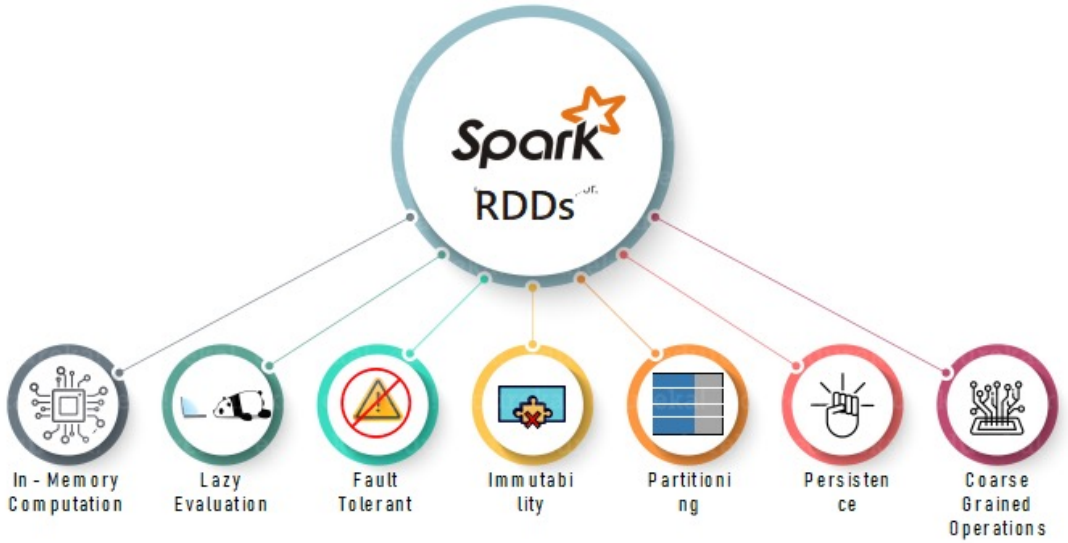
Spark
RDDs
In - Memory Computation
Lazy Evaluation
Fault Tolerant
Immutability
Partitioning
Persistence
Coarse Grained Operations

Spark RDD

О чем поговорим

1) Общая информация про RDD
2) Операции над RDD
3) Shuffle operations
4) StorageLevel

RDD

Абстракция RDD – Resilient Distributed Dataset. Это основа Spark. Кстати, RDD является неизменяемым объектом!

- Восстанавливаемый распределенный набор данных
- Входы операций должны быть RDD
- Все промежуточные операции также RDD
- Так как известна цепочка вычислений DAG, то потерянные части восстанавливаются из входных данных

Как создать RDD

- SparkContext.parallelize – создать RDD из объекта в памяти
- SparkContext.textFile – создать RDD из текстового файла
- SparkContext.wholeTextFiles – создать RDD из файлов в папке

Операции над RDD

- Трансформации (RDD -> RDD)

Все трансформации являются ленивыми вычислениями, то есть мы вычисляем тогда, когда нужен результат. Например операции map, reduceByKey, join.

- Действия

Действия приводят к запуску DAG для расчета RDD, отдавая результат. Пример: saveAsTextFile, collect, count, write.

- Иные операции, такие как persist, cache. Они заставляют Spark сохранить RDD в памяти для получения быстрого результата.

Как менять количество партиций

Количество партиций задается в контексте через spark.default.parallelism. По умолчанию берется максимум между 2 и количеством ядер (в YARN).

Методы для смены количества партиций в RDD:

- repartiion() – может как увеличивать, так и уменьшать количество партиций

- coalesce() – может только снижать количество партиций

repartition()всегда приводит к равномерному перераспределению данных, что ведет к shuffle. Если Вы уменьшаете число партиций, то стоит использоватьcoalesce(), который может избежать shuffle.
Например, если данные лежат на 4 нодах, то снижение количества партиций до 2 затронет в случае repartition приведет с shuffle всех данных, а coalesce затронет только 2 ноды.

Немного про transformations

- Трансформации не модифицируют исходных RDD, они возвращают новый объект, в основном RDD.
- map и flatMap отличаются тем, что применяя map к списку списков мы в ответ также получим список списков, а flatMap их размажет в один список, как flatten в нейросетках
- Некоторые операции могут приводить к shuffle данных и в случае неаккуратного использования привести в OOM
- Операция сложения двух rdd – это их объединение

Shuffle operations

Shuffle нужен для перераспределения данных между исполнителями\машинами.

К shuffle приводят следующие операции:

- groupByKey
- reduceByKey
- Join

Это дорогие операции из-за:

- Disk I/O
- Сериализация и десериализация данных
- Network I/O

Shuffle operations

К shuffle также приводит repartition, coalesce и cogroup.
countByKey – операция, не приводящая к shuffle.

