w4

16. Structured and Optimization

- 이번 주 주제인 Spark SQL의 동기인 structure와 optimization에 대해 알아보자.

* 학생에게 장학금을 인구통계정보와 재정상태 아래의 데이터 셋(pair RDDs)을 이용해 지급하려고 한다.
- 장학금의 조건이 스위스사람이고, 빚과 부양할 사람이 있는 것이라면, 다음과 같은 여러 방법으로 구할 수 있다.

```
case class Demographic(id: Int,
                       age: Int,
                       codingBootcamp: Boolean,
                       country: String,
                       gender: String,
                       isEthnicMinority: Boolean,
                       servedInMilitary: Boolean)
val demographics = sc.textfile(...)... // Pair RDD, (id, demographic)

case class Finances(id: Int,
                    hasDebt: Boolean,
                    hasFinancialDependents: Boolean,
                    hasStudentLoans: Boolean,
                    income: Int)
val finances = sc.textfile(...)... // Pair RDD, (id, finances)
```

a.
```
demographics.join(finances)
            .filter { p =>
              p._2._1.country == "Switzerland" &&
              p._2._2.hasFinancialDependents &&
              p._2._2.hasDebt
            }.count
```

Inner join → Filter to select people in 스위스 → Filter with debt & financial depend. for selection.

b. Filter down with debt & financial depend.
→ Filter to select people in 스위스
→ Inner join on filter downed dataset

```
val filtered
  = finances.filter(p => p._2.hasFinancialDependents && p._2.hasDebt)

demographics.filter(p => p._2.country == "Switzerland")
            .join(filtered)
            .count
```

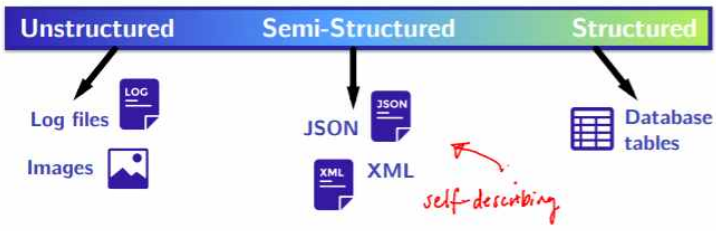
c. Cartesian prod. on both datasets →
filter to select with same IDs →
filter with debt & financial depend. and country for selection

```
val cartesian                    finances
  = demographics.cartesian(demographics)

cartesian.filter {
  case (p1, p2) => p1._1 == p2._1
}
.filter {
  case (p1, p2) => (p1._2.country == "Switzerland") &&
                   (p2._2.hasFinancialDependents) &&
                   (p2._2.hasDebt)
}.count
```

- 위 3 방법이 결과는 같지만 걸리는 시간이 다르다.
b는 a보다 3.6배 빠르다. 그리고 c는 거의 177배 느리다.
- 그래서 스파크가 필터링을 하고 조인하는게 카테시안 곱을 하고 필터링하는 것 보다 빠르다는 것을 자동적으로 알 수 있는게 좋지 않겠냐.
- **그리고 한 단계 나아가 structural information을 더 주면 스파크가 optimization을 수행할 수 있게 한다.**

* Structured vs. Unstructured Data
- Structured는 db 테이블과 같이 엄격한 스키마 구조를 갖고 있다.
- semi는 self describing이라고도 하며 자신의 구조를 자기가 설명하고 structured에 비해 엄격하진 않다.



- RDDs는 Unstructured와 Semi에 focus해 계산을 했지만 지금까지 배운 스칼라나 RDDs는 데이터의 스키마를 어떻게 해야하는지 알지 못한다. 스파크가 아는 것은 주어진 RDD에 대해 person, account, demographic(인구통계)와 같은 임의의 type으로 parameterized 되어있다는 것만 알고 RDD 이름만 아는 것이다. 예를 들어 person이라는 RDD에 name이라는 것이 string인지 신경을 안쓴다.

* Structured vs. RDDs

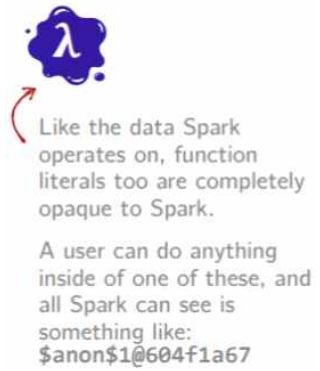
a. case class Account(name: String, balance: Double, risk: Boolean)와 같은 데이터셋에서

- spark/RDD가 보는 것은
  이를 읽으면 읽으면 Account라고 불리는 거
  밖에 모르고 안에 구조를 통해 어떻게 이를 통해
  어떻게 최적화할 수 있는지 모르고 opague(불투명)하다.

- DB/Hive는
  안의 구조를 알아 필요한 datatype을 정하고 클러스터에
  보낼 수 있다.

b. Computation할 때

- 스파크에서 우리는 지금까지, 데이터에서 **functional transformations**을 한 것이다.
  그리고 map, flatmap, filter와 같은 high-order 함수를 통해 user-defined function을
  사용했다. 이 또한 opaque하며 스파크는 어떤 operation을 하려는지 몰라 최적화할
  수 없다.



- DB/Hive에서는 데이터에서 **declarative transformation**을 한다. D.T는 굉장히 엄격하고
  미리 정의된 연산들이여서 어떤 연산을 하는지 알 수 있어 최적화가 가능하다.

* 비정형데이타에서의 RDDs 연산들은 이런 최적화가 제한적이지만
  Spark SQL은 최적화를 가능하게 한다.

17. Spark SQL

- 스파크가 인기를 많이 얻고 있지만, 여전히 SQL이 분석에 있어서 lingua franca(공통어)이다.
- 스파크나 하둡에서 SQL을 연결해 빅데이터 연산을 수행하는 것은 어렵지만, Spark SQL이 이를
  seamless하게 가능하게 한다.

* Spark SQL goals (3 mainly)

a. Support relational processing both within Spark programs (on RDDs) and on external data sources with a friendly API. (어떤 경우에는 SQL로 연산하는 것이 더 좋고, 반대로 functional API가 더 좋은 경우도 있다.)

