Week4: Advanced topics on Apache Spark @Sieun Bae

Execution & Scheduling

Spark Context: 스파크가 동작하기위한 각종 백엔드 서비스에 대한 참조를 가지고 있는 객체

✔ RDD를 생성하려면 SparkContext 객체를 먼저 생성하고, DataFrame 또는 DataSet을 생성하려면 SparkSession 객체를 먼저 생성해야 한다. 하지만 정확하게 말하면 스파크세션 객체 안에는 스파크컨텍스트 객체가 포함돼 있기 때문에 RDD를 만들때나 데이터프레임을 만들때나 상관없이 스파크세션 객체를 먼저 생성하면 된다.

```
// spark는 스파크세션 객체
val input = List("a", "b", "c")
val rdd = spark.sparkContext.parallelize(input)
```

스파크세션 클래스의 생성자 인자는 모두 4개이며 각각의 타입이 SparkContext, SharedState, SessionState, SparkSessionExetensions 이다. 즉, 스파크세션 객체를 만들려면 스파크컨텍스트를 비롯한 SessionState 객체가 먼저 생성돼 있어야 하며 실제로 어딘가에 이 부분을 처리하는 코드가 있다는 것을 의미한다. 이부분은 굳이 찾아보지 않더라도 스파크세션을 만드는 방법을 생각해보면 대충 감을 잡을 수 있다.

```
// 스파크세션 생성
val spark = SparkSession
        .builder()
        .appName("MyApp")
        .master("local[*]")
        .getOrCreate()
```

위 코드는 스파크세션을 생성하는 코드로, 필요한 설정값을 지정한 다음 맨 마지막에 getOrCreate를 호출하게 돼 있다. 바로 이 부분이 스파크컨텍스트를 포함한 나머지 객체를 생성하는 부분이며 이를 통해 새로운 스파크세션 객체를 생성할 수 있다.

스파크세션은 스파크컨텍스트에 세션 상태 정보를 추가로 담은 것이라고 할 수 있다.

스파크세션에 세션 상태 정보라는 것을 추가한 이유는 무엇일까? 스파크 컨텍스트는 스파크가 동작하기 위한 각종 백엔드 서비스에 대한 참조를 가지고 있는 객체라고 할 수 있다.
실제로 스파크컨텍스트 클래스는 statusTracker, dagScheduler, taskScheduler 등 백엔드 서비스들에 직접 접근할 수 있는 참조 변수를 가지고 있으며 RDD API를 이용해 필요한 연산을 수행할 경우 내부적으로 이러한 참조 변수를 직접 사용해 요청한 작업을 처리하게 된다.

따라서 스파크에서 동작하는 모든 애플리케이션은 백엔드 서버와 통신하기 위해 스파크컨텍스트 객체를 사용해야 하며 같은 이유로 스파크세션의 경우에도 스파크컨텍스트를 먼저 생성한 후 이에 대한 참조를 내부적으로 유지하고 있는 것이다.

스파크에서 사용자가 RDD와 같은 백엔드 서비스에 직접 접근 가능한 API를 사용하는 대신 RDD보다 한 단계 추상화된 API를 사용해 코드를 작성하게 하고, 이 코드를 내부적인 최적화 과정을 거쳐 실제 동작 가능한 RDD 기반의 코드로 전환하는 방법을 채택하게 됐는데, 이 새로운 API가 바로 데이터프레임과 데이터셋이라고 할 수 있다.

