
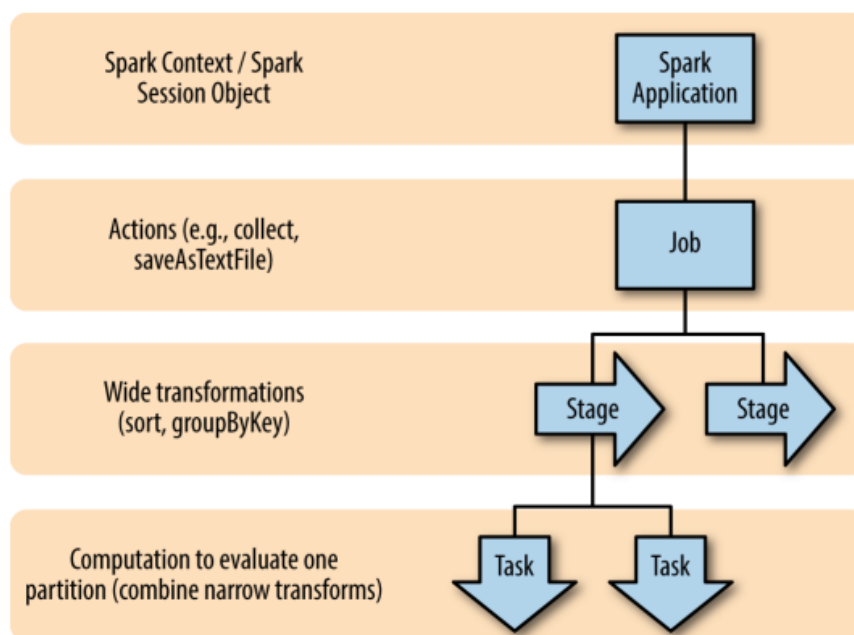


План выполнения запроса, DAG

☰ Tags	
📎 Files & media	

▼ Единицы



Задача (Job)

Задача (Job) — это последовательность **этапов (stages)**, инициируемая **действиями (actions)**, такими как `.count()`, `.foreachRDD()`, `.sortBy()`, `read()` или `write()`. Каждый раз, когда происходит вызов действия, Spark создает **задачу (job)**, которая включает одно или несколько **преобразований (transformations)**, которые были применены к данным до этого.

Этап (Stage)


Этап (Stage) — это набор **задач (tasks)**, которые могут быть выполнены параллельно, без необходимости в shuffle-операциях. Количество задач в этапе зависит от количества **разделов (partitions)** в вашем наборе данных. Spark разбивает задачу на этапы на

основании **логики выполнения DAG (Directed Acyclic Graph)**. В Spark можно выделить два типа этапов:

1. **ShuffleMapStage**: Этап, результаты которого являются промежуточными данными для следующего этапа.
2. **ResultStage**: Этап, на котором выполняются действия и результаты отправляются драйверу.

Задачи (Tasks)

Задача (Task) — это минимальная единица выполнения Spark. Каждая задача применяется к **одной партиции** данных и выполняется **на одном исполнителе** (executor).

 *Пример: если набор данных имеет 2 партиции, то операция фильтрации (`filter()`) вызовет две задачи, по одной на каждый раздел. Планировщик задач (TaskScheduler) отвечает за распределение задач между исполнителями, обеспечивая эффективное использование ресурсов кластера.*

▼ DAG (Ориентированные ациклический граф)

Разберем каждое слово:

1. **Граф** означает, что это структура, состоящая из узлов. Некоторые из них могут быть соединены между собой ребрами.
2. **Ацикличность** означает, что в графе нет циклов. Цикл может быть обнаружен при обходе графа, когда один конкретный узел посещается более одного раза.
3. Наконец, **ориентированный** граф — это граф, в котором отношения между узлами имеют направление. Например, отношение (1)-(2) не является направленным, поскольку оно связывает только 2 узла. С другой стороны, (1)->(2) направлено

Вершинами графа у нас являются RDD/Dataframe, а ребра представляют операции, которые должны выполняться на этих RDD/Dataframe.

