МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

ЛЕКЦИОННЫЕ МАТЕРИАЛЫ

Технологии хранения в системах кибербезопасности

(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень	бакалавриат
	(бакалавриат, магистратура, специалитет)
Форма обучения	очная
	(очная, очно-заочная, заочная)
Направление(-я) подготовки	10.05.04 Информационно-аналитические системы безопасности
	(код(-ы) и наименование(-я))
Институт	Кибербезопасности и цифровых технологий (ИКБ)
	(полное и краткое наименование)
Кафедра	КБ-2 «Прикладные информационные технологии»
	(полное и краткое наименование кафедры, реализующей дисциплину (модуль))
Лектор	к.т.н., Селин Андрей Александрович
	(сокращенно – ученая степень, ученое звание; полностью – ФИО)

Используются в данной редакции с учебного года 2024/2025

(учебный год цифрами)

Проверено и согласовано «____» _____2024 г. _____ А.А. Бакаев

(подпись директора Института/Филиала с расшифровкой)

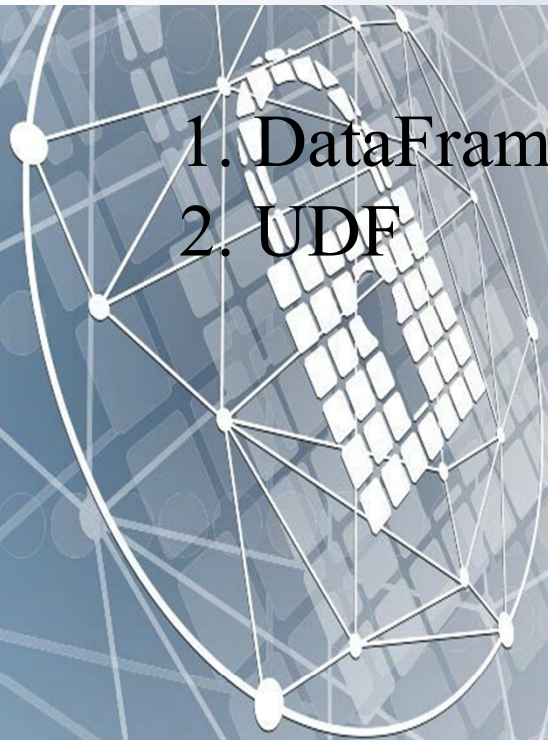
Москва 2024 г.

Технологии хранения в системах кибербезопасности

2024 год

Лекция 15.
Apache Spark. Встроенные и пользовательские функции

Учебные вопросы лекции:



1. DataFrame API: SparkSession
2. UDF

Введение

Современные информационные системы ориентированы на данные, генерирующиеся в огромных количествах ежедневно. Для получения значимых результатов в современных реалиях в промышленных, бизнес- или научных задачах необходимо эффективно обрабатывать доступные большие данные (Big Data), используя разнообразный инструментарий.

DataFrame API: SparkSession

Использование DataFrame API начинается с создания специального объекта SparkSession, служащего входной точкой для дальнейших операций. Пример создания входной точки DataFrame API – объекта SparkSession:

```
val spark = SparkSession.builder
  .appName("user-traces")
  .master("local[*]")
  .config("spark.executor.cores", "8")
  .config("spark.executor.instances", "4")
  .config("spark.cores.max", "128")
  .config("spark.executor.memory", "15g")
  .config("spark.sql.shuffle.partitions", "2000")
  .getOrCreate()
```

SparkSession не заменяет, а дополняет собой базовый объект SparkContext. Последний все еще может быть доступен для получения и использования с помощью обращения к соответствующему полю SparkSession: spark.sparkContext. Как и в SQL, DataFrame API позволяет производить проекции и агрегации над данными с помощью трансформаций с соответствующими названиями.

SparkSession

Пример агрегации данных с помощью DataFrame API с использованием встроенных в Spark SQL агрегационных функций:

```
var followersDf=spark.read.parquet("/storage/followers.parquet")
```

Будем использовать только 3 колонки, это может позволить не читать остальные колонки из хранилища, в том случае если адаптер к хранилищу позволяет произвести такую оптимизацию

```
followersDf = followersDf.select("profile", "key","follower")
```

Группируем по уникальным follower и считаем по группам количество. Выходная схема: follower, following_count

```
import org.apache.spark.sql.functions.count
val following = followersDf
        .groupBy("follower")
        .agg(count("profile").alias("following_count"))
```

Записываем в хранилище

```
following.write.parquet("/storage/counted_followers.parquet")
```

SparkSession

В рассмотренном примере для агрегации данных (подсчета количества фолловеров) используется функция count из стандартной библиотеки Spark SQL.