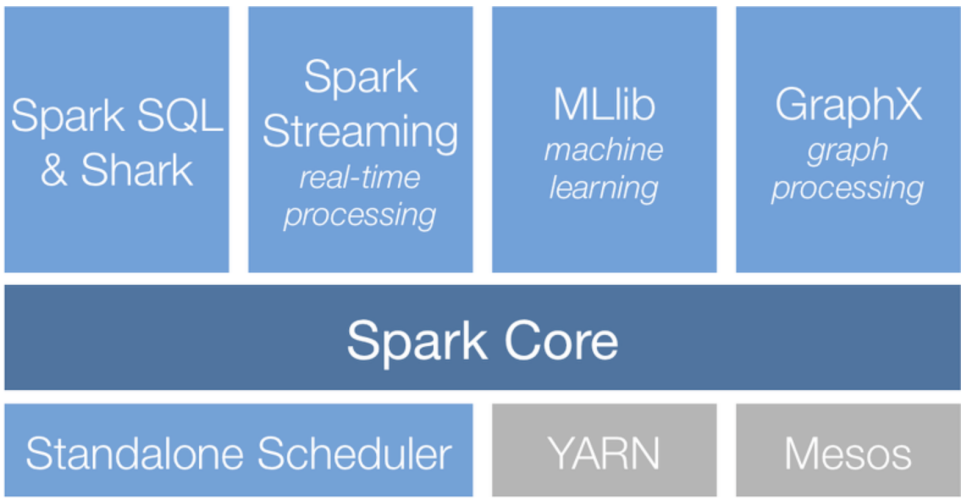
```
private[sql] class SessionState(
    sharedState: SharedState,
    val conf: SQLConf,
    val experimentalMethod: ExperimentalMethods,
    val functionRegistry: FunctionRegistry,
    val udfRegistration: UDFRegistration,
    catalogBuilder: () ⇒ SessionCatalog,
    val sqlParser: ParserInterface,
    analyzerBuilder: () ⇒ Analyzer,
    optimizerBuilder: () ⇒ Optimizer,
    val planner: SparkPlanner,
    val streamingQueryManager: StreamingQueryManager,
    val listenerManager: ExecutionListenerManager,
    resourceLoaderBuilder: () ⇒ SessionResourceLoader,
    createQueryExecution: LogicalPlan ⇒ QueryExecution,
    createClone: (SparkSession, SessionState) ⇒ SessionState) {
    ...
}
```

데이터프레임이 사용자가 입력한 코드를 최적화해서 실제 동작 가능한 코드로 바꾼다 하였는데, Analyzer/Optimizer/SparkPlanner/QueryExecution 등이 이와 관련된 역할을 수행하는 것들이다. 구체적으로는 아래와 같은 절차를 따르게 된다.

- 데이터프레임 연산을 통해 **LogicalPlan 생성**(QueryExecution의 logical로 조회되는 쿼리)
- 생성된 LogicalPlan을 SessionState의 **Analyzer**에 전달해서 미식별 정보에 대한 처리를 진행한 후 그 결과로 수정된 LogicalPlan을 생성

(QueryExecution의 analyzed로 조회되는 쿼리)
- Analyzer가 생성한 LogicalPlan을 SessionState의 **Optimizer**에 전달, 최적화 과정을 수행한 후 새로운 LogicalPlan을 생성(QueryExecution의 optimizedPlan으로 조회되는 쿼리)
- 최적화된 LogicalPlan을 SessionState의 **SparkPlanner**에 전달해서 SparkPlan 생성(QueryExecution의 sparkPlan으로 조회되는 쿼리)
- 생성된 SparkPlan에 추가적인 최적화 과정을 적용한 후 **최종 SparkPlan 생성**(QueryExecution의 executedPlan으로 조회되는 쿼리)

쿼리 실행 계획은 내부적으로 이 같은 최적화 단계를 거쳐서 생성된 것이며 이러한 최적화 기능이 데이터프레임 내부에서 이뤄지고 있음을 확인할 수 있다.
QueryExecution 객체를 이용하면 런타임에 생성된 코드 정보와 최적화에 사용된 각종 클래스 정보 등도 확인할 수 있으므로 실제 업무를 수행하는 과정에서는 최신 API를 참고해서 이러한 정보들을 잘 활용하것도 내부 구조를 이해하고 성능을 최적화하는데 도움이 될것이다.
출처: https://12bme.tistory.com/432 [길은 가면, 뒤에 있다.]



