# О чем поговорим

- Как упростить работу
- Неочевидные ошибки
- Кейс про транзакции
- Кейс про БКИ
- Кейс про данные Collection
- Spark on GPU

# Как упростить работу

- Все начинается с конфига и многое зависит от него
  - `Spark.python.worker.reuse` конфиг интересный, но если у вас много операций, которые затратны по памяти, то высок риск, что java заругается и упадет;
  - Если при считывании таблицы, у вас все падает по памяти, то есть конфиг, который использует `hive serde` для паркетов вместо встроенных алгоритмов. `spark.sql.hive.convertMetastoreParquet` нужно задать с параметром `false`. Будет больше job, дольше работать, но зато есть шанс посчитать.
  - Мы уже знающие и помним, что `ruspark` 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 и то, что динамика может забрать все ресурсы.
  - Помните, что если задаете статические параметры ресурсов, то на них динамическая аллокация не распространяется.

# Как упростить работу

- Партицирование
  - Смотрите на таблицы и попадайте в партиции! Фуллскан огромных таблиц занимает много времени, забивает память и в итоге ваши скрипты падают с ошибками.
  - Если сохраняете свои паркететы, то их тоже можно и нужно партицировать, когда данных много.
- Правильно закрывайте контекст:
  - `ss.catalog.clearCache()` – чистим cache, полезно делать в целом при работе, если тянете много данных;
  - `ss.stop()` – выключает контекст и все чистит, это лучше, чем `kernel - restart`.

# Как упростить работу

- Следите на shuffle
  - В большинстве случаев Stages предполагают shuffle при переходе от одного к другому, подписывая об этом информацию.
  - Помните про то, какие операции приводят к полному shuffle, а какие делают это менее агрессивно.

← ↻ ↺ pklis-cdh000109.lbiac.df.sbrf.ru:8090 Trans - Stages for All Jobs

Spark 2.4.0.cloudera2 Jobs Stages Storage Environment Executors SQL Trans application UI

### Stages for All Jobs

Active Stages: 2  
Pending Stages: 1  
Completed Stages: 6

▼ Active Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
7	save at NativeMethodAccessorImpl.java:0	2021/08/18 17:33:32	12.0 h	5672/280654	194.7 GB			2.8 GB
6	save at NativeMethodAccessorImpl.java:0	2021/08/19 12:26:57	1.7 h	104/200 (16 running)			151.0 GB	8.1 GB

▼ Pending Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
8	save at NativeMethodAccessorImpl.java:0	Unknown	Unknown	0/200				

▼ Completed Stages (6)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
5	save at NativeMethodAccessorImpl.java:0	2021/08/19 02:09:39	10.3 h	200/200			3.8 TB	254.0 GB
4	save at NativeMethodAccessorImpl.java:0	2021/08/18 17:33:29	8.6 h	117116/117116 (96 failed)	3.1 TB			3.7 TB
3	save at NativeMethodAccessorImpl.java:0	2021/08/18 17:33:25	2.0 min	32/32	2.1 GB			2.9 GB
2	Listing leaf files and directories for 5546 paths: hdfs://clsklrisk/data/custom/risk/agrret/pa/t_transactions/ctl_src_id=3000000/trans_date_part=0109-02-13, ... save at NativeMethodAccessorImpl.java:0	2021/08/18 17:31:01	23 s	5546/5546				
1	showString at NativeMethodAccessorImpl.java:0	2021/08/18 17:15:52	4 s	1/1	127.5 MB			
0	parquet at NativeMethodAccessorImpl.java:0	2021/08/18 17:10:23	8 s	1/1				

# Неочевидные ошибки

Они бывают у всех, но очень легко ошибиться тем, кто захочет писать sql-скрипты и сильно привык к oracle. Многие работают 1 в 1, но все же стоит смотреть документацию <https://spark.apache.org/docs/latest/api/sql/> и быть очень внимательными при работе с датами:

- `to_date` работает криво, если писать дату начиная с дня месяца, надежнее `to_date('2016-12-31', 'yyyy-MM-dd')` или просто `date('2016-12-31')`;
- `date_trunc` работает без проблем только для дат в формате string, либо datetime value;
- `trunc` кушает даты, но округляет только до года, квартала, месяца и недели;
- `date_add` добавляет только дни.

