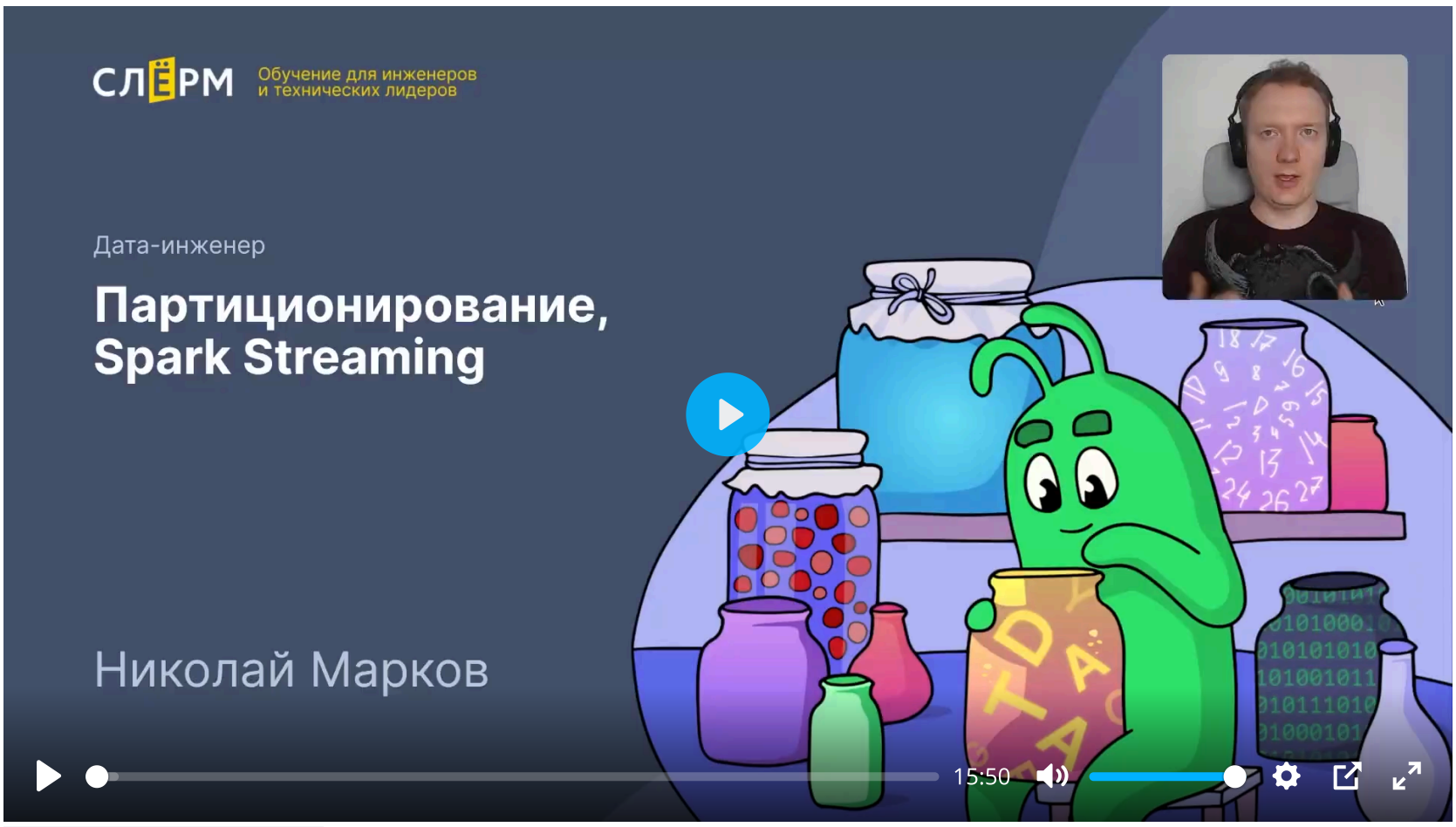
4.4 Партицирование, репартицирование Spark Streaming



Презентация к уроку 4.4

Текстовая расшифровка видео:

ВВЕДЕНИЕ

ПАРТИЦИОНИРОВАНИЕ, SPARK STREAMING

На этом уроке мы узнаем:

- Как посмотреть логический и физический план запроса;
- Как улучшить размещение данных на кластере;
- Можно ли сохранять промежуточные результаты вычислений;
- Как анализировать потоковые данные с помощью Spark.