Лучше использовать reduceByKey вместо groupByKey:

- groupByKey сначала на разных исполнителях\машинах распределит данные по ключам, делая полный shuffle
- reduceByKey сначала сделает агрегацию внутри партиций, а уже после shuffle

Как посмотреть данные

- take - пытается минимизировать число обращений к партициям, поэтому может возвращать смещенные результаты, стоит использовать просто «на посмотреть»
- collect – сложит все в оперативную память, стоит быть осторожнее
- takeSample - Если нужно получить небольшое число записей на драйвер и, при этом, сохранить распределение

RDD persistence

Используя методы cache() и persist(), PySpark предоставляет механизм оптимизации для хранения промежуточных вычислений RDD, чтобы их можно было повторно использовать в последующих действиях.

Метод PySpark RDD cache() по умолчанию сохраняет вычисления RDD на уровне хранения MEMORY_ONLY, что означает, что он будет хранить данные в памяти JVM в виде несериализованных объектов.

Метод PySpark cache() в классе RDD внутренне вызывает метод persist(), который, в свою очередь, использует spark Session.shared State.CacheManager.cachequery для кэширования результирующего набора RDD.

PySpark persist() используется для хранения результатов RDD на одном из уровней хранения.

Unpersist

PySpark автоматически отслеживает все выполняемые вызовы persist() и cache(), проверяет использование на каждом узле и удаляет сохраненные данные, если они не используются или с помощью алгоритма наименее недавно использованного (LRU).

Вы также можете удалить вручную, используя метод unpersist(). unpersist() помечает RDD как non-persistent и удаляет все блоки для него из памяти и с диска.

pyspark.StorageLevel

MEMORY_ONLY – используется по умолчанию для метода RDD cache(), который сохраняет RDD в виде десериализованных объектов в памяти JVM. Когда недостаточно доступной памяти, некоторые разделы RDD не будут сохранены, они будут пересчитаны по мере необходимости. Это занимает больше места, но работает быстрее, так как для чтения из памяти требуется несколько циклов процессора.

MEMORY_ONLY_SER - это то же самое, что и MEMORY_ONLY, но разница в том, что он хранит RDD в виде сериализованных объектов в памяти JVM. Это занимает меньше памяти (экономит место), чем MEMORY_ONLY, так как сохраняет объекты как сериализованные, но требует дополнительных нескольких циклов процессора для десериализации.

MEMORY_ONLY_2, MEMORY_ONLY_SER_2 – аналогично предыдущим, но делают реплики партиций на 2 ноды.

pyspark.StorageLevel

MEMORY_AND_DISK - на этом уровне хранения RDD будет храниться в памяти JVM в виде десериализованных объектов. Когда объем требуемого хранилища превышает объем доступной памяти, он сохраняет некоторые лишние разделы на диске и считывает данные с диска, когда это требуется. Это происходит медленнее, так как задействован ввод-вывод.

MEMORY_AND_DISK_SER – аналогично MEMORY_AND_DISK, но все объекты будут храниться в сериализованном виде. Меньше тратим памяти, но больше времени на десериализацию.

MEMORY_AND_DISK_2, MEMORY_AND_DISK_SER_2 – аналогично двум пунктам выше + репликация партиций на 2 ноды

pyspark.StorageLevel

DISK_ONLY - на этом уровне хранения RDD хранится только на диске, а время вычислений CPU сильно возрастает, поскольку задействован ввод-вывод

DISK_ONLY_2 – все на диске + репликация партиций на 2 ноды

dfPersist = rdd.persist(pyspark.StorageLevel. DISK_ONLY)

Broadcast

Broadcast variables - это общие переменные только для чтения, которые кэшируются и доступны на всех узлах кластера для доступа или использования задачами. Вместо того чтобы отправлять эти данные вместе с каждой задачей, PySpark распределяет переменные на машины, используя эффективные алгоритмы для снижения затрат на связь.

Полезно делать broadcast если у вас есть список стоп-слов, счетчик для подсчета вхождени элементов в заданный список, для join операций, если один RDD небольшого размера.

```
broadcastVar = sc.broadcast([0, 1, 2, 3])
```

Accumulators

Аккумуляторы PySpark - это еще одна общая переменная, которая “добавляется” только с помощью ассоциативной и коммутативной операции и используется для выполнения счетчиков (аналогично счетчикам уменьшения карты) или операций суммирования.

PySpark по умолчанию поддерживает создание аккумулятора любого числового типа и предоставляет возможность добавлять пользовательские типы аккумуляторов.

Типы аккумуляторов:

- именованные аккумуляторы – отображается в Spark-UI в виде таблицы и можно отслеживать результаты
- безымянные аккумуляторы – не отображаются

Рекомендуется использовать именованные аккумуляторы

```
accum = sc.longAccumulator("SumAccumulator")
sc.parallelize([1, 2, 3]).foreach(lambda x: accum.add(x))
```

Accumulators

Аккумуляторы по типам данных:

- Long Accumulator
- Double Accumulator
- Collection Accumulator