# Кейс про транзакции

Немного про витрину:

- БВ Транзакции и карты ФЛ;
- Весит порядка 15-20 Тб;
- Партицирована по дате (`trans_date_part`) и по источнику (`ctl_src_id`);
- Для каждого человека нужно собрать N его последних транзакций с метаданными.

Какие сложности:

- Большой объем, full scan всей витрины убьет любой наш кластер и так делать не нужно;
- Нужно собрать N транзакций и в последовательность.

# Кейс про транзакции – общий подход

Хочется собрать N последних одним кодом, но мы нарвемся на full scan всей витрины. Почему? Потому что мы не можем сказать, что все N транзакций мы найдем только за один или два года и ограничить витрину по партициям.

Как решаем:

- Разбиваем задачу на подзадачи по годам: собираем N последних транзакций внутри каждого года;
- Объединяем данные, сортируя по клиенту транзакции и отбирая N самых свежих.

**Важно:**

Это очень тяжелый источник, забивающий диски и YARN, после сохранения данных за какой-то год рекомендуется выполнять команду для очистки cache `ss.catalog.clearCache()`.



# Кейс про транзакции – попадаем в партии

```
query = """
select
t.*
from
(
select
t.*,
row_number() over(partition by t.application_id order by t.trans_date desc) as nm
from
(select distinct
t.application_id,
t.application_date,
0 as client_id,
0 as date_open,

d.trans_date,
d.trans_type,
d.mcc_code,
d.trans_country,
d.trans_currency_cd,
d.amount_rur,
date_trunc(d.trans_date, 'DD') as report_date,
d.trans_city

from way4_pprb_id t
join risk_custom_risk_agrret.t_transactions d
on d.client_id = t.client_id
where date(d.trans_date_part) between date('2022-01-01') and date('2022-03-31') and
d.trans_date < t.application_date and d.ctl_src_id in (3000000)
) t
) t
where nm <= 800
"""
tranz_2022 = ss.sql(query)
```

Стоит обратить внимание , что условие `trans_date < application_date` в целом вам никогда не отсекает партии. Spark не посмотрит тут на всю выборку и не поймет, что максимальный `application_date` можно вычислить и ограничить партии

Партии ограничиваем в явном виде через `where` или `filter` в DataFrame API

# Кейс про транзакции – собираем данные

Считываем данные за все года, объединяем, берем свежие N.

Далее нужно собрать структуру данных, чтобы от связи клиент – транзакция перейти в связи клиент – последовательность всех его транзакций.

```
root
|-- application_id: decimal(38,10) (nullable = true)
|-- application_date: timestamp (nullable = true)
|-- way4_client_id: string (nullable = true)
|-- client_id: string (nullable = true)
|-- date_open: timestamp (nullable = true)
|-- trans_date: timestamp (nullable = true)
|-- trans_type: decimal(10,0) (nullable = true)
|-- mcc_code: decimal(10,0) (nullable = true)
|-- trans_country: string (nullable = true)
|-- trans_currency_cd: decimal(10,0) (nullable = true)
|-- amount_rur: decimal(38,3) (nullable = true)
|-- report_date: timestamp (nullable = true)
|-- trans_city: string (nullable = true)
|-- nm: integer (nullable = true)
```



```
root
|-- application_id: decimal(38,10) (nullable = true)
|-- application_date: timestamp (nullable = true)
|-- card_tranz: array (nullable = false)
|   |-- element: struct (containsNull = false)
|   |   |-- way4_client_id: string (nullable = true)
|   |   |-- mcc_code: decimal(10,0) (nullable = true)
|   |   |-- trans_date: timestamp (nullable = true)
|   |   |-- trans_type: decimal(10,0) (nullable = true)
|   |   |-- trans_country: string (nullable = true)
|   |   |-- trans_currency_cd: decimal(10,0) (nullable = true)
|   |   |-- amount_rur: decimal(38,3) (nullable = true)
```

# Кейс про транзакции – собираем данные

```
ds_tranz = data_original.select('application_id', 'application_date',  
                                F.struct('way4_client_id', 'mcc_code', 'trans_date',  
                                          'trans_type', 'trans_country',  
                                          'trans_currency_cd', 'amount_rur').alias('card_tranz')).drop_duplicates()
```

```
loaner_ds = ds_tranz.groupby('application_id', 'application_date').agg(  
    F.collect_list('card_tranz').alias('card_tranz')  
).drop_duplicates()
```

Далее это счастье можно обрабатывать через `.rdd.map` при помощи python кода, например.