Как было отмечено ранеее, информация о структуре результирующего датафрейма может быть получена на каждом шаге обработки с помощью специальных служебных методов: df.schema или df.printSchema(). Схемы датафреймов из рассмотренного скрипта:

```
followersDf.printSchema()
```

```
root
 |-- follower: long (nullable = true)
 |-- key: string (nullable = true)
 |-- profile: long (nullable = true)
```

followersDf

```
followingDf.printSchema()
```

```
root
 |-- follower: long (nullable = true)
 |-- following_count: long (nullable = false)
```

followingDf

Важной составляющей обработки с помощью DataFrame API является работа с несколькими различными таблицами, принадлежащих одному или нескольким датасетам. В частности, это операции объединения данных и фильтрации одного датафрейма на основе данных из другого датафрейма. Рассмотрим пример из листинга, посвященный этим двум операциям. Фильтрация здесь осуществляется исключительно за счет использования операции join специального типа.

SparkSession

Фильтрация одного датафрейма за счет данных из другого с помощью операции **join.**

```
val traces = spark.read.parquet("/storage/users.parquet")
  .where("uid > 0")
val userWallPosts = spark.read.parquet("/storage/wall_posts.parquet")
```

Оставляем только одну колонку, состоящую из уникальных значений

```
val uids = traces.select("uid").distinct()
```

Фильтруем первую таблицу (левую) с помощью 2-ой, используя left_semi join. Только записи левой таблицы, имеющие соответствующие правой таблице. owner_id останутся. К*олонки правой таблицы не будут присутствовать в результирующей таблице*

```
userWallPosts.join(
    uids,
    userWallPosts.col("owner_id") === userWallPosts.col("uid"),
    "left_semi"
 )
 .write.parquet("/storage/filtered_wall_posts.parquet")
```

Для начала получим explain обработки, вызвав соответствующий метод у результирующего датафрейма: userWallPostsDf.explain().

SparkSession

В следующем листинге приведен вывод этой команды. В корне дерева физического плана исполнения (а именно его выдает эта команда) можно видеть операцию SortMergeJoin. Эта операция описывает конкретный тип операции join, который будет применен в данном случае. В данном случае операция будет выполнена с помощью Sort Shuffle.

```
== Physical Plan ==
SortMergeJoin [owner_id#818L], [uid#0L], LeftSemi
:- Sort [owner_id#818L ASC NULLS FIRST], false, 0
:   +- Exchange hashpartitioning(owner_id#818L, 2000)
:      +- Project [...]
:         +- Filter isnotnull(owner_id#818L)
:                                    +- FileScan parquet […] Batched: false, Format: Parquet, Location:
InMemoryFileIndex[/storage/wall_posts.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(owner_id)],
ReadSchema:
struct<attachments:array<struct<album:struct<created:bigint,description:string,id:string,owner_id...
+- *Sort [uid#0L ASC NULLS FIRST], false, 0
  +- Exchange hashpartitioning(uid#0L, 2000)
     +- InMemoryTableScan [uid#0L]
           +- InMemoryRelation [uid#0L], true, 10000, StorageLevel(disk, memory, deserialized, 1 replicas)
                 +- *Project [uid#0L]
                    +- *Filter (isnotnull(uid#0L) && (uid#0L > 0))
                       +-      *FileScan parquet [uid#0L] Batched: true, Format: Parquet, Location:
InMemoryFileIndex[/storage/users.parquet, PartitionFilters: [], PushedFilters: [IsNotNull(uid), GreaterThan(uid,0)],
ReadSchema: struct<uid:bigint>
```

SparkSession

В случае, если один датасет существенно больше другого, как в приведенном в примере, выполнение join может быть оптимизировано за счет применения другого типа этой операции – BroadcastHashJoin. В этом случае меньший датасет (если он достаточно маленький, что определяется параметром spark.sql.autoBroadcastJoinThreshold) будет целиком собран с узлов в драйвер приложения Spark, для него будет построена HashMap структура данных (ключ – кортеж по значениями колонок, по которым происходит объединение, значение – соответствующая запись из малого датасета), и ее копия однократно будет разослана на все имеющиеся executor'ы. Объединенный датафрейм будет получен путем фильтрации большего датасета по ключам из этой HashMap и объединения с записями из нее в случае совпадения ключей. Последнее необходимо для того, чтобы Spark оценил размеры обоих датасетов и убедился, что один из них подпадает под ограничение spark.sql.autoBroadcastJoinThreshold, что позволит ему заменить тип join в физическом плане выполнения.