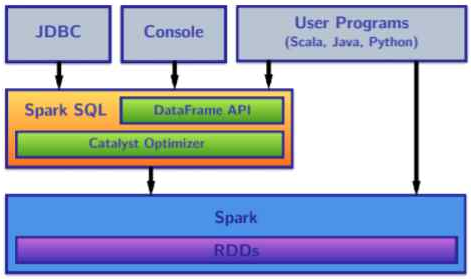
b. High performance, achieved by using techniques from research in DBs.

c. Easily support new data source such as semi-structured data and external DBs.

- Spark SQL은 새로 배워야하는 다른 시스템이 아니다. 스파크에서 돌아가는 일종의 라이브러리로 봐도 된다.
- 3 main APIs: SQL literal syntax
  DataFrames: 데이터 타입이 없고, 쿼리만 써서 사용할 수 있는 자료구조.
  Datasets: RDDs와 DF의 장점을 합친 자료구조이다.
- 2 specialized backend components: Catalyst (query optimizer)
  Tungsten (off-heap serializer : 시리얼라이즈 된 정보를 off-heap에 jvm의 가비지컬렉션에서 벗어나기 위해)
  가비지컬렉션안에 있으면 성능이나 메모리 관리에 영향을 받는다.

cf) https://soul0.tistory.com/89
http://woowabros.github.io/experience/2017/10/17/java-serialize.html

- 콘솔이나 user program으로 spark sql의 DF API로 접근할 수 있는데 이렇게 하는 것은 Catalyst Optimizer를 통해 RDDs로 전달되어 바로 접근하는 것보다 더 빠르고 좋게 접근하게 해준다.
- 이렇게 스파크위에 SQL을 가능하게 한 것이고, SQL 자체가 Structural Query Language이므로 굉장히 neat하다.
  SQL에는 fixed된 data type과 fiexed operations가 있다.
  ex) Int, Long, String, etc / SELECT, WHERE, GROUP BY, etc.



- 수 십 년간 research와 industry에서 관계형 DB의 rigidness를 통해 성능을 높이는 연구를 하는데 집중했다.

* Dataframe: Spark SQL's core abstraction이다.
- RDDs와 다르게 known schema가 있고 DF는 untyped이고
  DF에 대한 Transformation은 untyped transformation이라고 한다. 뒤에 자세히 다룬다.
  ex) RDD[T] 같이 안하고 그냥 DataFrame 한다.

* SparkSession, to start using Spark SQL
- SparkSession은 Spark SQL에서의 SparkContext 이다.
  SparkContext를 SparkSession으로 바꿔 SQL을 사용하자.
  오른쪽이 세션을 키기위한 최소한의 필요조건이다.

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession
    .builder()
    .appName("My App")
    //.config("spark.some.config.option", "some-value")
    .getOrCreate()
```

* Creating DF.

a. From existing RDD. (Schema inferece 또는 explicit schema를 통해 할 수 있다.)

b. Reading in a specific data source from file. ex) Json

a-1. Creating DF from RDD by Schema inferece

Given pair RDD, RDD[(T1, T2, ... TN)], a DataFrame can be created with its schema automatically inferred by simply using the toDF method.

```
val tupleRDD = ... // Assume RDD[(Int, String String, String)]
val tupleDF = tupleRDD.toDF("id", "name", "city", "country") // column names
```

Note: if you use toDF without arguments, Spark will assign numbers as attributes (column names) to your DataFrame.

If you already have an RDD containing some kind of case class instance, then Spark can infer the attributes from the case class's fields.

```
case class Person(id: Int, name: String, city: String)
val peopleRDD = ... // Assume RDD[Person]
val peopleDF = peopleRDD.toDF
```

a-2. Creating DF from RDD by explicit schema

3단계: create RDD of Rows from original RDD

→ Create the schema represented by StructType matching the structure of Rows in the RDD created above.

→ Apply the schema to the RDD of Rows via createDataFrame method provided by SparkSession

Given:

```
case class Person(name: String, age: Int)
val peopleRdd = sc.textFile(...) // Assume RDD[Person]
```

```
// The schema is encoded in a string
val schemaString = "name age"

// Generate the schema based on the string of schema
val fields = schemaString.split(" ")
  .map(fieldName => StructField(fieldName, StringType, nullable = true))
val schema = StructType(fields)
```

```
// Convert records of the RDD (people) to Rows
val rowRDD = peopleRDD
  .map(_.split(","))
  .map(attributes => Row(attributes(0), attributes(1).trim))

// Apply the schema to the RDD
val peopleDF = spark.createDataFrame(rowRDD, schema)
```

b. Create DF by reading data source from file.

- read method 사용. readtextfile과 비슷하지만 더 좋다.

```
// 'spark' is the SparkSession object we created a few slides back
val df = spark.read.json("examples/src/main/resources/people.json")
```

Json, CSV, Parquet, JDBC 와 같은semi 나 structured data은 가능하며 읽을 수 있는 파일들 아래에서 확인 가능
http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameReader

* SQL Literals

- peopleDF를 생성했으면 임시 SQL view를 먼저 만들면서 FROM에 필요한 이름을 정한다.

```
// Register the DataFrame as a SQL temporary view
peopleDF.createOrReplaceTempView("people")
// This essentially gives a name to our DataFrame in SQL
// so we can refer to it in an SQL FROM statement
```

- 그러면 다음과 같이 SQL 구문을 사용할 수 있다.