# Кейс про БКИ

Немного про витрины:

- Много различных таблиц, которые делятся по блокам и бюро;
- Самые тяжелые витрины – хранящие информацию по каждому кредиту клиента во времени;
- Различные форматы хранения;
- Партицированы по `part_date`.

Какие сложности:

- Множество таблиц;
- Сложные структуры;
- Нетривиальная обработка.

Что нужно:

- Для каждого клиента собрать 1D, 2D и 2D вектора.

# Кейс про БКИ – парсинг строк

В эквифаксе информация о просрочках хранится в виде последовательности символов, где каждый отвечает за свой бакет просрочки и месяц. Если кредит закрыт, то все равно тянутся статусы о закрытии. Нам нужно перевести это в вектор, развернуть (в примере дальше мы в итоге развернем в исходный порядок для целей дальнейшей обработки ) и проверить длину по сроку закрытия.

`discipline` | `cccccccccccccc`  `[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, C]`

Можно было бы это когда-то потом парсить через python, но у другого бюро данные хранятся по аналогии с транзакциями, поэтому хочется все сделать сразу в pyspark + данных много. Как делаем? UDF!

А еще хочется понять, какой у нас самый актуальный статус, то есть забрать первый элемент после препроцессинга.

# Кейс про БКИ – парсинг строк

```
@F.udf(returnType=StringType())
def preprocess_discipline(duration, sausage):
    duration = max(0, int(duration))
    sausage = sausage[:: -1][:duration + 1]
    sausage = sausage[:: -1]
    sausage_new = []
    k = 0
    for token in sausage:
        if token == 'C':
            k += 1
        if k == 2:
            break
        sausage_new.append(token)
    return ''.join(sausage_new)
```

Пишем udf, говорим, что хотим вернуть StringType()

✓ Всему столбцу присваиваем 1

## Применяем udf к нескольким столбцам /

```
eq_df = d2_eq_tmp.filter('response_id_eq is not null and cred_id is not null')\
    .withColumn('eq_flag', F.lit(1))\
    .withColumn('cred_mob', F.months_between(F.col('application_date'), F.col('cred_date')))\
    .withColumn('cred_duration', F.months_between(F.col('cred_enddate'), F.col('cred_date')))\
    .withColumn('3d_eq', F.split(preprocess_discipline(F.col('cred_duration'), F.col('discipline')), ''))\
    .withColumn('cred_day_overdue', F.col('3d_eq').getItem(0))\
    .withColumn('cred_day_overdue', F.col('cred_day_overdue').cast('int'))
```


Вытягиваем 1 элемент

# Кейс про БКИ – оконные функции

Есть таблица, нужно в ней найти максимум, последний элемент. Тут помогут оконные функции.

```
from pyspark.sql.window import Window
```

```
d3_okb_tmp = d3_okb_tmp.select('response_id_okb', 'account_id', F.max('arrearbalance')\
                                .over(Window.partitionBy('response_id_okb', 'account_id'))\
                                .alias('cred_max_overdue'),\
                                F.first('accountbalance', True).over(Window.partitionBy('response_id_okb', 'account_id')\
                                .orderBy(d3_okb_tmp.historydate.desc()))\
                                .alias('accountbalance'),\
                                F.struct('historydate', 'accountpaymenthistory', 'instalment', 'arrearbalance').alias('3d_okb'))
```



Тут можно заменить на  
F.col('historydate').desc(),  
особенно если это новый или  
преобразованный столбец