ПЛАН ВЫПОЛНЕНИЯ

План:

- Вспомним о проектах;
- Вспомним основы;
- Многоступенчатый процесс;
- Кейс – исходные данные;
- Смотрим запрос;
- Физический план;
- Логический план;

Вспомним о проектах

Рассмотрим данные проекты:

Запустить стенд

Дедлайн 07 июля, 23:59 Мск

Catalyst Optimizer;

Whole-Stage Java Code Generation;

Все они существуют совместно со Spark и решают разные задачи, касаемые вычислений на нем.

Мы уже упоминали о них, в частности о Catalyst Optimizer. Именно Catalyst Optimizer решает, как именно будут запускаться вычисления на кластере. На основе кода для Spark, который мы пишем, он создает цепочку того, что и где должно запуститься, а также отдает дальнейшие приказы менеджеру кластера. Благодаря этому мы получаем возможность эффективно считать на кластере.

Вспомним основы

RDD и **Dataframe** – это классы-абстракции над **коллекциями** данных.

Коллекции разбиты на **блоки (partitions)**.

Когда мы пишем код, чтобы посчитать что-либо на этих данных, он раскладывается на драйверы в граф исполнения. Это то, что называется «Spark Job».

- **Job** – граф целиком, от момента создания DF до применения action к нему.
- **Stage** – компонент Job. Создается каждый раз при shuffle.
- **Task** – компонент Stage. Это базовая операция над данными.

Более подробную информацию о shuffle вы можете прочитать, перейдя по ссылке: https://sparkbyexamples.com/spark/spark-shuffle-partitions/

Многоступенчатый процесс

Когда мы закинули данные на кластер и запустили код, под «капотом» многое произошло:

- Оптимизатор получает от нас код и составляет цепочку планов. Сначала составляет базовые прикидки.
- В процессе тестирования гипотез план уточняется, оптимизируется.
- На основе логического плана оптимизатор составляет физический план (план, описывающий передвижение данных на определенных машинах и т.д.). При составлении физического плана используется cost based optimization. То есть каждое действие, которое мы хотим выполнить на кластере, требует определенных ресурсов (память, вычислительные мощности и т.д.). На основе этого решается, где именно все будет запускаться. Физических планов может быть несколько, в зависимости от конфигурации кластера, от доступных ресурсов и т.д. За те секунды, пока оптимизатор обсчитывает, он должен выбрать один из физических планов.
- Физический план из общего набора физических планов, рассчитанных оптимизатором, выбирается по методу ветвей и границ. Примерно просчитывается затратность его выполнения, а лишние ветви (то, что нам не подходит) обрезаются. Так, остается один из конкретных планов.

Кейс – исходные данные

Рассмотрим реальный кейс:

```
from pyspark.sql.functions import sum
from decimal import Decimal

inventory_schema = ("id integer, name string, price decimal(20,2)")
orders_schema =("id integer, item_id integer, count integer")
items = spark.createDataFrame([[0, "Cheese", Decimal(2.1)], \
                               [1, "Wine", Decimal(6.2)], \
                               [2, "Bread", Decimal(0.8)]], \
                              schema=inventory_schema)
orders = spark.createDataFrame([[100, 0, 1], \
                                [100, 1, 1], \
                                [101, 2, 3], \
                                [102, 2, 8]],\
                               schema=orders_schema)
```

У нас есть исходные данные и несколько dataframʼов, которые мы будем джойнить между собой.

