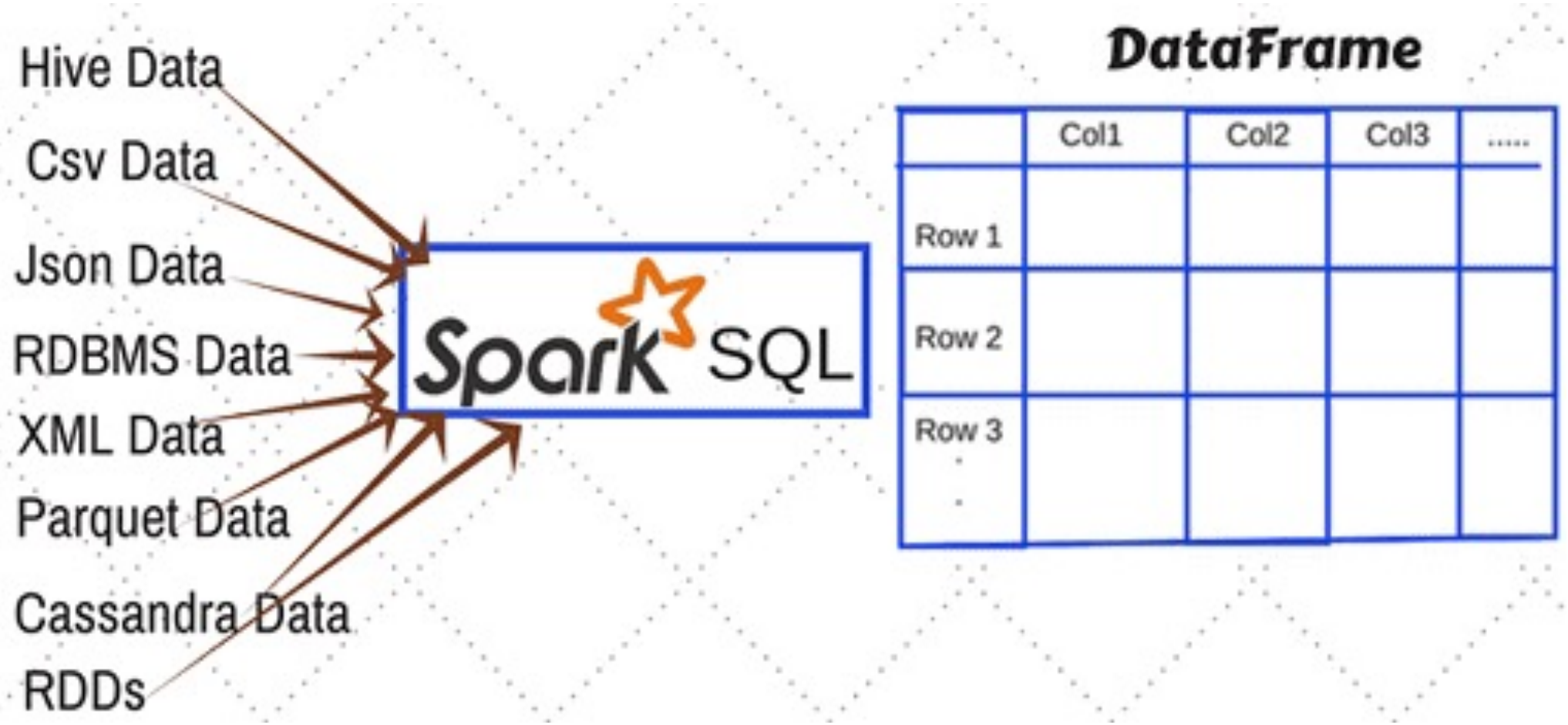


Spark DataFrame

О чем поговорим

1) DataFrame
2) Catalyst

DataFrame

Это распределенная коллекция данных в виде именованных столбцов, аналогично таблице в реляционной базе данных. DataFrame работает только со структурированными и полуструктурированными данными, организуя информацию по столбцам, как в реляционных таблицах. Это позволяет Spark управлять схемой данных.

- Автоматически определяет схему данных
- DataFrame обрабатывается быстрее RDD (классно, когда есть схема данных и можно делать больше оптимизаций)
- DataFrame представляет концепцию схемы для описания данных, позволяя Spark управлять схемой и передавать данные только между узлами гораздо более эффективным способом, чем при использовании сериализации Java
- API DataFrame кардинально отличается от API RDD поскольку это API для построения плана реляционных запросов, который затем может выполнить оптимизатор Sparks Catalyst. API является естественным для разработчиков, которые знакомы с построением планов запросов

Как создать DataFrame

Создать DataFrame можно из большого числа объектов/источников, основные:

- RDD
- Csv
- Txt
- Xml
- Json
- Parquet
- HDFS
- DBFS
- S3
- Avro
- ORV
- Из различных объектов python, в том числе pandas

В курсе все рассматривать не будем, для таких вещей есть гугл.