```
// SQL literals can be passed to Spark SQL's sql method
val adultsDF
  = spark.sql("SELECT * FROM people WHERE age > 17")
```

- SQL 기본 구문 뿐만아니라 HiveQL도 가능하다.

WHERE, COUNT, HAVING, GROUP BY, ORDER BY, SORT BY, DISTINCT, JOIN,
(LEFT|RIGHT|FULL) OUTER JOIN, Subqueries (ex) SELECT col FROM (SELECT a+b AS col FROM t1) t2

- supported SQL syntax

https://docs.datastax.com/en/datastax_enterprise/4.6/datastax_enterprise/spark/sparkSqlSupportedSyntax.html

- HiveQL cheatsheet

https://hortonworks.com/blog/hive-cheat-sheet-for-sql-users

- Updated list of supported Hive features in Spark SQL

https://spark.apache.org/docs/latest/sql-programming-guide.html#supported-hive-features

ex) Employee DF에서 시드니에 있는 사람의 ID와 성을 ID순으로 정렬해 찾아보자.

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)

// DataFrame with schema defined in Employee case class
val employeeDF = sc.parallelize(...).toDF          registered "employees"

val sydneyEmployeesDF
  = spark.sql("""SELECT id, lname
                   FROM employees
                  WHERE city = "Sydney"
               ORDER BY id""")
```

18. DataFrames(1)

- SQL query 외에 Spark SQL이 갖고 있는 own API가 무엇인지 알아보자.

* Detour on DFs

- A relational API over Spark's RDDs
- Able to be automatically aggressively optimized
- Untyped !

* DF Data types

Basic Spark SQL Data Types:

Scala Type	SQL Type	Details
Byte	ByteType	1 byte signed integers (-128,127)
Short	ShortType	2 byte signed integers (-32768,32767)
Int	IntegerType	4 byte signed integers (-2147483648,2147483647)
Long	LongType	8 byte signed integers
java.math.BigDecimal	DecimalType	Arbitrary precision signed decimals
Float	FloatType	4 byte floating point number
Double	DoubleType	8 byte floating point number
Array[Byte]	BinaryType	Byte sequence values
Boolean	BooleanType	true/false
Boolean	BooleanType	true/false
java.sql.Timestamp	TimestampType	Date containing year, month, day, hour, minute, and second.
java.sql.Date	DateType	Date containing year, month, day.
String	StringType	Character string values (stored as UTF8)

Complex Spark SQL Data Types:

Scala Type	SQL Type
Array[T]	ArrayType(elementType, containsNull)
Map[K, V]	MapType(keyType, valueType, valueContainsNull)
case class	StructType(List[StructFields])

- containsNull: null값을 가질 수 있는지
- Structs:

Structs

Struct type with list of possible fields of different types. `containsNull` *is set to true if the elements in* `StructFields` *can have null values.*

Example:

```
// Scala type
case class Person(name: String, age: Int)
```

```
// SQL type
StructType(List(StructField("name", StringType, true),
                StructField("age", IntegerType, true)))
```

- ex) Complex Data Types Can be combined

```
// Scala type
case class Account(
  balance: Double,
  employees:
  Array[Employee])

case class Employee(
  id: Int,
  name: String,
  jobTitle: String)

case class Project(
  title: String,
  team: Array[Employee],
  acct: Account)
```

```
// SQL type
StructType(
  StructField(title,StringType,true),
  StructField(
    team,
    ArrayType(
      StructType(StructField(id,IntegerType,true),
      StructField(name,StringType,true),
      StructField(jobTitle,StringType,true)),
      true),
    true),
  StructField(
    acct,
    StructType(
      StructField(balance,DoubleType,true),
      StructField(
        employees,
        ArrayType(
          StructType(StructField(id,IntegerType,true),
          StructField(name,StringType,true),
          StructField(jobTitle,StringType,true)),
          true),
        true)
      ),
    true)
  )
```

* Acessing Spark SQL Types → import org.apach.spark.sql.types._

* RDD API vs. DF API

- The main difference between the RDD API and the DataFrames API was that DataFrame APIs accept Spark SQL expressions, instead of arbitrary user-defined function literals like we were used to on RDDs.
This allows the optimizer to understand what the computation represents, and for example with filter, it can often be used to skip reading unnecessary record.

* Getting a look at your data

- show(): pretty-prints DataFrame in tabular form. Shows first 20 elements.

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)
val employeeDF = sc.parallelize(...).toDF
employeeDF.show()
// +---+-----+-------+---+--------+
// | id|fname|  lname|age|    city|
// +---+-----+-------+---+--------+
// | 12|  Joe|  Smith| 38|New York|
// |563|Sally|  Owens| 48|New York|
// |645|Slate|Markham| 28|  Sydney|
// |221|David| Walker| 21|  Sydney|
// +---+-----+-------+---+--------+
```

- printSchema() prints the schema of your DataFrame in a tree format.

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)
val employeeDF = sc.parallelize(...).toDF
employeeDF.printschema()

// root
//  |-- id: integer (nullable = true)
//  |-- fname: string (nullable = true)
//  |-- lname: string (nullable = true)
//  |-- age: integer (nullable = true)
//  |-- city: string (nullable = true)
```

* Common DataFrame Transformations

- RDD와 비슷하게, operation which returns DF , lazily evaluated
- filter, limit, orderBy, where, as, sort, union, drop

```
def select(col: String, cols: String*): DataFrame
// selects a set of named columns and returns a new DataFrame with these
// columns as a result.

def agg(expr: Column, exprs: Column*): DataFrame
// performs aggregations on a series of columns and returns a new DataFrame
// with the calculated output.

def groupBy(col1: String, cols: String*): DataFrame // simplified
// groups the DataFrame using the specified columns. Intended to be used before an aggregation.

def join(right: DataFrame): DataFrame // simplified
// inner join with another DataFrame
```

- 보통 groupby하고 agg한다. 모두 DF를 return.

* Specifying Columns: 열은 이렇게 부른다.

1. **Using $-notation**

```
// $-notation requires: import spark.implicits._
df.filter($"age" > 18)
```

2. **Referring to the Dataframe**

```
df.filter(df("age") > 18))
```

3. **Using SQL query string**

```
df.filter("age > 18")
```

가끔 이 방법은 에러날 때 가 있다.

* 전 강의의 Example by Scala

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)
val employeeDF = sc.parallelize(...).toDF

val sydneyEmployeesDF = employeeDF.select("id", "lname")
                                  .where("city == 'Sydney'")
                                  .orderBy("id")
```

* Filtering: filter와 where는 같다.