DAG — Логический план выполнения запроса (Logical plan)

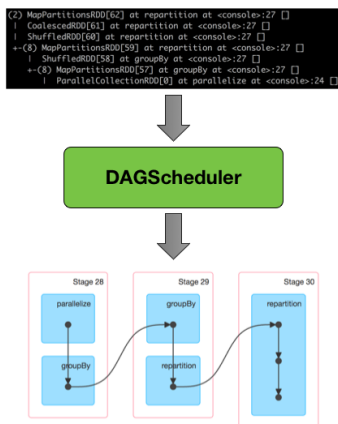
DAG помогает создавать окончательные результаты. Но не только. Его дополнительная функция обеспечивает **отказоустойчивость**. Представьте, что фильтрация выполняется

на двух разных узлах, и один из них дает сбой. С узлом терпят неудачу все данные, которые он содержал. Теперь Spark найдет другой **узел**, способный обработать неудачные запросы. Но у этого узла нет данных, необходимых для работы фильтра. Именно в этот момент этот узел может прочитать DAG и выполнить все родительские преобразования неудачного шага. Благодаря этому он получает те же данные, что и данные отказавшего узла.

DAG scheduler



DAG Scheduler — это компонент планировщика в Apache Spark, который управляет распределением задач и этапов в Spark-приложении на основе построенного графа зависимостей операций (DAG — Directed Acyclic Graph).



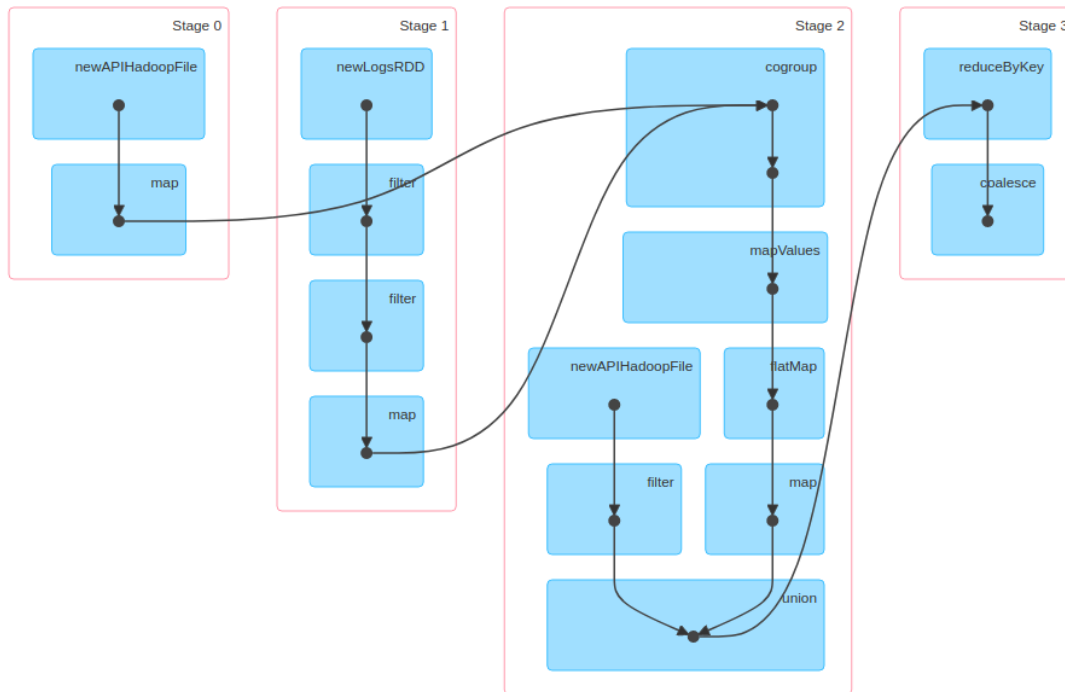
После того как для RDD вызывается действие, **SparkContext** передает логический план в **DAGScheduler**, который преобразует его в **набор этапов (Stages)**, отправляемых для выполнения в виде **TaskSets**.

Выполнение действия приводит к созданию нового **ResultStage** и **ActiveJob** в **DAGScheduler**.

Глянуть DAG можно через UI (в веб интерфейсе можно на джобы посмотреть), либо `toDebugString()` - который показывает, что у нас делает над каждым этапом RDD.

Обязанности:

- Просчитывает Execution DAG, то есть DAG of stages для джобы.
- Определяет предпочитаемую локацию для запуска таски (локальность данных)
- Обработка сбоев из-за потери выходных файлов shuffle.



Обрати внимание!