# Кейс про БКИ – структуры

Как и в задаче с транзакциями очень часто используется конструкция `groupby – agg` для сбора последовательностей.

```
caps_okb = caps_okb.withColumn('2d_okb_caps', F.struct('record_id', 'enquirydate', 'amountoffinance', 'applicationchannel',  
                                                    'currency', 'enquirydate_diff', 'application_flag', 'financetype',  
                                                    'paymentfrequency', 'reason', 'is_sbrf_caps'))\  
                .select('application_number', 'application_date', 'response_id_okb', '2d_okb_caps')
```

```
eq_df_grouped = eq_df.groupby('application_number', 'application_date', 'response_id_eq')\  
                .agg(F.collect_list('2d_eq').alias('2d_eq'),  
                    F.collect_list('3d_eq').alias('3d_eq'))  
  
okb_final_grouped = okb_final.groupby('application_number', 'application_date', 'response_id_okb')\  
                .agg(F.collect_list('2d_okb').alias('2d_okb'),  
                    F.collect_list('3d_okb').alias('3d_okb'))  
  
caps_okb_grouped = caps_okb.groupby('application_number', 'application_date', 'response_id_okb')\  
                .agg(F.collect_list('2d_okb_caps').alias('2d_okb_caps'))
```

А далее через `.rdd.map` применяем `python` код для сборки `dict` в формате, который удобен для нейронки. Ну и главное все аккуратно собрать, не забывать про партиции.



# Кейс про данные Collection

В данном случае мы работаем как с большим числом агрегатов из различных источников, так и с большим количеством тяжелых последовательностей.

Некоторые советы:

- Не стоит пытаться за 1 раз (в одном графе все сразу собрать). Лучше делать частями, последовательно добавляя по источнику.
- Всегда помните, что написав много кода, он не выполнится до action. Если вы захотите сделать show и посмотреть, то рассчитается весь граф, а когда увидите, что все ок и захотите сохранить, посчитается еще раз. Отсюда вывод – persist или сохраняем сразу в hdfs.
- При большом числе источников часто дублируются имена столбцов при join, делайте drop лишнего.
- Очень помогают оконные функции в сочетании в F.first(), F.mean(), F.sum(), F.count(), F.row\_number() + groupBy().agg().

```
rb_agg = rb_agg.join(dep_agrmnt_bal_wxn_mnth_3yr, on=[rb_agg.epk_id==dep_agrmnt_bal_wxn_mnth_3yr.epk_id,
                                                    rb_agg.report_date_agg==dep_agrmnt_bal_wxn_mnth_3yr.report_date],
                    how='left')\
    .drop(dep_agrmnt_bal_wxn_mnth_3yr.epk_id)\
    .drop(dep_agrmnt_bal_wxn_mnth_3yr.report_date)
```

# Кейс про данные Collection

- Иногда нужно взять запись и размножить ее на несколько (например, n дней назад):

```
waiting = waiting.withColumn('min_report_date', F.date_sub(F.col('report_date'), 90 + 2))\
    .withColumn('max_report_date', F.date_sub(F.col('report_date'), 2))\
    .withColumn('report_date_comm', F.explode(F.expr('sequence(min_report_date, max_report_date, interval 1 day)'))
```

- Не бойтесь в select писать выражения вместо перечислений, главное, чтобы ruyspark понял:

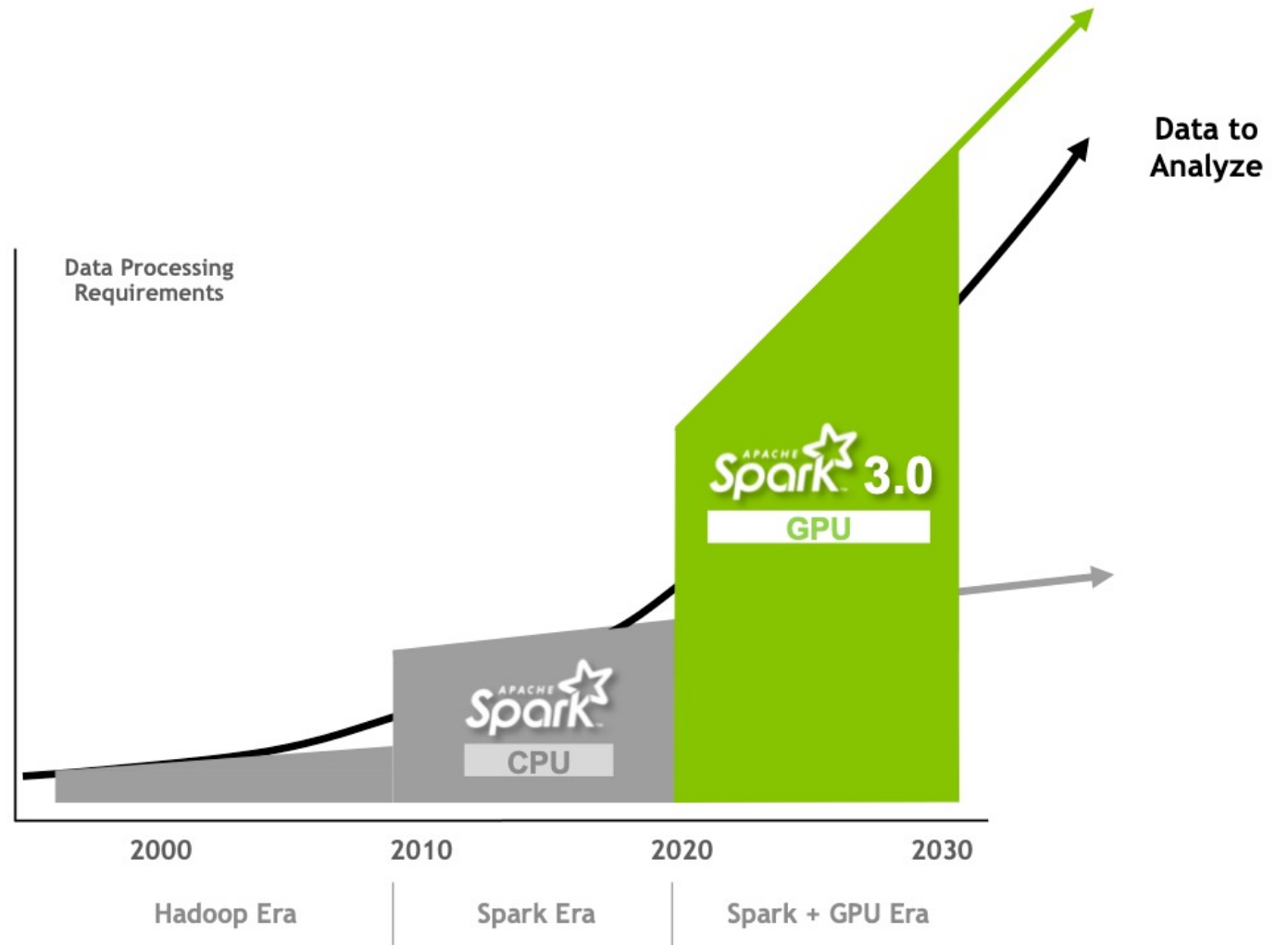
```
dep_agrmnt_bal_wxn_mnth_3yr = dep_agrmnt_bal_wxn_mnth_3yr.select(*[F.col(s).alias(s.replace('sum(', '').replace(')', '')) if 'sum' in s else s for s in dep_agrmnt_bal_wxn_mnth_3yr.columns])
```

- Используйте все удобства ruyspark:

```
communications_with_waiting = communications_with_waiting.withColumnRenamed('comm_type_code', 'comm_type_code_hist')\
    .fillna({'comm_type_code_hist': 'WAITING_CMD', 'meeting_result_code': 0, 'problem_sign_code': 0,
            'agreement_code': 0, 'call_result_code': 0, 'visit_result_code': 0,
            'regulating_tech_code': 0, 'ovrd_reason_code': 0})
```

# Spark on GPU

- Все развивается, сначала был Hadoop, потом Spark. Начиная с версии 3.0 Spark неплохо начал общаться с GPU.