```
val over30 = employeeDF.filter("age > 30").show()
val over30 = employeeDF.where("age > 30").show()
```

- 근데 filter는 더 많은 조건 추가 가능

```
employeeDF.filter(($"age" > 25) && ($"city" === "Sydney")).show()
```

* Grouping and Aggregating on DataFrames

- ex)

```
case class Listing(street: String, zip: Int, price: Int)

val listingsDF = ... // DataFrame of Listings

import org.apache.spark.sql.functions._

val mostExpensiveDF = listingsDF.groupBy($"zip")
                                .max("price")

val leastExpensiveDF = listingsDF.groupBy($"zip")
                                 .min("price")
```

- another ex) 결과 →

```
import org.apache.spark.sql.functions._

val rankedDF =
  postsDF.groupBy($"authorID",$"subforum")
         .agg(count($"authorID")) // new DF with columns authorID, subforum, count(authorID)
         .orderBy($"subforum",$"count(authorID)".desc)
```

```
// postsDF:
// +--------+--------+-----+----+
// |authorID|subforum|likes|date|
// +--------+--------+-----+----+
// |       1| design|    2|    |
// |       1| debate|    0|    |
// |       2| debate|    0|    |
// |       3| debate|   23|    |
// |       1| design|    1|    |
// |       1| design|    0|    |
// |       2| design|    0|    |
// |       2| debate|    0|    |
// +--------+--------+-----+----+
```

rankedDF:

authorID	subforum	count(authorID)
2	debate	2
1	debate	1
3	debate	1
1	design	3
2	design	1

- groupby를 하고 RelationalGroupedDataset에 할 수 있는 metods

http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.RelationalGroupedDataset

- agg로 할 수 있는 methods: ex) min, max, sum, mean, stddev, count, avg, first, last.

http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$

19. DataFrames(2)

- Working with missing values / common actions on DFs / Joins on DFs / Optimizations of DFs 에 대해

* Cleaning Data with DFs

- Dropping records with unwanted values

- **drop()** drops rows that contain null or NaN values in **any** column and returns a new DataFrame.
- **drop("all")** drops rows that contain null or NaN values in **all** columns and returns a new DataFrame.
- **drop(Array("id", "name"))** drops rows that contain null or NaN values in the **specified** columns and returns a new DataFrame.

- Replacing unwanted values

- **fill(0)** replaces all occurrences of null or NaN in **numeric columns** with **specified value** and returns a new DataFrame.
- **fill(Map("minBalance" -> 0))** replaces all occurrences of null or NaN in **specified column** with **specified value** and returns a new DataFrame.
- **replace(Array("id"),Map(1234 -> 8923))** replaces **specified value** (1234) in **specified column** (id) with **specified replacement value** (8923) and returns a new DataFrame.

* Common Actions on DFs

collect(): Array[Row]
Returns an array that contains all of Rows in this DataFrame.

count(): Long
Returns the number of rows in the DataFrame.

first(): Row/head(): Row
Returns the first row in the DataFrame.

show(): Unit
Displays the top 20 rows of DataFrame in a tabular form.

take(n: Int): Array[Row]
Returns the first n rows in the DataFrame.

returns to master nodes!

* Joins on DFs

- inner, outer, left_outer, right_outer, leftsemi
- RDD와 다르게 모두 join을 쓰며, join 메소드 안에 어떤 join인지 명시한다.

```
val abosDF = sc.parallelize(as).toDF
val locationsDF = sc.parallelize(ls).toDF
```

```
abosDF:
+---+--------------------+
| id|                   v|
+---+--------------------+
|101|         [Ruetli,AG]|
|102|  [Brelaz,DemiTarif]|
|103|[Gress,DemiTarifV...|
|104|[Schatten,DemiTarif]|
+---+--------------------+
```

```
locationsDF:
+---+---------+
| id|        v|
+---+---------+
|101|     Bern|
|101|     Thun|
|102| Lausanne|
|102|   Geneve|
|102|     Nyon|
|103|   Zurich|
|103|St-Gallen|
|103|     Chur|
+---+---------+
```

```
val trackedCustomersDF =
  abosDF.join(locationsDF, abosDF("id") === locationsDF("id"))
```

```
trackedCustomersDF:
+---+--------------------+---+---------+
| id|                   v| id|        v|
+---+--------------------+---+---------+
|101|         [Ruetli,AG]|101|     Bern|
|101|         [Ruetli,AG]|101|     Thun|
|103|[Gress,DemiTarifV...|103|   Zurich|
|103|[Gress,DemiTarifV...|103|St-Gallen|
|103|[Gress,DemiTarifV...|103|     Chur|
|102|  [Brelaz,DemiTarif]|102| Lausanne|
|102|  [Brelaz,DemiTarif]|102|   Geneve|
|102|  [Brelaz,DemiTarif]|102|     Nyon|
+---+--------------------+---+---------+
```