Смотрим запрос

```
counts = (items.join(orders,items.id == orders.item_id, how="inner")) \
         .where(items.id == 2) \
         .groupBy("name", "price").agg(sum("count").alias("units"))
```

Попробуем посчитать по этому коду базовый джойн. Объединим по айдишникам датасеты, выберем одно значение (с groupBy, агрегацией и т.д.) и вызовем функцию «.explain»:

```
counts.explain()
```

Если вы базово знаете SQL, то вам известно, что в БД тоже есть оператор explain, который показывает план выполнения. В Spark происходит почти так же, так как Spark близок по схожести с SQL.

Физический план

Если все пройдет успешно, то должно получиться следующее:

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=true
+- == Final Plan ==
   *(6) HashAggregate(keys=[name#1, price#2], functions=[sum(count#8)])
   +- AQEShuffleRead coalesced
      +- ShuffleQueryStage 2
         +- Exchange hashpartitioning(name#1, price#2, 200), ENSURE_REQUIREMENTS, [plan_id=368]
            +- *(5) HashAggregate(keys=[name#1, price#2], functions=[partial_sum(count#8)])
               +- *(5) Project [name#1, price#2, count#8]
                  +- *(5) SortMergeJoin [id#0], [item_id#7], Inner
                     :- *(3) Sort [id#0 ASC NULLS FIRST], false, 0
                     :  +- AQEShuffleRead coalesced
                     :     +- ShuffleQueryStage 0
                     :        +- Exchange hashpartitioning(id#0, 200), ENSURE_REQUIREMENTS, [plan_id=272]
                     :           +- *(1) Filter (isnotnull(id#0) AND (id#0 = 2))
                     :              +- *(1) Scan ExistingRDD[id#0,name#1,price#2]
                     +- *(4) Sort [item_id#7 ASC NULLS FIRST], false, 0
                        +- AQEShuffleRead coalesced
                           +- ShuffleQueryStage 1
                              +- Exchange hashpartitioning(item_id#7, 200), ENSURE_REQUIREMENTS, [plan_id=279]