А можно потом в pandas?

Да, можно, если очень нужно:

pandas_df = df.toPandas()

Но бывают ситуации, что данных очень много, а у java есть ограничения на память worker’а и будем ловить что-то вроде java heap space.

Как с этим жить?

df.toLocalIterator()

Но оба варианта долгие и лучше сделать так: сохранили паркет и прочитали его через pandas.

Show

.show(n=20, truncate=True, vertical=False)

- n – количество записей для отображения
- truncate – обрезать ли данные, False – не трогать, 20 – до 20 символов на запись, True – 20 символов по умолчанию будут отображаться
- vertical – отображать вертикально или горизонтально

Sample

Sample при пересчете графа может выдавать разные результаты даже при фиксированном seed.

pyspark.sql.DataFrame.sample

`DataFrame.sample(withReplacement: Union[float, bool, None] = None, fraction: Union[int, float, None] = None, seed: Optional[int] = None) → pyspark.sql.dataframe.DataFrame` [source]

Returns a sampled subset of this `DataFrame`.

New in version 1.3.0.

Parameters: **withReplacement** : ***bool, optional***

Sample with replacement or not (default `False`).

fraction : ***float, optional***

Fraction of rows to generate, range [0.0, 1.0].

seed : ***int, optional***

Seed for sampling (default a random seed).

Notes

This is not guaranteed to provide exactly the fraction specified of the total count of the given `DataFrame`.

fraction is required and, *withReplacement* and *seed* are optional.

Оптимизация

Немного про СУБД SQL-оптимизацию

- Логическая оптимизация, когда заданный запрос переписывается на основе эвристик, правил в эквивалентный, но потенциально более декларативную и оптимальную форму.

- Физическая оптимизация предполагает выбор методов доступа, последовательность соединений и методы соединений для генерации эффективного плана выполнения запроса.

Планирования SQL запроса

- Выборка результатов с помощью вложенных циклов (итеративный процесс поиска данных в каждой из запрашиваемых соединяемых таблиц) и слияние, если объединяемые таблицы имеют индексы по сравниваемым полям.

- Сортировка и группировка, если не найдено путей доступа для получения в запрошенном порядке, а также выполнение агрегаций.

Оптимизатор запросов старается выбрать самый эффективный план выполнения запроса. Для этого он использует хранящуюся в базе данных статистическую информацию для оценки альтернативных способов формирования результатов запроса.

Оптимизация SQL запросов

RBO (rule-based optimizator) – оптимизатор на основе фиксированнх правил. Например, применение фильтраций на более ранних этапах, если это возможно.

CBO (cost-based optimizator) – оптимизатор на основе оценки стоимости выполнения запроса. Для оценки качества полученного плана используется стоимостная функция, которая обычно зависит от объема данных, количества строк, попадающих под фильтры, стоимости выполнения тех или иных операций. Для корректной работы оптимизатору необходимо знать и хранить информацию по статистике данных из запроса.

Лучший вариант – использовать оба подхода.

Catalyst

Это оптимизатор запросов, написанных как непосредственно на языке SQL, так и на доменном DataFrame DSL (Domain Structured Language). Благодаря Catalyst SQL-запросы в DataSet и DataFrame выполняются намного быстрее, чем их функциональные аналоги в RDD.

Написан на Scala и является расширяемым - можно самостоятельно определять внешние источники данных и заданные пользователем типы данных.

Библиотеки Catalyst

- Общая библиотека для представления деревьев и применения правил для управления ими;
- Специфичные библиотеки для обработки реляционных запросов, выражений и планов логических запросов;
- Наборы правил, которые обрабатывают различные фазы выполнения запросов: анализ, логическая оптимизация, физическое планирование и генерация кода для компиляции частей запросов в байт-код Java;
- Функции Scala для генерации кода из составных выражений.

План запроса в Catalyst

Это встроенный механизм оптимизации структурированных запросов в Spark SQL, который недоступен в RDD, но появляется в DataFrame. План каждого запроса проходит следующие стадии:

- Parsed Logical Plan - проверяется только синтаксическая корректность запроса.
- Analyzed Logical Plan - добавляется информация о структуре используемых сущностей, проверяется соответствие структуры и запрашиваемых атрибутов.
- Optimized Logical Plan — самое интересное для нас. На данном этапе происходит преобразование получившегося дерева запроса на основании доступных правил оптимизации.
- Physical Plan — начинают учитываться особенности доступа к исходным данным, включая оптимизации по фильтрации партиций и данных для минимизации получаемого набора данных. Выбирается стратегия выполнения join.
- генерация кода для компиляции частей запроса в байт-код Java.