Изменение типа join за счет получения дополнительной информации через материализацию одного из датафреймов.

```
val uids = traces.select("uid").distinct().cache()
uids.count()

userWallPosts
   .join(uids,
    userWallPosts.col("owner_id") === userWallPosts.col("uid"),
    "left_semi"
   )
    .write.parquet("/storage/filtered_wall_posts.parquet")
```

SparkSession

Физический план для модифицированной программы из предыдущего листинга:

```
== Physical Plan ==
BroadcastHashJoin [owner_id#818L], [uid#0L], LeftSemi, BuildRight
:- Project [attachments#805, can_delete#806L, collected_timestamp#807, comments#808, date#809L,
final_post#810L, geo#812, id#813L, is_pinned#814L, key#815, likes#816, marked_as_ads#817L,
owner_id#818L, post_source#819, reply_owner_id#821L, reply_post_id#822L, reposts#823,
signer_id#824L, text#825, is_reposted#826, repost_info#827]
: +- Filter isnotnull(owner_id#818L)
:    +-
FileScan
[attachments#805,can_delete#806L,collected_timestamp#807,comments#808,date#809L,final_post#810L
,from_id#811L,geo#812,pinned#814L,key#815,likes#816,marked_as_ads#817L,owner_id#818L,post_so
urce#819,post_type#820,reply_owner_id#821L,822L,reposts#823,signer_id#824L,text#825,is_reposted#
826,repost_info#827] Batched: false, Format: Parquet, InMemoryFileIndex[/storage/wall_posts.parquet],
PartitionFilters: [], PushedFilters: [IsNotNull(owner_id)],
struct<attachments:array<struct<album:struct<created:bigint,description:string,id:string,owner_id...
+- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, true]))
  +- InMemoryTableScan [uid#0L]
       +- InMemoryRelation [uid#0L], true, 10000, StorageLevel(disk, memory, deserialized, 1 replicas)
            +- *Project [uid#0L]
               +- *Filter (isnotnull(uid#0L) && (uid#0L > 0))
                  +- *FileScan parquet [uid#0L] Batched: true, Format: Parquet, Location:
InMemoryFileIndex[/storage/users.parquet, PartitionFilters: [], PushedFilters: [IsNotNull(uid),
GreaterThan(uid,0)], ReadSchema: struct<uid:bigint>
```

Использование пользовательских функций (UDF)

Чтение файлов с данными, нужными для обработки (данные существенного объема)

```
with open(filename, "r", encoding="utf-8") as f:
    phrases = f.readlines()
```

Создание broadcast переменных.
в этот момент реальные данные рассылаются по executor'ам

```
ses}
ases)}
file_to_met
for ph, i in phrases.items()}
```

```
phrasesBcs = spark.sparkContext.broadcast(phrases)
iphToNameBcs = spark.sparkContext.broadcast(iph_to_name)
```

Функции обработки данных, определенные пользователем
пишутся на языках поддерживаемых Spark (Scala, Java, Python, R) и могут использовать любые библиотеки из экосистемы языка. Функция может использовать больше чем одну колонку
из DataFrame, к которому она будет применена. Важен порядок в котором ей будут передаваться колонки в момент применения

```
def correct_text(is_reposted, text, repost_info):
  return repost_info.orig_text if is_reposted else text
```

```
def check_appearance(text):
  phrasesDict = phrasesBcs.value
  foundPhrases = [i for ph, i in phrasesDict.items() if ph in text.lower()]-
  return foundPhrases

def calculate_appearance(found_phrases):
  iph_to_name_dict = iphToNameBcs.value
  calcOfAppearance = {i: 0 for i, name in iph_to_name_dict.items()}
  for ph_in_post in found_phrases:
      for iph in ph_in_post: calcOfAppearance[iph] += 1
  return {iph_to_name_dict[iph]: count for iph, count in calcOfAppearance.items()}
```

Эти 2 функции используют ранее созданные broadcast-переменные для обработки. Для этого созданные переменные "замыкаются" в функции обработки.
Это возможно благодаря тому, что broadcast переменная не содержит сами данные, а только идентификатор,
по которому их можно будет получить во время выполнения на executor

Использование пользовательских функций (UDF)

```
correctText = udf(correct_text, returnType=StringType())
checkAppearance = udf(check_appearance, returnType=ArrayType(IntegerType()))
calculateAppearance = udf(
    calculate_appearance,
    returnType=MapType(StringType(), IntegerType())
)
```

Создание UDF-ов путем регистрации их в драйвере их сериализованное представление будет рассылаться вместе с task'ами на executor'ы необходимо указывать возвращаемый тип данных

Применение UDF - функций для трансформации DataFrame'ов

```
userWallCheckedPhrasesDf = userWallPostsDf \
    .select("owner_id", correctText("is_reposted", "text", "repost_info").name("text"))

userWallCheckedPhrasesDf = userWallCheckedPhrasesDf \
    .select("owner_id", checkAppearance("text").name(PH_CHECK_COL))

userWallCheckedPhrasesDf.write.parquet("/storage/wall_phrases.parquet")
```