                                 +- *(2) Project [item_id#7, count#8]
                                    +- *(2) Filter ((item_id#7 = 2) AND isnotnull(item_id#7))
                                       +- *(2) Scan ExistingRDD[id#6,item_id#7,count#8]
```

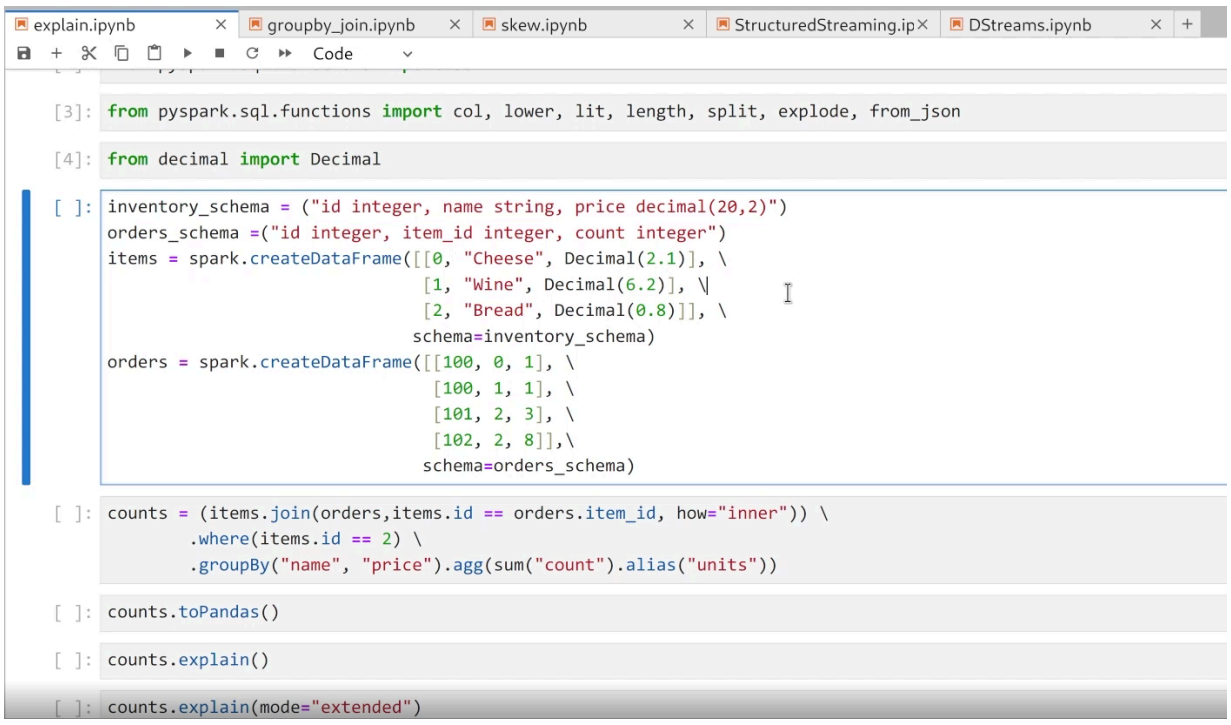
Это выбранный физический план, который получился в результате применения Spark оптимизатора. Все операции, которые мы делали, выстраиваются по порядку. В этом порядке они и будут выполняться, причем часть из них будет обрабатываться в параллель.

Мы можем присмотреться и решить, верно ли оптимизатор выбрал подходящий план.

Важно понимать, что вы получите немного отличающиеся планы в зависимости от того, на какой версии Spark вы это запускаете. Например, данный план для одной из последних версий Spark. В старых версиях Spark вообще не существовало **AdaptiveSparkPlan**. Это более продвинутый план, имеющий под собой дополнительные шаги. Он может меняться в процессе исполнения, а именно в шагах, где написано **AQES**. Это значит, что в процессе выполнения плана, Spark делает один из нескольких

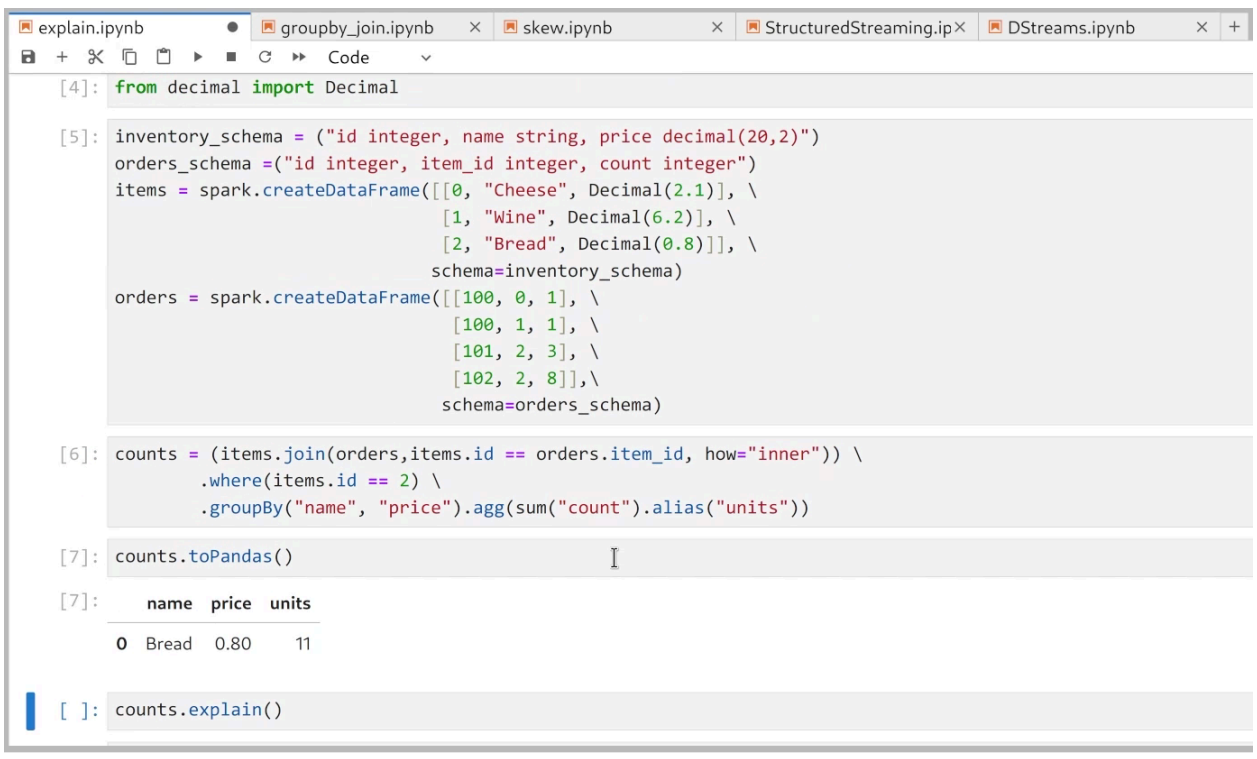
Дедлайн 07 июля, 23:59 Мск

Обратите внимание на следующий код:



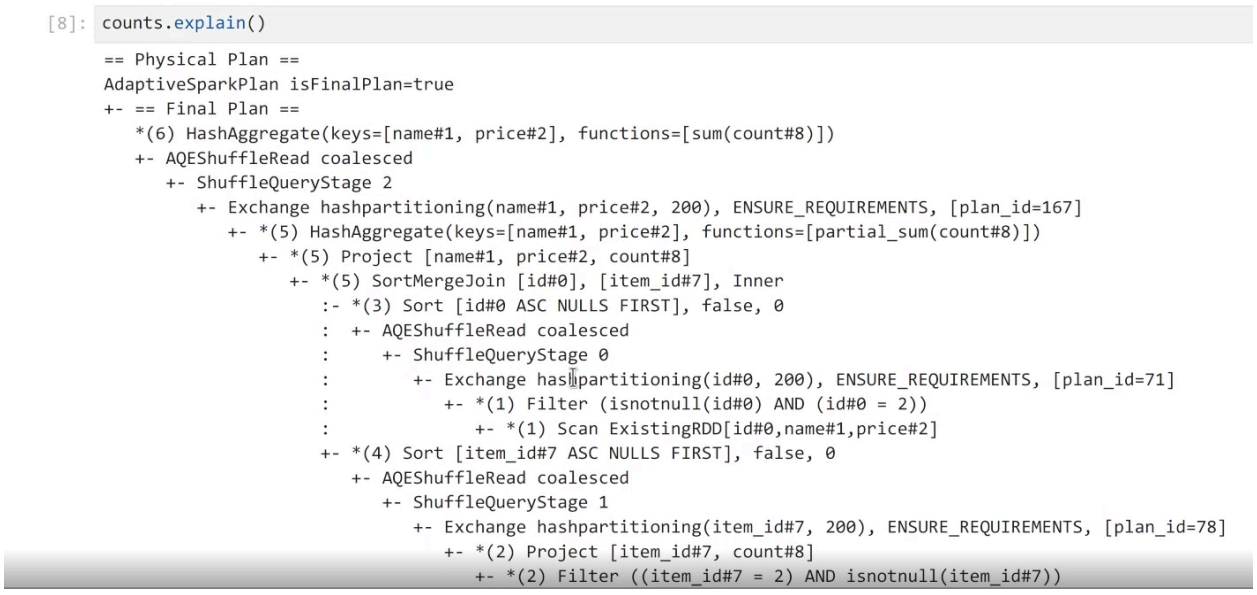
У нас еще не было подобного синтаксиса задания схемы. Можно создать колонки через запятую, можно сделать красивую схему с импортами и т.д., а можно считерить и сделать колонки через запятую.

Можно попробовать выгрузить все в Pandas Dataframe для того, чтобы проверить, что все вычисляется и работает. Получается следующее:



Так получается, поскольку мы берем значение по одному конкретному айдишнику.

Вызовем explain. Видим следующее:



Мы получили физический план. Можно увидеть, что есть **Initial Plan** – физический план, составленный Spark на первом этапе:

Дедлайн 07 июля, 23:59 Мск