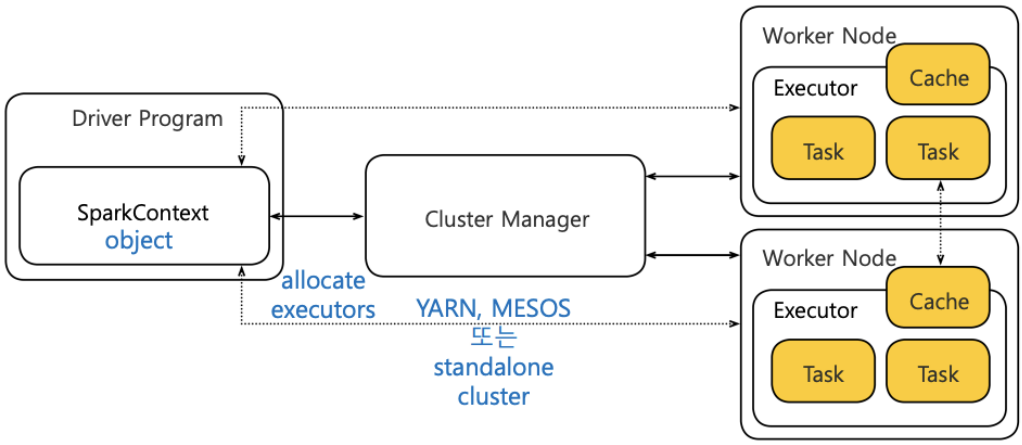
- 인프라 계층
  - Standalone Scheduler(기본)
  - Yarn(하둡)
  - Mesos(Docker 또는 가상화 플랫폼)

- 스파크 코어
  - 메모리 기반의 분산 클러스터 컴퓨팅

- 스파크 라이브러리
  - 빅데이터를 SQL로 핸들링하는 Spark SQL
  - 실시간 스트리밍 Spark Streaming
  - 머신러닝 MLlib
  - 그래프 데이터 프로세싱 GraphX

SparkContext

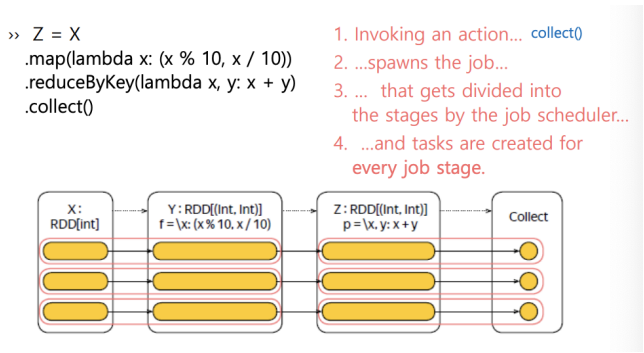
›› Tells your application how to access a cluster
›› Coordinates processes on the cluster to run your application



spawn: 멀티태스킹이 지원되는 운영체계에서 하위 프로세스를 만드는 것

Jobs, Stages, Tasks

- task is a unit of work to be done
- tasks are created by a job scheduler for every job stage
- job is spawned in response to a Spark action
- job is divided in smaller sets of tasks called stages