```
val abosWithOptionalLocationsDF
  = abosDF.join(locationsDF, abosDF("id") === locationsDF("id"), "left_outer")
//   +---+--------------------+----+---------+
//   | id|                   v|  id|        v|
//   +---+--------------------+----+---------+
//   |101|         [Ruetli,AG]| 101|     Bern|
//   |101|         [Ruetli,AG]| 101|     Thun|
//   |103|[Gress,DemiTarifV...| 103|   Zurich|
//   |103|[Gress,DemiTarifV...| 103|St-Gallen|
//   |103|[Gress,DemiTarifV...| 103|     Chur|
//   |102|  [Brelaz,DemiTarif]| 102| Lausanne|
//   |102|  [Brelaz,DemiTarif]| 102|   Geneve|
//   |102|  [Brelaz,DemiTarif]| 102|     Nyon|
//   |104|[Schatten,DemiTarif]|null|     null|
//   +---+--------------------+----+---------+
```

* Revisiting Selecting Scholarship Example (pg. 1)

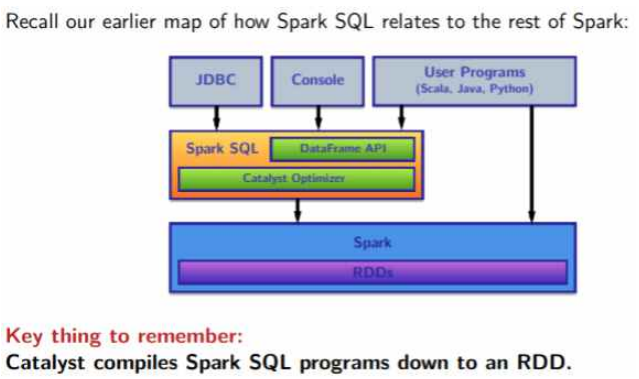
- RDDs에서 join하고 filtering 했던 첫 번째 방법을 DF API로 하면 아래와 같다.

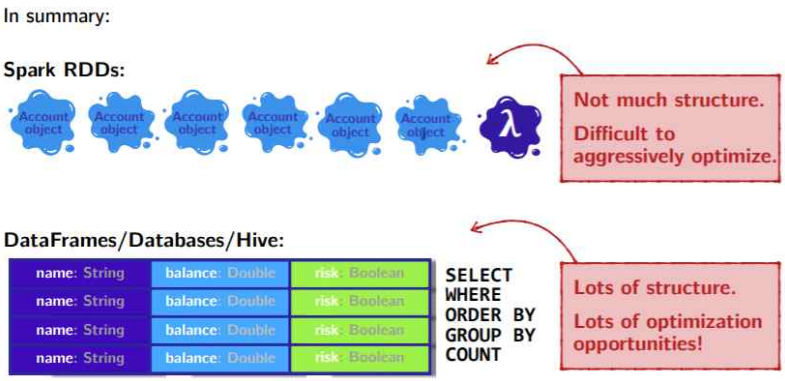
```
demographicsDF.join(financesDF, demographicsDF("ID") === financesDF("ID"), "inner")
        .filter($"HasDebt" && $"HasFinancialDependents")
        .filter($"CountryLive" === "Switzerland")
        .count
```

```
demographics.join(finances)
            .filter { p =>
              p._2._1.country == "Switzerland" &&
              p._2._2.hasFinancialDependents &&
              p._2._2.hasDebt
            }.count
```

- 첫 번째 방법은 필터링 하고 join한 두 번째 보다 3~4배 느렸는데 데이터프레임으로 한 것은 두 번째 방법보다 빠르다.
- DF API는 성능을 위해 연산을 reoder 한다.

* Optimization.





*** Optimizations by Catalyst (Spark SQL's query optimizer)**

- Catalyst has full knowledge and understanding of all data types.
  knows the exact schema of our data.
  has detailed knowledge of the computation we'd like to do 하기 때문에
- 다음 과 같은 것이 가능하다.

a. Reordering operations:

lazy함과 structure이 연산 DAG에서 실행되기 전에 연산들에 대한 비용을 통해 reordering을 가능하게 한다.

b. Reduce the amount of data we must read.

연산에 필요 없는 데이터 parts에 대한 reading, serializing, sending을 skip한다.

ex) 필요 없는 fields(cols) -> 이게 안에 어떤 구조인지 아니까 가능한 것이다.

c. Pruning unneeded partitioning.

연산에 필요 없는 파티션을 skip한다.

*** Optimizations by Tungsten (Spark SQL's off-heap data encoder)**

- data types 가 제한적이기 때문에 Tungsten은

a. highly-specialized data encoders

텅스텐은 스키마 정보를 통해 serialize를 tight하게 메모리에 pack할 수 있으며, CPU의 일인 deserialization, serialization을 더 빠르고 더 많은 데이터에 대해 할 수 있게 한다.

b. column-based

DBMS에서 잘 알려진 방식으로 대부분의 연산에서 column 기반으로 저장하는 것이 row 기반보다 더 빠르다.

c. off-heap (free from garbage collection overhead)

시리얼라이즈 된 정보를 off-heap에 jvm의 가비지컬렉션에서 벗어나기 위해
가비지컬렉션안에 있으면 성능이나 메모리 관리에 영향을 받는다.

* Limitations of DFs

a. Untyped 이기 때문에 아래와 같이 'state' col이 없어도 컴파일이 되고 에러를 나중에 난다.

```
listingsDF.filter($"state" === "CA")

// org.apache.spark.sql.AnalysisException:
//cannot resolve ''state'' given input columns: [street, zip, price];;
```

b. Limited Data Types

case classes/ Products 또는 표준 spark SQL data types로 표현이 안되면
Tungsten encoder에 그 데이터 타입이 있을지 모른다. ex) regular scala class 와 같은 것들

c. Semi-Structured / Structured Data 여야 한다.

이게 안되면 RDDs를 쓰는게 낫다.

20. Datasets

- newest major SPARK API 이다.

* Example: DF를 사용해 zip코드 별 집 가격을 구하려고 할 때, 오른쪽 아래와 같이 짜고 collect()를 해보면 말이 되는 줄 알았지만, type casting을 안 해줬기 때문에 Array안에 Row가 있는 형태로 반환하고 이를 다시

```
case class Listing(street: String, zip: Int, price: Int)
val listingsDF = ... // DataFrame of Listings

import org.apache.spark.sql.functions._
val averagePricesDF = listingsDF.groupBy($"zip")
                                .avg("price")
```

```
val averagePricesAgain = averagePrices.map {
  row => (row(0).asInstanceOf[String], row(1).asInstanceOf[Int])
}
```

위와 같이 type casting을 해주면, java.lang.ClassCastException 에러가 난다. 그래서 row(averagePrices)의 스키마를 보면

```
val averagePrices = averagePricesDF.collect()
// averagePrices: Array[org.apache.spark.sql.Row]
```

zip은 integer, avg(price)는 double인 것을 통해 아래와 같이 다시 type casting을 해줄 수 있다.
(type casting을 안하면 type을 유추해야한다.)