```
+- == Initial Plan ==
   HashAggregate(keys=[name#1, price#2], functions=[sum(count#8)])
   +- Exchange hashpartitioning(name#1, price#2, 200), ENSURE_REQUIREMENTS, [plan_id=44]
      +- HashAggregate(keys=[name#1, price#2], functions=[partial_sum(count#8)])
         +- Project [name#1, price#2, count#8]
            +- SortMergeJoin [id#0], [item_id#7], Inner
               :- Sort [id#0 ASC NULLS FIRST], false, 0
               :  +- Exchange hashpartitioning(id#0, 200), ENSURE_REQUIREMENTS, [plan_id=36]
               :     +- Filter (isnotnull(id#0) AND (id#0 = 2))
               :        +- Scan ExistingRDD[id#0,name#1,price#2]
               +- Sort [item_id#7 ASC NULLS FIRST], false, 0
                  +- Exchange hashpartitioning(item_id#7, 200), ENSURE_REQUIREMENTS, [plan_id=37]
                     +- Project [item_id#7, count#8]
                        +- Filter ((item_id#7 = 2) AND isnotnull(item_id#7))
                           +- Scan ExistingRDD[id#6,item_id#7,count#8]
```

Помимо физического плана существует и логический. В старых версиях Spark было два варианта посмотреть план:

- **обычный**