У RDD в Spark **не строится** отдельный план выполнения запроса, подобный тому, что создается для **DataFrame** или **Dataset**. **RDD**: Spark не строит **логический план** для RDD. Вместо этого он просто выполняет операции в том порядке, в котором они были указаны, без предварительного анализа и оптимизации (lineage).

▼ Ленивые вычисления



Ленивые вычисления — операции над данными (трансформации) не выполняются немедленно после их вызова. Вместо этого Spark откладывает выполнение до тех пор, пока не будет вызвана операция действия (**Action**), например, `collect()`, `count()` или `saveAsTextFile()`. Только при вызове Action Spark начинает выполнение всех накопленных трансформаций, что позволяет оптимизировать вычисления и избежать излишней работы.

Оптимизация за счет ленивых вычислений

Одним из главных преимуществ ленивых вычислений является то, что Spark может **оптимизировать** выполнение, комбинируя операции:

- Например, если вы вызываете `map()` и `filter()` на одном RDD, Spark не будет выполнять эти операции отдельно. Вместо этого он отправит инструкции на выполнение обеих операций одновременно, что снизит количество проходов по данным.
- Это уменьшает накладные расходы на вычисления и делает программу более эффективной.

✓ Преимущества ленивых вычислений

- **Улучшенная производительность:** Ленивые вычисления позволяют Spark объединять операции, что снижает количество операций над данными и минимизирует количество их проходов. Например, вместо того чтобы дважды проходить по данным при применении `map` и `filter`, Spark выполнит обе операции за один проход.
- **Меньше кода:** Поскольку Spark автоматически объединяет и оптимизирует операции, вам не нужно вручную управлять сложными зависимостями, как это требуется в других системах, таких как MapReduce.
- **Fault-tolerance (устойчивость к сбоям):** Поскольку RDD содержит информацию о своей родословной (lineage), Spark может восстановить утраченные данные, просто пересчитав нужные разделы (partitions).

Ленивые вычисления и отладка

Ленивые вычисления могут усложнить процесс отладки, так как ошибки могут возникнуть только на этапе **действия**. Например, если вы допустите ошибку в трансформации, ошибка появится только в тот момент, когда будет вызвано действие (например, `collect()`).

- Это означает, что ошибки могут не проявляться сразу, что затрудняет отладку.
- **Решение:** Использовать промежуточные действия, такие как `show()` или `take()`, чтобы проверить результат на ранних этапах.

▼ Теоритический минимум

▼ Spark Metastore



Spark Metastore — это хранилище метаданных о структуре и содержании данных. В нем хранятся информация о базах данных, таблицах, колонках, типах данных и других аспектах данных, необходимых для выполнения запросов и управления данными в Spark. Метастор часто используется вместе с технологиями типа Apache Hive, но его также можно интегрировать в другие системы.

Spark Metastore

Основная цель **Metastore** — это централизованное хранилище метаданных для управления структурой данных и их схемами, а также предоставление информации о расположении данных и их типах. Эта информация нужна Spark для правильного построения и выполнения запросов, особенно при работе с табличными данными в форматах, таких как Hive, Parquet, Delta Lake и других.

Что хранит Metastore?

- **Базы данных (Databases):** наборы таблиц.
- **Таблицы (Tables):** наборы данных, структурированных в виде строк и столбцов.
- **Колонки (Columns):** метайнформация о типах данных, связанных с колонками в таблицах.
- **Разделы (Partitions):** используется для ускорения запросов и оптимизации за счет организации данных по частям (например, по дате или региону).
- **Индексы:** вспомогательная информация для ускорения доступа к данным.
- **Формат хранения:** указания на то, как данные хранятся (например, Parquet, ORC).

▼ Catalog



Catalog — это компонент, который управляет метаданными в приложении Spark, используя данные из метастора. Он предоставляет **API** для работы с базами данных, таблицами и другими объектами, упрощая доступ и управление метаданными. Catalog взаимодействует с метастором для получения информации о таблицах, базах данных и схемах, которые используются в Spark.