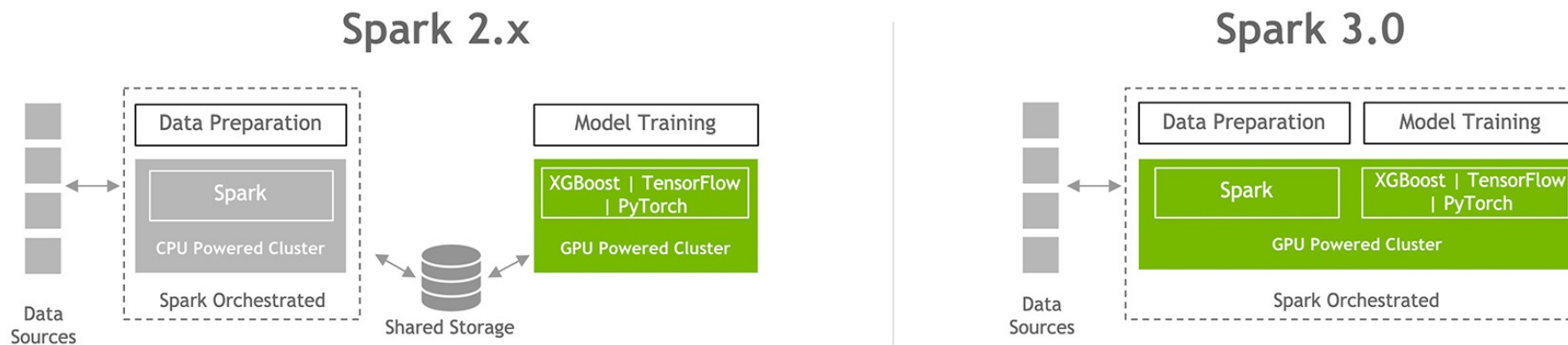


# Spark on GPU

Все новшества решают какие-то проблемы. Spark решил проблему I/O у Hadoop, добавив обработку в памяти, но все развивается и появляется новая проблема: приложений слишком много, вычислений также много, хочется считать все быстрее.

NVIDIA посидели с сообществом Apache Spark и выпустили Spark 3.0 вместе с RAPIDS Accelerator for Spark. Что получили:

- Ускорение времени обработки данных и обучения моделей;
- Единая инфраструктура может быть использована под Spark и остальные DL/ML фреймворки;
- Можно купить меньше серверов – снизим затраты на инфраструктуру (стоимость GPU не в счет).