job stage: for pipelining, RDD level

vs

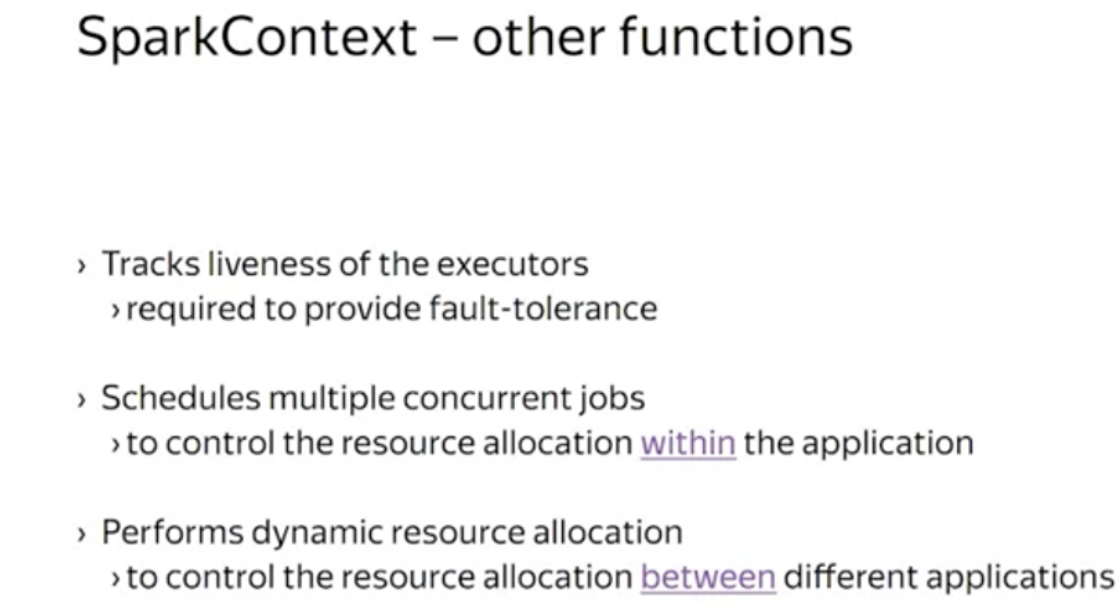
task: partition level

› Job stage is a pipelined computation spanning between materialization boundaries
›not immediately executable

› Task is a job stage bound to particular partitions
›immediately executable

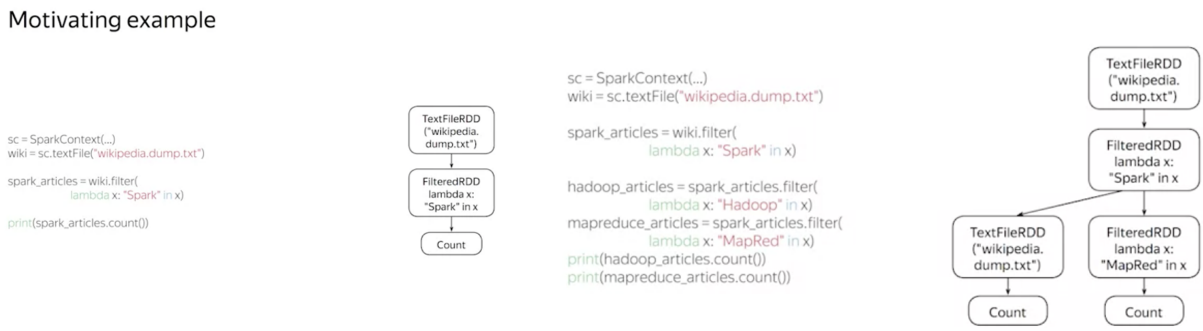
Materialization(==building) happens when reading, shuffling or passing data to an action

- narrow dependencies allow pipelining
- wide dependencies forbid it



Caching & Persistence: Intermediate Data

- RDDs are partitioned
- Execution is build around the partitions
- **Block** is a unit of input and output in Spark

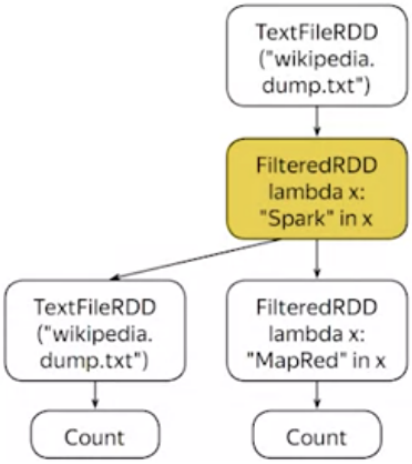


→ 데이터 로딩을 반복 수행하게됨.. 매우 비효율적

⇒ 캐싱을 통해 해결

```
sc = SparkContext(...)
wiki = sc.textFile("wikipedia.dump.txt")

spark_articles = wiki.filter(
                 lambda x: "Spark" in x)
spark_articles.cache()
hadoop_articles = spark_articles.filter(
                 lambda x: "Hadoop" in x)
mapreduce_articles = spark_articles.filter(
                 lambda x: "MapRed" in x)
print(hadoop_articles.count())
print(mapreduce_articles.count())
```



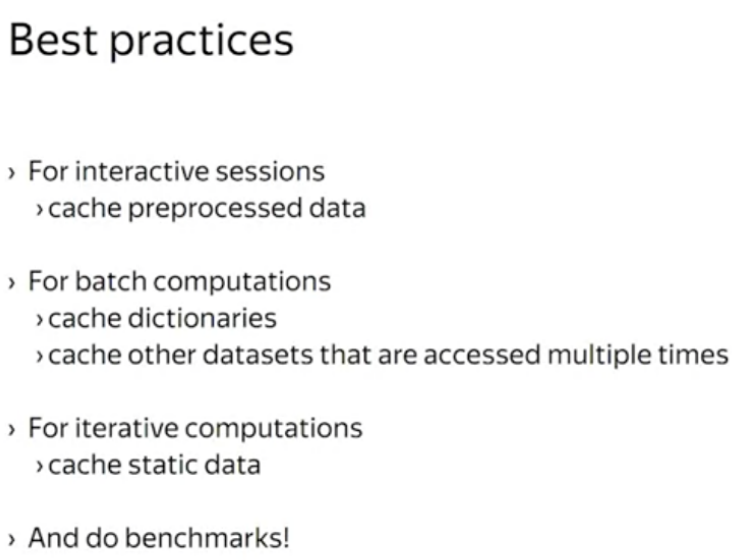
⇒ spark_articles.cache()