```
val averagePricesAgain = averagePrices.map {
  row => (row(0).asInstanceOf[Int], row(1).asInstanceOf[Double]) // Ew...
}
// mostExpensiveAgain: Array[(Int, Double)]
```

이러한 과정의 문제점(DF가 untyped인 것으로 인한)을 해소를 위해 Dataset을 만들었고, Spark SQL optimization과 typesafety를 갖는 Holy grail과 같은 데이터 구조로 여겨진다.

- type safety란 계속 타입을 유지하고, track할 수 있으면 safe하다 한다.

* Dataset의 정의: type DataFrame = Dataset[Row]

데이터프레임은 Dataset의 type이 row인 것이다. (DFs are actually Datasets.)

- Datasets can be thought of as **typed** distributed collections of data.
- Dataset API unifies the **DataFrame** and **RDD APIs**. Mix and match!
- Datasets require strucutred/semi-structured data. Schemas and Encoders core part of Datasets. (DF 같이)

→ Think of Datasets as a compromise between RDDs & DataFrames.

You get more type information on Datasets than on DataFrames, and you get more optimizations on Datasets than you get on RDDs.

- DF에서 배운 relational operation과 typed operations를 사용할 수 있고,
RDDs에서 사용했던 higher-order functions (map, flatMap, filter ...)도 사용 가능하다.

- type safety는 완전 완벽하지는 않고 almost perfectly safe 하다.
(Functional은 Type safety를 보장하지만, Relational operation은 아니므로)

* Example (RDDs와 DF의 중간이라는 것이라는 느낌은 다음과 같다.)
위와 같은 예제에서 첫 줄은 RDD 같이 두 번째 줄은 DF와 같이 섞어 사용가능하다.

```
listingsDS.groupByKey(l => l.zip)     // looks like groupByKey on RDDs! passing lambda function!
      .agg(avg($"price").as[Double])  // looks like our DataFrame operators!
```

* Creating Datasets
- From JSON: `val myDS = spark.read.json("people.json").as[Person]`
people.json이 Person의 인스턴스로 구성되어있고 Person이라는 case class가 defined되어 있을 때,
위와 같이 dataset을 읽을 수 있다.
- From a DF: `myDF.toDS  // requires import spark.implicits._`
- From a RDD: myRDD.toDS // requires import spark.implicits._
- From common Scala types: List("yay", "ohnoes", "hooray!").toDS // requires import spark.implicits._

* Typed Columns
DF와 다르게 Dataset은 typed이기 때문에 typed columns의 개념이 나왔다. 이전에 하던 것처럼 아래와 같이 코드를 작성하면 에러가 날 것이고 Typed column이 필요하므로 오른쪽과같이 Typed column으로 만들어 준다.

```
<console>:58: error: type mismatch;
 found   : org.apache.spark.sql.Column
 required: org.apache.spark.sql.TypedColumn[...]
                .agg(avg($"price")).show
                     ^
```

```
$"price".as[Double]  // this now represents a TypedColumn.
```

- 이제 설명할 method들은 Typed column 이여야 수행된다.

* Transformations on Datasets
- Untyped Transformation from DFs, typed Transformation from RDDs. 로 크게 2가지로 볼 수 있다.
Untyped를 쓸 때는 type safety를 잃을 수 있으니 유념하고 해야한다.
- API가 통합되어 있어, DF에서 Dataset으로 map과 같은 functional operation을 할 수 있지만, type 정보를 제공해야한다. (Untyped to Typed 이며 이는 이쁘진 않다.) 그리고 RDD의 모든 함수를 Dataset에서 사용할 수 있는 것은 아니므로 API Docs를 참고하는 것이 좋다.

```
val keyValuesDF = List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(3,"-)"),(2,"cre"),(2,"t")).toDF
val res = keyValuesDF.map(row => row(0).asInstanceOf[Int] + 1) // Ew...
```

- Common (Typed) Transformation on Datasets

map	**map[U](f: T => U): Dataset[U]** Apply function to each element in the Dataset and return a Dataset of the result.
flatMap	**flatMap[U](f: T => TraversableOnce[U]): Dataset[U]** Apply a function to each element in the Dataset and return a Dataset of the contents of the iterators returned.
filter	**filter(pred: T => Boolean): Dataset[T]** Apply predicate function to each element in the Dataset and return a Dataset of elements that have passed the predicate condition, pred.
distinct	**distinct(): Dataset[T]** Return Dataset with duplicates removed.

대부분은 RDD에서 본것들과 비슷하지만 groupByKey의 return type은 KeyValueGroupedDataset 으로 좀 다르다. Dataset을 return하지 않으니 transformation이 아닌 것이 아니라 이것이 궁극적으로 dataset을 return하는 것을 배울 것이다.

groupByKey	**groupByKey[K](f: T => K): KeyValueGroupedDataset[K, T]** Apply function to each element in the Dataset and return a Dataset of the result.
coalesce	**coalesce(numPartitions: Int): Dataset[T]** Apply a function to each element in the Dataset and return a Dataset of the contents of the iterators returned.
repartition	**repartition(numPartitions: Int): Dataset[T]** Apply predicate function to each element in the Dataset and return a Dataset of elements that have passed the predicate condition, pred.

- DF에서의 Groupby와 같이 Dataset에서 GroupByKey를 한 후에 사용하는 특별한 aggregation operation이 있다. KeyValueGroupedDataset은 Dataset을 return하는 aggregation operation들이 있다.
- KeyValueGroupedDataset Aggregation Operations

reduceGroups `reduceGroups(f: (V, V) => V): Dataset[(K, V)]`
Reduces the elements of each group of data using the specified binary function. The given function must be commutative and associative or the result may be non-deterministic.

agg `agg[U](col: TypedColumn[V, U]): Dataset[(K, U)]` agg는 TypedColumn을 받는다
Computes the given aggregation, returning a Dataset of tuples for each unique key and the result of computing this aggregation over all elements in the group.

따라서 다음과 같이한다. ex) `someDS.agg(avg($"column").as[Double])`
// as 로 typed로 만들어주고 typed가 아닌 일반 column이면 에러!가 난다.

mapGroups `mapGroups[U](f: (K, Iterator[V]) => U): Dataset[U]` // group애대해 reduce하는 것과 비슷하다.
Applies the given function to each group of data. For each unique group, the function will be passed the group key and an iterator that contains all of the elements in the group. The function can return an element of arbitrary type which will be returned as a new Dataset. //

flatMapGroups `flatMapGroups[U](f: (K, Iterator[V]) => TraversableOnce[U]): Dataset[U]` // flatmap과 비슷하다.
Applies the given function to each group of data. For each unique group, the function will be passed the group key and an iterator that contains all of the elements in the group. The function can return an iterator containing elements of an arbitrary type which will be returned as a new Dataset.

* 근데 그러면 reduceByKey는 어디갔니?
- 없어졌다. 하지만 우리가 지금까지 배운 것으로 만들어 낼 수 있다.

ex)
```
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))
```

Q. keyValuesRDD.reduceBykey(_+_) 의 RDD의 reduceByKey operation을 Dataset에서 구현해보자.

A. `val keyValuesDS = keyValues.toDS` 다양한 방법이 있다.