Задачи Catalog:

1. **Чтение метаданных:** Catalog позволяет Spark получать доступ к информации о базах данных, таблицах, колонках и других объектах, хранящихся в метасторе.
2. **Управление таблицами и базами данных:** через Catalog можно создавать, удалять и изменять таблицы и базы данных.
3. **Схемы и колонки:** Catalog предоставляет информацию о схемах данных (типы колонок, разделы) для правильной работы с ними в запросах.
4. **Интеграция с Hive:** Spark часто использует Hive Metastore в качестве источника метаданных, и в этом случае Spark Catalog взаимодействует с Hive Metastore для получения нужной информации.

Пример работы Catalog и Metastore

Предположим, у нас есть таблица в формате Parquet, которая сохранена в Hive Metastore, и мы хотим выполнить запрос через Spark:

1. **Создание запроса:** Пользователь пишет запрос на SQL или использует DataFrame API.
2. **Запрос в Catalog:** Spark через Catalog обращается в Metastore для получения информации о таблице (ее схема, типы данных, местоположение файлов).
3. **Метаданные:** Catalog получает метаданные из Metastore, такие как разделы таблицы, колонки, типы данных.
4. **Выполнение запроса:** Spark использует полученные метаданные для оптимизации и выполнения запроса.

▼ Analyzer



Analyzer — это компонент, который выполняет **семантический анализ** логического плана запроса. Его основная задача — **разрешить все ссылки** на объекты данных (такие как таблицы, колонки, функции и схемы), чтобы убедиться, что они корректны и согласуются с метаданными. Это важный этап в процессе оптимизации запросов, который идет сразу после построения неразрешенного логического плана (**Unresolved Logical Plan**).

Основные задачи Analyzer:

1. Проверка и разрешение ссылок на объекты:

- **Разрешение имен колонок:** Analyzer проверяет, существуют ли указанные в запросе колонки, и заменяет их общие ссылки (например, просто `name`) на уникальные идентификаторы (например, `name#1`). Это делает план конкретным и позволяет избежать конфликтов имен.
- **Разрешение ссылок на таблицы:** Если в запросе указана таблица, Analyzer проверяет наличие этой таблицы в **Catalog** и получает ее схему (имена и типы колонок). Это важно для правильного анализа запроса и последующего выполнения.
- **Проверка типов данных:** Analyzer проверяет, соответствуют ли типы данных операциям. Например, он убедится, что к числовому столбцу не применяется строковая операция, такая как `LIKE`.

2. Работа с метаданными через Catalog:

- Analyzer взаимодействует с **Catalog** для получения информации о таблицах и колонках, таких как схема данных, типы и разделы. Это позволяет Spark проверять, существуют ли объекты, указанные в запросе, и использовать правильные типы данных для дальнейшей обработки.

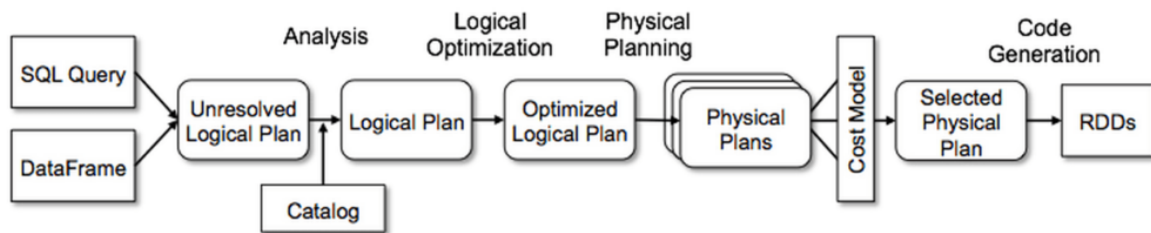
3. Семантический анализ:

- На этом этапе Spark выполняет **семантический анализ**, чтобы убедиться, что все части запроса корректны. Семантический анализ включает проверку типов данных, имен столбцов и других аспектов запроса.

4. Разрешение неопределенностей:

- Если в исходном логическом плане (Unresolved Logical Plan) есть неопределенные элементы (например, неопознанные колонки или неясные типы данных), Analyzer пытается разрешить эти неопределенности с помощью данных из **Catalog**. Если в запросе есть ошибки, такие как указание несуществующего столбца, на этом этапе они будут выявлены.

▼ Как формируется план запроса?



В Spark процесс выполнения запроса включает несколько этапов, где важнейшим является трансформация логического плана в физический. Логический план описывает, какие операции нужно выполнить, а физический — как именно Spark будет их выполнять, учитывая доступные ресурсы и особенности данных.