Controlling persistence level

› rdd.persist(storageLevel)
  › sets RDD's storage to persist across operations after it is computed for the first time
  › storageLevel is a set of flags controlling the persistence, typical values are
  DISK_ONLY
    – save the data to the disk,
  MEMORY_ONLY
    – keep the data in the memory
  MEMORY_AND_DISK
    – keep the data in the memory; when out of memory – save it to the disk
  DISK_ONLY_2, MEMORY_ONLY_2, MEMORY_AND_DISK_2
    – same as about, but make two replicas
› rdd.cache() = rdd.persist(MEMORY_ONLY)

⇒ cache 함수는 메모리에 저장한다는 함수의 shortcut임

언제 어떻게 조절해야할까?

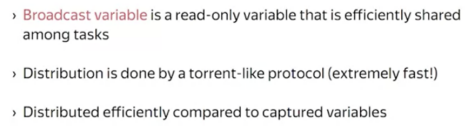


Summary

- performance may be imporved by persisting data across operations
  - in interactive sessions, iterative computations and hot datasets
- you can control the persistence of a dataset
  - whether to store in the memory or on the disk
  - how many replicas to create

Broadcast Variables: shared data

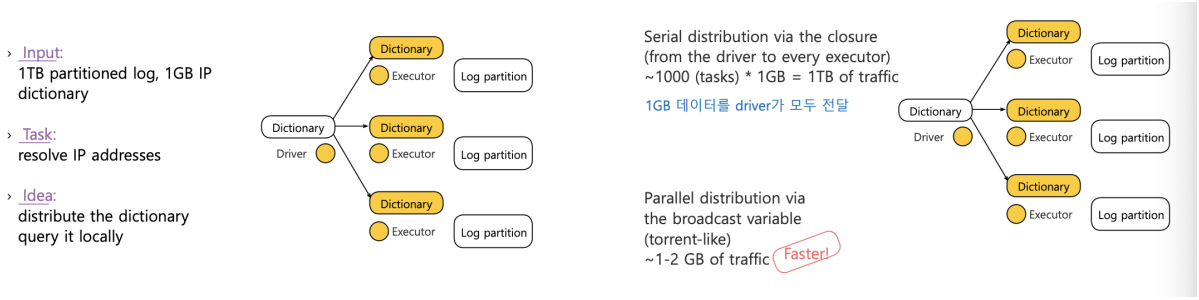
분산된 큰 값들에 사용, read-only



→ 일반 variable을 closure에 넣으면 1—many protocol (한 곳에서 많은 executor로 전달해야 함)

→ broadcast variable은 many—many protocol (aka 토렌트)

join? mapside join



Motivating example 2

```
sc = SparkContext(conf=...)

# compute the dictionary
my_dict_rdd = sc.textFile(...).map(...).filter(...)
my_dict_data = my_dict_rdd.collect()

# distributed the dictionary via the broadcast variable
broadcast_var = sc.broadcast(my_dict_data)

# use the broadcast variable within the task
my_data_rdd = sc.textFile(...).filter(
        lambda x: x in broadcast_var.value)
```

```
sc = SparkContext(conf=...)

# compute the dictionary
my_dict_rdd = sc.textFile(...).map(...).filter(...)
my_dict_data = my_dict_rdd.collect()

# distributed the dictionary via the broadcast variable
broadcast_var = sc.broadcast(my_dict_data)

# use the broadcast variable within the task
my_data_rdd = sc.textFile(...).filter(
        lambda x: x in broadcast_var.value)
```