```
keyValuesDS.groupByKey(p => p._1) Dataset[(int, string)]에서 Int로 그룹핑하고
           .mapGroups((k, vs) => (k, vs.foldLeft("")((acc, p) => acc + p._2))).show()

           String에 대해 foldLeft를 통해 concat하면 오른쪽과 같이 되고
           .sort($"_1").show()

           sort하면 완성된다
```

```
+---+----------+
| _1|        _2|
+---+----------+
|  1|   ThisIsA|
|  3|Message:-)|
|  2|    Secret|
+---+----------+
```

```
+---+----------+
| _1|        _2|
+---+----------+
|  1|   ThisIsA|
|  2|    Secret|
|  3|Message:-)|
+---+----------+
```

→ 이게 됐지만 mapGroups를 사용하면, partial aggregation을 지원하지 않는 함수이므로, Dataset의 모든 데이터에 셔플링이 일어나므로 latency를 증가할 수 있다.
따라서, **reduce function**이나 **Aggregator**를 사용하는 것을 API Docs에서 권장한다.

a. Using reduce function

```
keyValuesDS.groupByKey(p => p._1)
           .mapValues(p => p._2)
           .reduceGroups((acc, str) => acc + str)
```

* Aggregators: 데이터를 generically Aggregate 하게 도와주는 class. RDDs에서의 aggregate 함수와 비슷하다.
- org.apache.spark.sql.expressions.Aggregator에 있으니 사용할 때 import를 해야한다.

```
class Aggregator[-IN, BUF, OUT]
```

- **IN** is the input type to the aggregator. When using an aggregator after `groupByKey`, this is the type that represents the value in the key/value pair.
- **BUF** is the intermediate type during aggregation.
- **OUT** is the type of the output of the aggregation.

- RDD에서와 같이 type을 변경할 수 있게 허락하며, 작업을 parallel하게 할 수 있게 한다.
  (computation을 몇 몇의 function으로 나누는 것을 결과를 나중에 결합하는)
- IN, Buf, OUT 모두 다른 type이어도 가능하다.
- ex)

```
val myAgg = new Aggregator[IN, BUF, OUT] {
  def zero: BUF = ...                    // The initial value.
  def reduce(b: BUF, a: IN): BUF = ...   // Add an element to the running total
  def merge(b1: BUF, b2: BUF): BUF = ... // Merge intermediate values.
  def finish(b: BUF): OUT = ...          // Return the final result.
}.toColumn
```

b. Using Aggregator (위의 예시)

```
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))

val keyValuesDS = keyValues.toDS

val strConcat = new Aggregator[(Int, String), String, String]{
  def zero: String = ""
  def reduce(b: String, a: (Int, String)): String = b + a._2
  def merge(b1: String, b2: String): String = b1 + b2
  def finish(r: String): String = r
}.toColumn                                          // Step 4: pass it to your aggregator!

keyValuesDS.groupByKey(pair => pair._1)
           .agg(strConcat.as[String])
```

그래서 이렇게 짜면 될 줄 알았는데, bufferEncoder와 oupputEncoder에서 compile error가 나온다.

```
[error] object creation impossible, since: it has 2 unimplemented members.
[error] the missing signatures are as follows.
[error]   def bufferEncoder: org.apache.spark.sql.Encoder[String] = ???
[error]   def outputEncoder: org.apache.spark.sql.Encoder[String] = ???
[error]   val strConcat = new Aggregator[(Int, String), String, String]{
[error]                   ^
[error] one error found
```

* Encoder - DF에서 배웠었다.
- JVM object에서의 data를 Spark SQL's specialized internal (tabular) representation으로 변환해주는 것.
- 모든 dataset이 encoder를 필요로 하며 인코더는 custom bytecode for serialization and deserialization of data를 generate하는 highly specialized, optimized code generators이다.
- 그리고 이렇게 serialized된 데이터는 Spark의 텅스텐의 binary format으로 저장되어 serialized data에 대해 operation을 할 수 있고 memory utilization을 증가할 수 있게 한다.(unpack하지 않고 더 넣을 수 있게)
Q. 왜 일반적인 Java나 Kryo serialization을 안 쓰는가?
A. 데이터 타입이 한정되어 있고, Encoder는 Schema 정보를 포함하고 알고 있음을 통해 code generators와 데이터의 형태에 따라 최적화할 수 있다. 그리고 이러한 특징이 텅스텐이 최적화하는 것을 가능하게 한다.
그래서 java/Kryo serialization 보다 10배 더 빠르고 메모리도 적게 쓴다.
(RDDs 에서는 자바가 Kryo보다 빨라 Kryo를 권장하지만 Dataset에서는 Encoder를 권장한다.)

- Dataset에게 encoder를 introduce하는 2가지 방법

  a. Automatically (genrally the case) 이래서 아직 배우지 않은 것이다.

  via implicits from a SparkSession.importspark.implicits._

  b. Explicitly via org.apache.spark.sql.Encoders which contains a large selection of methods for creating Encoders from Scala primitive types and Products.

  ex) **Some examples of 'Encoder' creation methods in 'Encoders':**

  - INT/LONG/STRING etc, for *nullable* primitives.
  - scalaInt/scalaLong/scalaByte etc, for Scala's primitives.
  - product/tuple for Scala's Product and tuple types.

  Example: Explicitly creating Encoders.

```
Encoders.scalaInt // Encoder[Int]
Encoders.STRING // Encoder[String]
Encoders.product[Person] // Encoder[Person], where Person extends Product/is a case class
```

* 다시 예시로 돌아와 Aggregator에 Encoder를 Explicitly introduce해 Compile error를 다음과 같이 두 줄을 추가하면 된다.