1. Unresolved Logical Plan (Неразрешенный логический план)

Это самый первый шаг в процессе обработки запроса. **Unresolved Logical Plan** — это начальная форма плана, которую Spark строит на основе вашего кода (например, SQL-запроса или DataFrame-операций). На этом этапе план "неразрешенный", потому что Spark еще не проверил его на наличие ошибок, и не все элементы плана могут быть полностью определены. Например, на этом этапе могут быть ссылки на несуществующие столбцы, таблицы или функции (причем генерировать и хранить логический план будет **SparkContext**).

Пример:

```
df.select("name").filter("age > 30")
```

Неразрешенный логический план может выглядеть так:

```
Project [name]
+- Filter (age > 30)
   +- Relation[unknown schema]
```

Особенности:

- Spark не знает, что такое `name` или `age`. Он просто сохраняет эту информацию как "проект" на будущее разрешение.
- Spark также не знает, как выглядит источник данных — у него пока нет информации о схеме данных.

2. Resolved Logical Plan (Разрешенный логический план)

На этом этапе Spark начинает "разрешать" логический план, т.е. он проверяет наличие всех указанных столбцов, таблиц и других сущностей. Spark обращается к метаданным и схеме данных, чтобы уточнить, существуют ли указанные таблицы, столбцы и функции.

Analyzer в Spark выполняет **семантический анализ** на основе данных из метакранилища (Catalog). Этот процесс включает проверку на корректность типов данных, соответствие имен столбцов, типов данных и других семантических правил.



Обрати внимание!

DataFrame и **Dataset** в Spark считаются **semi-lazy** (полу-ленивыми) структурами данных. Это означает, что они начинают процесс анализа и оптимизации сразу при создании, но их реальное выполнение происходит только при вызове **Action** (действия).

Анализ и оптимизация: после создания логического плана Spark немедленно начинает выполнять его анализ и оптимизацию через **Analyzer** и **Catalyst Optimizer**, используя данные из метастора для разрешения схемы данных, имен колонок и типов данных.

В отличие от DataFrame и Dataset,

RDD (Resilient Distributed Dataset) в Spark является **полностью ленивой** структурой данных.

Пример:

После разрешения логический план может выглядеть так:

```
Project [name#1]
+- Filter (age#2 > 30)
   +- Relation[name#1, age#2]
```

Особенности:

- Теперь Spark знает, что столбцы `name` и `age` существуют в таблице. В результате он добавляет идентификаторы столбцов (`name#1` , `age#2`), что делает план более конкретным.
- Spark также знает схему таблицы и может "разрешить" все ссылки на данные.

3. Optimized Logical Plan (Оптимизированный логический план)

После того как Spark разрешил все элементы плана, он применяет **Catalyst Optimizer** для улучшения эффективности выполнения запроса. Оптимизации могут включать слияние операций, перестановку фильтров и выборок, удаление ненужных операций и

другие преобразования. Catalyst использует **rule-based transformations** (правила преобразования), чтобы улучшить план выполнения запросов.

- **Объединение задач:** Если несколько операций могут быть выполнены на одном этапе, Spark объединяет их для повышения эффективности.
- **Оптимизация порядка выполнения запросов:** Spark может изменить порядок выполнения операций, чтобы ускорить выполнение запросов с несколькими объединениями (joins).
- **Оптимизация фильтров:** Spark старается применить фильтры как можно раньше, до выполнения других операций, таких как выборка столбцов, чтобы уменьшить объем данных для обработки.

Примеры оптимизаций:

- **Predicate Pushdown (Проталкивание предикатов):** Spark может переместить фильтр на более ранний этап, чтобы снизить объем данных, с которыми нужно работать.
- **Projection Pushdown (Проталкивание выборки столбцов):** если запрос использует только несколько столбцов, Spark может избежать загрузки ненужных столбцов.

Пример:

```
df.select("name").filter("age > 30")
```

После оптимизации план может выглядеть так:

```
Project [name#1]
+- Filter (age#2 > 30)
   +- Relation[name#1, age#2]
```

4. Physical Plan (Физический план)

Теперь, когда логический план оптимизирован, Spark начинает преобразование в **физический план**.



Physical Plan — это конкретная стратегия выполнения операций (например, чтение данных, соединение таблиц, фильтрация), учитывающая доступные ресурсы и предполагаемую стоимость выполнения.