Summary

- broadcast variables are read-only shared variables with effective sharing mechanism
  - 단, memory에 맞는 양의 데이터 활용 가능
- useful to share dictionaries, models

Accumulator Variables

특정 컬렉션의 정보를 합산할 때 사용, read-write

› Accumulator variable is a read-write variable that is shared among tasks

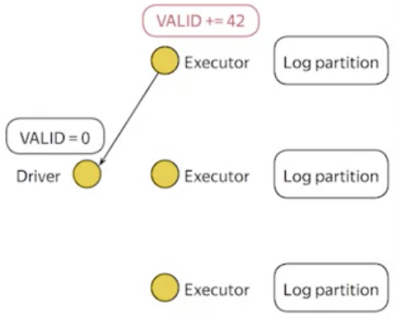
› Writes are restricted to increments!
  › i. e.: var += delta
  › addition may be replaced by any associate, commutative operation

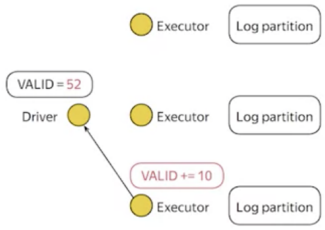
› Reads are allowed only by the driver program!

only by adding ⇒ 동기화 이슈를 피하기 위해, read only by the driver program

› Input:
1TB partitioned log

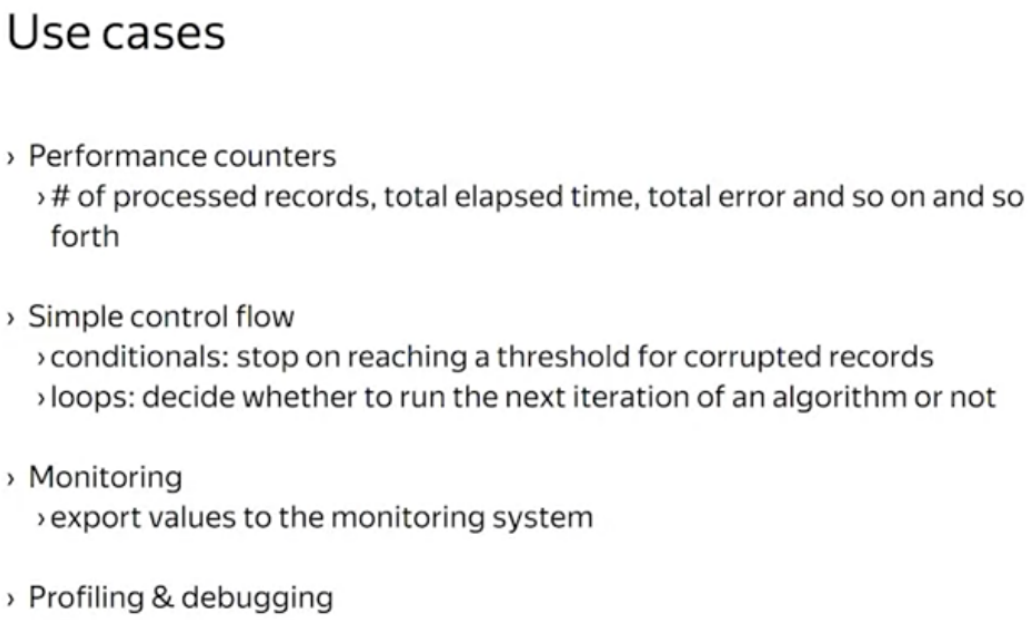
› Task:
resolve IP addresses
AND
collect metrics:
# of valid records





› In actions updates are applied exactly once

› In transformations there are no guarantees as the transformation code may be re-executed



Summary

- Accumulators are read-write shared variables with restricted updates
  - increments only
  - can use custom associative, commutative operation for the updates
  - can read the total value only in the driver
- Useful for the control flow, monitoring, profiling & debugging

아파치 스파크 #6 (고급 스파크 프로그래밍)

스파크는 두가지 다른 타입의 공유 변수를 갖는다 1. broadcast 변수 : 분산된 큰 값들에 사용 2. accumulators 변수 : 특정 컬랙션의 정보를 합산할 때 사용 Broadcast Variables 이는 프로그래머들이 각 머

https://ryuk.tistory.com/134#recentComments

Quiz 2

Week4: Working with Spark in Python