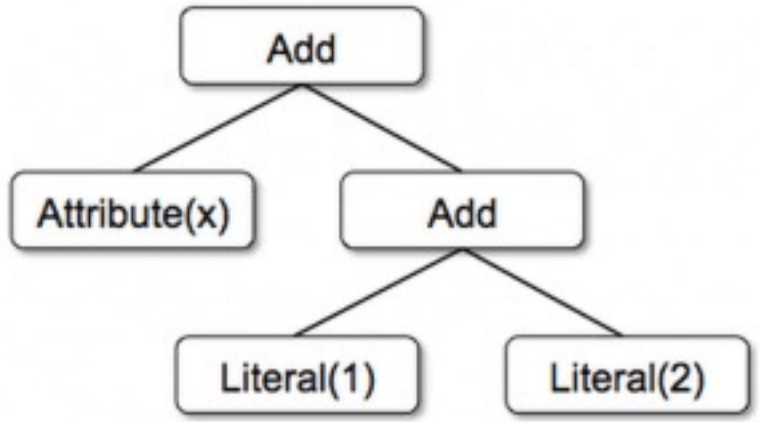
Дерево запроса

Дерево является основным типом данных в Catalyst. Дерево содержит объект узла, который может иметь один или нескольких дочерних элементов.

Например, пусть есть 3 класса узлов:

- Add(left: TreeNode, right: TreeNode) – сумма двух выражений
- Attribute(name: String) – атрибут из входной строки
- Literal(value: int) – постоянное значение.

Тогда дерево операции x + (1 + 2) выглядит так:



Оптимизация логического плана

Дерево запросов появляется на этапе построения оптимизированного логического плана при SQL-оптимизации в Apache Spark с помощью Catalyst.

Типовые правила оптимизации, которые применяются к логическому плану:

- свертка (constant folding);
- предикатное сжатие (predicate pushdown);
- сокращение проекций (projection pruning).

Catalyst позволяет добавлять свои правила для различных ситуаций.

Физическая оптимизация

Один или несколько физических планов формируются из логического с использованием физического оператора, соответствующего движку Apache Spark. Итоговый план выбирается на основе стоимостной модели (CBO, Cost-based optimization). CBO позволяет оценить стоимость рекурсивно для всего дерева с помощью правил.

Физическая оптимизация на основе правил (RBO, Rule-based Optimization), такая как конвейерные проекции или map-фильтры в Spark также выполняется физическим планировщиком. Помимо этого, он может передавать операции из логического плана в источники данных.

Join’s

Самый базовый вариант соединения - Nested Loops Join.

Nested Loops Join. Для каждого элемента первого списка пройдемся по всем элементам второго списка; если ключи элементов вдруг оказались равны — запишем совпадение в результирующую таблицу. Для реализации этого алгоритма достаточно двух вложенных циклов, именно поэтому он так и называется.

Базовый вариант можно улучшить, если одна из таблиц небольшая.

Hash join. Если размер одной из таблиц позволяет поместить ее целиком в память, значит, на ее основе можно сделать хеш-таблицу и быстренько искать в ней нужные ключи (это как если вы будете искать ключи в dict, а не list).

Но с оптимизацией можно пойти и дальше!

Оптимизация join’ов

На этапе физической оптимизации также выбирается тип соединения:

Broadcast hash join. Наилучший вариант в случае если одна из сторон join достаточно мала. В этом случае данная сторона целиком копируется на все executor'ы, где и происходит hash join с основной таблицей.

Sort merge join. Данный способ применяется по умолчанию, если ключи для join можно отсортировать.
Из особенностей можно отметить, что в отличие от предыдущего способа, оптимизация по кодогенерации для выполнения операции доступна только для inner join.

Shuffle hash join. В случае если ключи не поддаются сортировке, либо отключена настройка выбора sort merge join по умолчанию, Catalyst пытается применить shuffle hash join. Помимо проверки на настройки, проверяется также, что Spark хватит памяти, чтобы построить локальный hash map для партиции.

BroadcastNestedLoopJoin и CartesianProduct. В случае, когда отсутствует возможность прямого сравнения по ключу (например, условие по like) или отсутствуют ключи для соединения таблиц, в зависимости от размера таблиц, выбирается один из типов.