```
val strConcat = new Aggregator[(Int, String), String, String]{
  def zero: String= ""
  def reduce(b: String, a: (Int, String)): String = b + a._2
  def merge(b1: String, b2: String): String = b1 + b2
  def finish(r: String): String = r
  override def bufferEncoder: Encoder[String] = Encoders.STRING
  override def outputEncoder: Encoder[String] = Encoders.STRING
}.toColumn

keyValuesDS.groupByKey(pair => pair._1)
           .agg(strConcat.as[String]).show
```

```
// +-----+--------------------+
// |value|anon$1(scala.Tuple2)|
// +-----+--------------------+
// |    1|             ThisIsA|
// |    3|          Message:-)|
// |    2|              Secret|
// +-----+--------------------+
```

* Common Dataset Actions: RDDs와 DFs에서 본 action들과 같다.

collect(): Array[T]
Returns an array that contains all of Rows in this Dataset.

count(): Long
Returns the number of rows in the Dataset.

first(): T/head(): T
Returns the first row in this Dataset.

foreach(f: T => Unit): Unit
Applies a function f to all rows.

reduce(f: (T, T) => T): T
Reduces the elements of this Dataset using the specified binary function.

show(): Unit
Displays the top 20 rows of Dataset in a tabular form.

take(n: Int): Array[T]
Returns the first n rows in the Dataset.

* When to use Datasets vs. DataFrames vs. RDDs ?

Use RDDs when...

- you have unstructured data
- you need to fine-tune and manage low-level details of RDD computations
- you have complex data types that cannot be serialized with Encoders

(optimizer를 마냥 믿을 수 는 없다.)

- RDDs: RDDs는 스파크의 byte code로 볼 수 있는 low-level인 핵심 추상화이고 DFs나 DSs는 RDDs 위에서 구축되는 것이며, **결국 스파크가 실행하는 것은 RDD**이며 최적화를 하는 것은 사용자에게 달려있고 RDDs 내부에서 어떻게 작동하는지 알기 어렵다.

- 주로 Structured, semi-structured data(JSON, XML)일 때는 보통 DFs나 Datasets을 사용하는 것이 좋다.
  그리고 다음과 같은 상황으로 나누어 사용을 고려할 수 있다.

Use Datasets when...

- you have structured/semi-structured data
- you want typesafety
- you need to work with functional APIs
- you need good performance, but it doesn't have to be the best

Use DataFrames when...

- you have structured/semi-structured data
- you want the best possible performance, automatically optimized for you

* Limitations of Datasets

- Datasets don't get all of the optimizatino that DFs get.

Relational filter operation E.g., `ds.filter($"city".as[String] === "Boston")`.
Performs best because you're explicitly telling Spark which columns/attributes and conditions are required in your filter operation. With information about the structure of the data and the structure of computations, Spark's optimizer knows it can access only the fields involved in the filter without having to instantiate the entire data type. Avoids data moving over the network.
Catalyst optimizes this case.
When using Datasets with relational operations like select, you get all of Catalyst's optimizations .

Functional filter operation E.g., `ds.filter(p => p.city == "Boston")`. (람다 function을 볼 수 없기 때문에)
Same filter written with a function literal is opaque to Spark – it's impossible for Spark to introspect the lambda function. All Spark knows is that you need a (whole) record marshaled as a Scala object in order to return true or false, requiring Spark to do porentially a lot more work to meet that implicit requirement.
Catalyst cannot optimize this case.
When using Datasets with higher-order functions like map, you miss out on many Catalyst optimizations

하지만 모든 operation에 대해 Catalyst의 최적하는 갖지 못하지만 Tungsten은 여전히 사용하므로 RDDs 보다 좋은 성능을 기대할 수 있다.
또 DFs와 같이 제한적인 Data Types, Requiring Semi-Structured/Structured Data를 데이터셋의 한계이다.

* RDD를 사용해 filter와 join을 할 때 어떤 순서로 쓰는 것이 더 빠른가
- filtering을 하고 join을 하는 것이 약 3.6배 빠르다. 필터링을 하고 join의 연산을 하면 셔플링을 줄일 수 있기 때문이다.

* RDD와 Database/Hive에서 각각 다루는 데이터의 종류는 어떤 차이가 있는가.
- RDD unstructured 또는 semi-structured를 다루고, DB/Hive는 Structured 데이터를 다루며 엄격한 스키마 구조를 갖고 있다.

* Spark SQL의 세가지 목적
a. Support relational processing both within Spark programs (on RDDs) and on external data sources with a friendly API.
b. High performance, achieved by using techniques from research in DBs.
c. Easily support new data source such as semi-structured data and external DBs.

* Dataframe 이란
- Spark SQL의 core abstraction이며 known schema가 있으며, operation에 대해 자동적으로 최적화가 가능한 Untyped 구조의 Spark RDD에 대한 relational API이다.

* DataFrame을 사용해 filter와 join을 할 때 어떤 순서로 쓰는 것이 더 빠른가.
- 카탈리스트를 통해 reordering을 하므로 상관이 없다.

* 카탈리스트가 최적화 해주는 부분은 무엇인가.
- Reordering을 통한 operation 최적화, operation에 필요 없는 데이터에 대한 reading, serializing sending을 skip, operation에 필요없는 partition skip

* DataFrame의 제한점 (limitations)
- Untyped 이기 때문에 열의 이름을 잘못 불러도 컴파일이 되며 에러는 나중에 알 수 있다. Data type이 제한적이여서 regular scala class는 DF로 사용할 수 없다. Semi-Structed or Structed Data이여야한다.

* 데이터 셋
- newest major SPARK API이고, typed distributed collection of data이며, RDD와 Dataframe의 중간으로 생각할 수 있다.

* Dataset의 operation는 map과 select 중 카탈리스트가 최적화 가능한 것
- select는 가능하고 map은 안된다. 스파크에게 functional opeartion은 opaque하기 때문에, relational operation만 최적화가 가능하다.
- select이다. relational operation이 수행될 때 catalyst의 optimization이 효과적으로 발휘된다. map등의 higher-order function의 경우 catalyst가 optimization할 수 있는 여지가 별로 없기 때문에.