Пользовательские функции агрегации

В Spark имеется возможность задавать пользовательские функции агрегации. В этом случае необходимо унаследовать объект типа UserDefinedAggregateFunction и реализовать следующие методы:

- bufferSchema;
- datatype;
- deterministic;
- initialize;
- update;
- merge;
- evaluate;

После этого нужно будет создать экземпляр объекта пользовательской функции агрегации, зарегистрировать его через spark.udf.register и затем использовать в функции agg из стандартной библиотеки. Пример показан в листинге…

Пользовательские функции агрегации

```
import org.apache.spark.sql.expressions.MutableAggregationBuffer
import org.apache.spark.sql.expressions.UserDefinedAggregateFunction
import org.apache.spark.sql.Row
import org.apache.spark.sql.types._
class GeometricMean extends UserDefinedAggregateFunction {
```

Входные типы

```
override def inputSchema: org.apache.spark.sql.types.StructType =
    StructType(StructField("value", DoubleType) :: Nil)
```

Внутренние типы для агрегации

```
override def bufferSchema: StructType = StructType(
    StructField("count", LongType) :: StructField("product", DoubleType) :: Nil
)
```

Выходной тип

```
override def dataType: DataType = DoubleType
 override def deterministic: Boolean = true
```

Начальные значения для буффера

```
 override def initialize(buffer: MutableAggregationBuffer) {
   buffer(0) = 0L
   buffer(1) = 1.0
 }
```

Обновление буфера

```
 override def update(buffer: MutableAggregationBuffer, input: Row): Unit = {
   buffer(0) = buffer.getAs[Long](0) + 1
   buffer(1) = buffer.getAs[Double](1) * input.getAs[Double](0)
 }
```

Пользовательские функции агрегации

Слияние

```
override def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
  buffer1(0) = buffer1.getAs[Long](0) + buffer2.getAs[Long](0)
  buffer1(1) = buffer1.getAs[Double](1) * buffer2.getAs[Double](1)
}
```

Окончательное значение, по окончательному значению из буфера

```
override def evaluate(buffer: Row): Double = {
  math.pow(buffer.getDouble(1), 1.toDouble / buffer.getLong(0))
 }
```

Создание инстанса UDAF

```
 val gm = new GeometricMean
```

Регистрация UDAF

```
spark.udf.register("gm", gm)

userWallPosts.groupBy("group_id")
  .agg(gm($"id").as("GeometricMean"))
   .show()
```

Облачные платформы и Apache Spark

Для обработки данных в Big Data очень важна горизонтальная масштабируемость. Чем больше узлов в кластере, тем быстрее обрабатываются данные. И Spark не исключение. Строить кластеры для обработки данных на своем оборудовании сложно и невыгодно. Заранее тяжело угадать, сколько нужно серверов и какой они должны быть мощности. Если использовать слишком много серверов, они будут простаивать. Если слишком мало, то обработка данных будет занимать много времени. Поэтому стоит обратить внимание на облачные технологии.

Облачные платформы могут предоставить огромное количество ресурсов по требованию. Если у вас небольшая нагрузка — подключаете несколько небольших узлов. Когда нагрузка возрастет, можно быстро добавить новые узлы в кластер. Это позволит использовать ресурсы эффективнее.

Кроме того, облачные платформы предоставляют и другие инструменты для работы с большими данными. Например, на платформе VK Cloud (бывш. MCS) есть разные инструменты для работы с Big Data: Spark, Hadoop, Kafka, Storm. Эти инструменты предоставляются «как сервис», т. е. их не нужно устанавливать и поддерживать, этим занимается облачный провайдер.

Кроме очевидных инструментов для работы с большими данными, облачные платформы предоставляют и другие полезные технологии. Например, Apache Spark можно развернуть в Kubernetes, чтобы получить больше гибкости и решить несколько проблем классического Hadoop-кластера — вроде изоляции сред или разделения Storage- и Compute-слоев.

Заключение

Apache Spark – фреймворк для обработки данных в Big Data. Он работает в оперативной памяти и редко обращается к диску, поэтому обрабатывает данные очень быстро.

Раньше стандартом де-факто для обработки данных был Hadoop MapReduce. Но у него есть две ключевые проблемы: низкая производительность и высокая сложность разработки.

Сейчас стандартом практически стал Spark. Он создавался, чтобы устранить недостатки MapReduce и сохранить его преимущества.

Apache Spark и другие технологии для работы с большими данными активно используются в облаках, потому что позволяют получить все преимущества облачных сервисов.

СПАСИБО ЗА ВНИМАНИЕ!