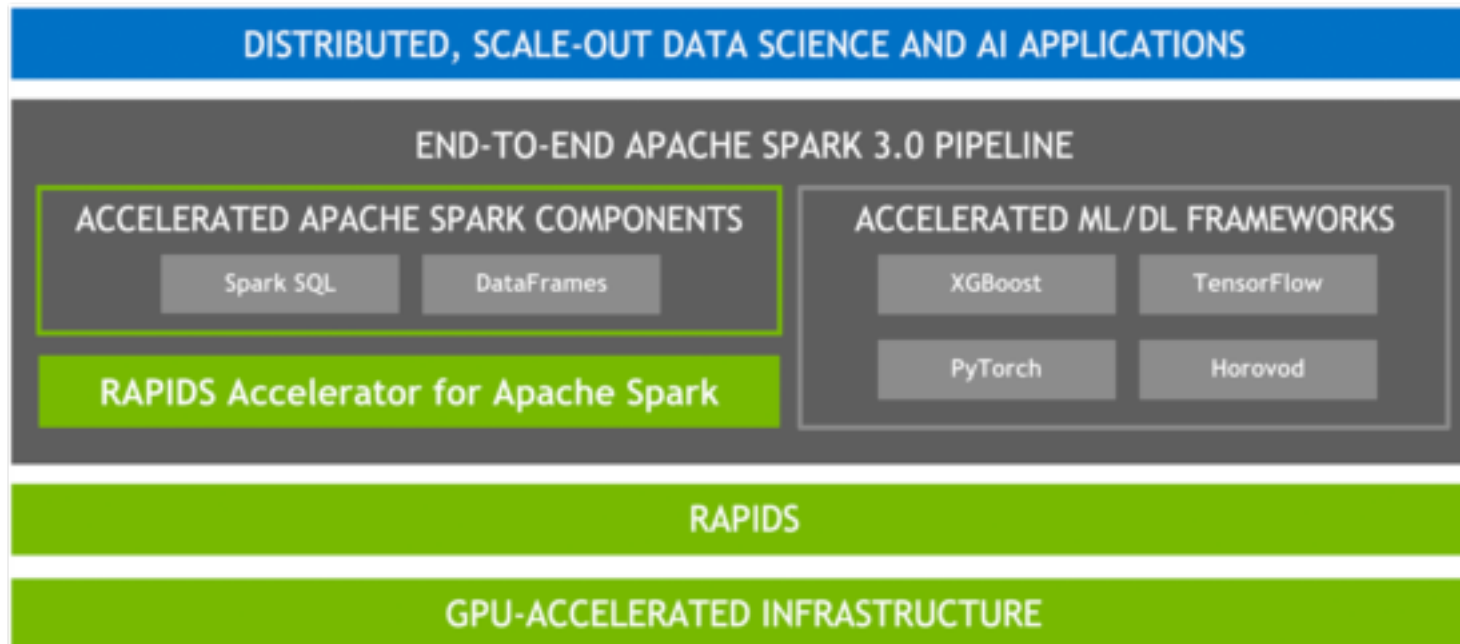


# RAPIDS Accelerator for Apache Spark

Про Rapids многие слышали и даже работали с ним.

Теперь Rapids подружился со Spark и у нас есть построенный поверх NVIDIA CUDA и UCX интересный RAPIDS Accelerator for Apache Spark, который позволяет выполнять SQL-операции и работать с DataFrame при помощи GPU без изменения кода + оптимизированы операции shuffle.

А дальше можно использовать на том же сервере наши любимые pytorch и прочее, строя нейроночки.



# RAPIDS Accelerator for Apache Spark

## SQL и DataFrame

Обработка идет не по строкам, а по столбцам, это более дружелюбная структура для GPU. Catalyst был изменен и при анализе задач, планов запросов он знает про RAPIDS, используя его возможности. То есть теперь мы анализируем задачу с точки зрения возможности применения GPU.

А что если памяти не хватит? Spark умный, где не хватит, перейдет на CPU.

## Shuffle

Операции Spark, которые сортируют, группируют или объединяют данные по значению, должны перемещать данные между нодами при создании нового DataFrame из существующего между этапами. Shuffle включает disk I/O, сериализацию данных и network I/O. Новая реализация RAPIDS accelerator shuffle использует UCX для оптимизации передачи данных на GPU, сохраняя как можно больше данных на GPU. Он находит самый быстрый путь для перемещения данных между узлами, используя лучшие из доступных аппаратных ресурсов, включая обход центрального процессора для передачи данных с графического процессора на графический процессор внутри и между узлами.

# RAPIDS Accelerator for Apache Spark

## Accelerator-aware scheduling

Планирование задачи с учетом наличия ускорителя в виде GPU.

В рамках крупной инициативы Spark по лучшей унификации DL и обработки данных в Spark графические процессоры теперь являются планируемым ресурсом в Apache Spark 3.0. Это позволяет Spark планировать исполнителей с заданным количеством графических процессоров, и можно указать, сколько графических процессоров требуется для каждой задачи. Spark передает эти запросы ресурсов базовому менеджеру кластера: Kubernetes, YARN или Standalone.

# Spark и numba

В курсе по GPU мы рассматриваем numba и разбираемся, как с ее помощью писать kernel functions на GPU. Хорошая новость для тех, кто хочет писать кастомные штуки, а не использовать то, что написали за вас – функции, которые разогнаны в numba, могут использовать в Spark.

Что нужно кроме знаний по numba, чтобы запустить функции на GPU:

- ~~Загуглить;~~
- Написать функцию через `@vectorize` или `@cuda.jit`;
- Написать функцию-обертку для вызова (определить сетку вычислений, то есть количество блоков и потоков на блок);
- Применить к RDD через `mapPartitions`, а если несколько GPU, то через `mapPartitionWithIndex`.