Этапы:

- **Catalyst Optimizer** создает несколько альтернативных физических планов.
- Каждый план оценивается по времени выполнения и ресурсам, необходимым для его реализации.
- Spark выбирает наилучшую стратегию выполнения.

Затем Spark создает **Directed Acyclic Graph (DAG)**, который состоит из **RDD** (Resilient Distributed Dataset). Этот DAG разбивается на **стадии** (stages), каждая из которых включает набор задач (**tasks**), которые будут выполняться параллельно на узлах кластера.

Физический план состоит из набора операторов, каждый из которых отвечает за выполнение конкретной задачи. Эти операторы включают:

- **FileScan** — оператор для чтения данных из файлов.
- **Filter** — оператор для фильтрации данных.
- **Project** — оператор для выборки нужных столбцов.
- **Exchange** — оператор, который отвечает за перераспределение данных между узлами кластера.

Пример физического плана:


```
== Physical Plan ==
*(1) Project [name#1]
+- *(1) Filter (age#2 > 30)
   +- *(1) FileScan parquet [name#1, age#2]
```

Особенности:

- **FileScan** указывает, что Spark будет сканировать данные в формате Parquet и загружать только нужные столбцы (`name` и `age`).
- **Filter** указывает, что фильтрация будет применена к столбцу `age` , и только строки с возрастом больше 30 будут переданы дальше.
- **Project** указывает, что из этих строк будет выбран только столбец `name` .

5. Code Generation (Whole-Stage Codegen)

Этот шаг является одной из уникальных черт Spark. После создания физического плана Spark может применить **Whole-Stage Codegen**, что позволяет сгенерировать оптимизированный байт-код для выполнения плана. Whole-Stage Codegen уменьшает количество инструкций и улучшает производительность за счет объединения нескольких операций в одну скомпилированную функцию.

 **Пример:** Вместо того чтобы выполнять каждую операцию по отдельности (например, сначала фильтровать данные, а затем выбирать столбцы), Spark может сгенерировать единый байт-код, который выполнит обе операции вместе.

6. Execution (Выполнение)

После того как физический план построен и сгенерирован код, Spark приступает к его выполнению. Выполнение происходит на нескольких узлах кластера параллельно. Данные делятся на **partition** (разделы), и каждый узел выполняет свою часть работы. Spark использует **executor** — процессы, которые выполняют задачи (tasks) на разных узлах.

Каждая задача соответствует одной стадии выполнения физического плана, а на каждом узле выполняется набор задач, связанных с определенной частью данных.⁴



Обрати внимание! Кэширование и менеджер кэша

На этапе выполнения плана Spark может обращаться к **менеджеру кэша**. Если результаты предыдущего запроса были сохранены в кэше и совпадают с текущим планом, Spark использует эти данные, чтобы избежать повторного выполнения тех же операций. Это позволяет значительно ускорить выполнение запросов.

7. Adaptive Query Execution (AQE) [Подробнее [здесь](#)]

С версии Spark 3.0 появилась новая функция — **Adaptive Query Execution (AQE)**. AQE позволяет динамически изменять физический план во время выполнения запроса на основе фактических данных. Например:

- **Изменение количества разделов (shuffle partitions):** если Spark обнаруживает, что данные распределены неравномерно, он может уменьшить или увеличить количество разделов для более эффективного выполнения.
- **Динамическое обрезание партиций (Dynamic Partition Pruning):** Spark может не сканировать ненужные партиции, если видит, что они не участвуют в запросе.

Пример процесса от начала до конца:

1. **Unresolved Logical Plan:** Spark получает запрос, но еще не знает всех деталей, таких как схема данных или существование столбцов.
2. **Resolved Logical Plan:** Spark разрешает все ссылки на столбцы и таблицы, уточняет схему данных.
3. **Optimized Logical Plan:** Catalyst Optimizer применяет правила оптимизации, улучшая логический план.
4. **Physical Plan:** Spark решает, как именно выполнять запрос на уровне кластера, выбирая физические операторы.
5. **Code Generation:** Spark генерирует байт-код для эффективного выполнения запроса.
6. **Execution:** запрос выполняется на узлах кластера.
7. **Adaptive Query Execution:** Spark может динамически оптимизировать выполнение запроса во время его работы, если обнаружит новые данные или условия.