```
counts = (items.join(orders,items.id == orders.item_id, how="inner")) \
         .where(items.id == 2) \
         .groupBy("name", "price").agg(sum("count").alias("units"))
```

- extended

```
counts.explain(mode="extended")
```

На сегодняшний день появилась целая пачка вариантов с разными аргументами. Можно указать конкретный mode. Мы получим не только физический план, но и логический в самом начале:

```
== Parsed Logical Plan ==
'Aggregate ['name, 'price], ['name, 'price, sum('count) AS units#82]
+- Filter (id#0 = 2)
   +- Join Inner, (id#0 = item_id#7)
      :- LogicalRDD [id#0, name#1, price#2], false
      +- LogicalRDD [id#6, item_id#7, count#8], false

== Analyzed Logical Plan ==
name: string, price: decimal(20,2), units: bigint
Aggregate [name#1, price#2], [name#1, price#2, sum(count#8) AS units#82L]
+- Filter (id#0 = 2)
   +- Join Inner, (id#0 = item_id#7)
      :- LogicalRDD [id#0, name#1, price#2], false
      +- LogicalRDD [id#6, item_id#7, count#8], false

== Optimized Logical Plan ==
Aggregate [name#1, price#2], [name#1, price#2, sum(count#8) AS units#82L]
+- Project [name#1, price#2, count#8]
   +- Join Inner, (id#0 = item_id#7)
      :- Filter (isnotnull(id#0) AND (id#0 = 2))
      :  +- LogicalRDD [id#0, name#1, price#2], false
      +- Project [item_id#7, count#8]
         +- Filter ((item_id#7 = 2) AND isnotnull(item_id#7))
            +- LogicalRDD [id#6, item_id#7, count#8], false
```

Логический план

Возможна ситуация, когда мы, на основе написанного нами кода, лучше понимаем, как нужно запускать агрегаты и т.д.

Посмотрим на то, как выглядит логический план. Запустим explain с модом «Extended» и увидим полную картину:

```
[9]: counts.explain(mode="extended")
```

```
== Parsed Logical Plan ==
'Aggregate ['name, 'price], ['name, 'price, sum('count) AS units#31]
+- Filter (id#0 = 2)
   +- Join Inner, (id#0 = item_id#7)
      :- LogicalRDD [id#0, name#1, price#2], false
      +- LogicalRDD [id#6, item_id#7, count#8], false

== Analyzed Logical Plan ==
name: string, price: decimal(20,2), units: bigint
Aggregate [name#1, price#2], [name#1, price#2, sum(count#8) AS units#31L]
+- Filter (id#0 = 2)
   +- Join Inner, (id#0 = item_id#7)
      :- LogicalRDD [id#0, name#1, price#2], false
      +- LogicalRDD [id#6, item_id#7, count#8], false

== Optimized Logical Plan ==
Aggregate [name#1, price#2], [name#1, price#2, sum(count#8) AS units#31L]
+- Project [name#1, price#2, count#8]
   +- Join Inner, (id#0 = item_id#7)
      :- Filter (isnotnull(id#0) AND (id#0 = 2))
      :  +- LogicalRDD [id#0, name#1, price#2], false
      +- Project [item_id#7, count#8]
```

```
== Analyzed Logical Plan ==
name: string, price: decimal(20,2), units: bigint
Aggregate [name#1, price#2], [name#1, price#2, sum(count#8) AS units#31L]
+- Filter (id#0 = 2)
   +- Join Inner, (id#0 = item_id#7)
      :- LogicalRDD [id#0, name#1, price#2], false
      +- LogicalRDD [id#6, item_id#7, count#8], false

== Optimized Logical Plan ==
Aggregate [name#1, price#2], [name#1, price#2, sum(count#8) AS units#31L]
+- Project [name#1, price#2, count#8]
   +- Join Inner, (id#0 = item_id#7)
      :- Filter (isnotnull(id#0) AND (id#0 = 2))
      :  +- LogicalRDD [id#0, name#1, price#2], false
      +- Project [item_id#7, count#8]
         +- Filter ((item_id#7 = 2) AND isnotnull(item_id#7))
            +- LogicalRDD [id#6, item_id#7, count#8], false

== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=true
+- == Final Plan ==
   *(6) HashAggregate(keys=[name#1, price#2], functions=[sum(count#8)], output=[name#1, price#2, units#31L])
   +- AQEShuffleRead coalesced
      +- ShuffleQueryStage 2
+- == Initial Plan ==
   HashAggregate(keys=[name#1, price#2], functions=[sum(count#8)], output=[name#1, price#2, units#31L])
   +- Exchange hashpartitioning(name#1, price#2, 200), ENSURE_REQUIREMENTS, [plan_id=44]
      +- HashAggregate(keys=[name#1, price#2], functions=[partial_sum(count#8)], output=[name#1, price#2, sum#36L])
         +- Project [name#1, price#2, count#8]
            +- SortMergeJoin [id#0], [item_id#7], Inner
               :- Sort [id#0 ASC NULLS FIRST], false, 0
               :  +- Exchange hashpartitioning(id#0, 200), ENSURE_REQUIREMENTS, [plan_id=36]
               :     +- Filter (isnotnull(id#0) AND (id#0 = 2))
               :        +- Scan ExistingRDD[id#0,name#1,price#2]
               +- Sort [item_id#7 ASC NULLS FIRST], false, 0
                  +- Exchange hashpartitioning(item_id#7, 200), ENSURE_REQUIREMENTS, [plan_id=37]
                     +- Project [item_id#7, count#8]
                        +- Filter ((item_id#7 = 2) AND isnotnull(item_id#7))
                           +- Scan ExistingRDD[id#6,item_id#7,count#8]
```

У нас будет не только физический план, но и логический (со всеми промежуточными).

Когда вы работаете со Spark, довольно большую долю времени будут занимать именно эти планы.

Дедлайн 07 июля, 23:59 Мск