Кодогенерация

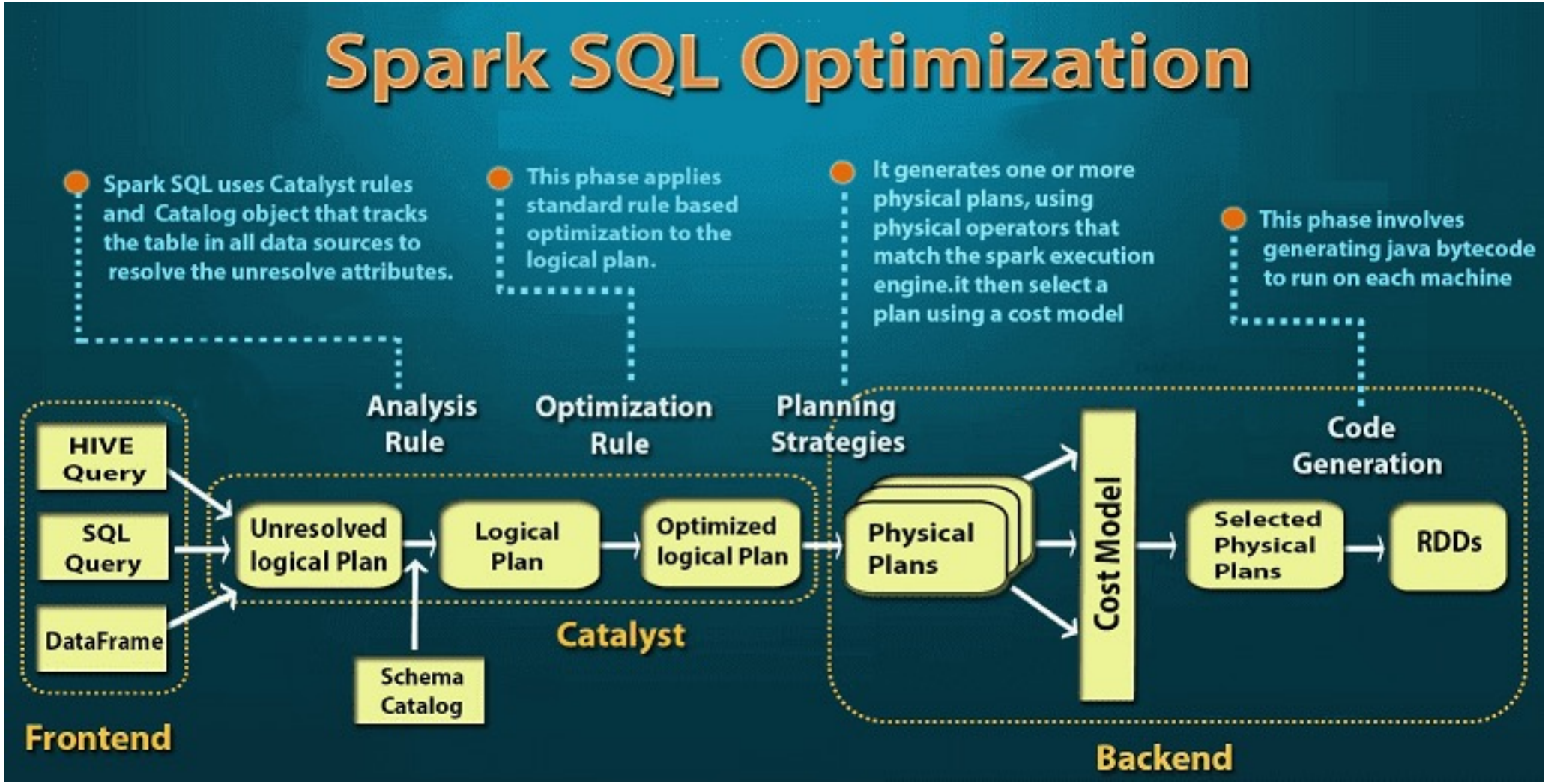
Это достаточно сложный процесс, когда создается байт-код Java для запуска на каждом узле кластера.

Тут много специфических деталей, но если кратко…

Catalyst использует специальную особенность языка Scala, квазиквоты (Quasiquotes), чтобы упростить генерацию кода, потому что очень сложно создавать механизмы генерации кода.

Квазиквоты позволяют на уровне программы создавать абстрактные синтаксические деревья на языке Scala, которые затем могут быть переданы компилятору Scala во время выполнения для генерации байт-кода.

Картинка



https://www.bigdataschool.ru/blog/spark-sql-catalyst-codegeneration.html