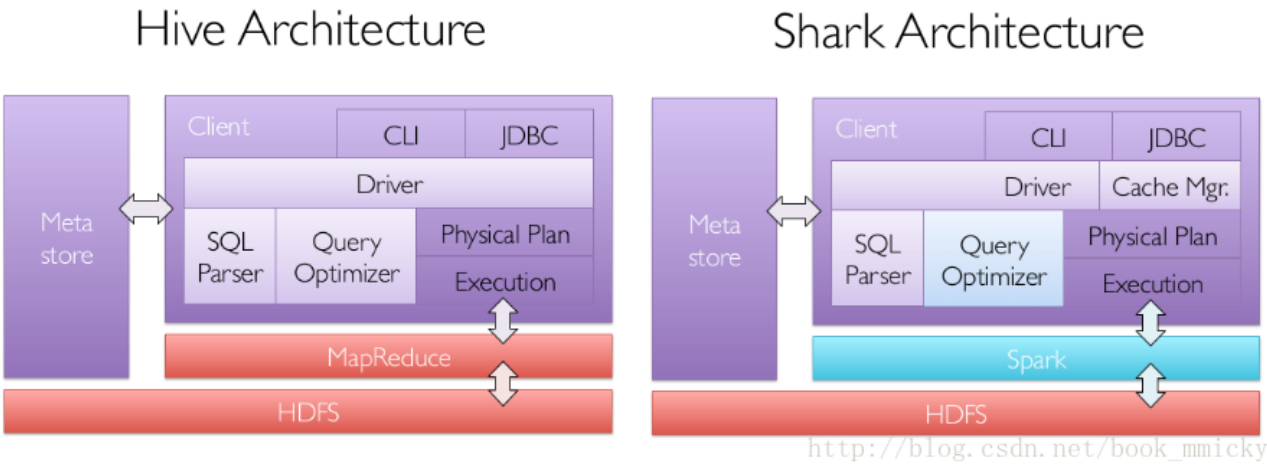
第五节课

SparkSQL历史

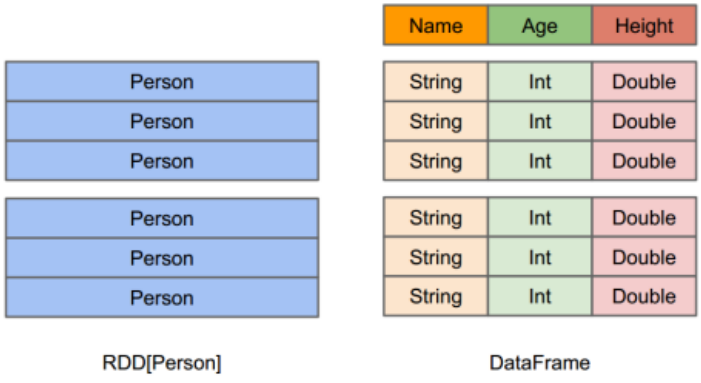
在hadoop发展过程中，为了给不理解MapReduce的技术人员提供快速上手的SQL工具，SQL-on-Hadoop的工具hive应运而生。但是，Hive中间环节消耗了大量的I/O，降低的运行效率，进而演化出

- MapR的Drill
- Cloudera的Impala
- Shark



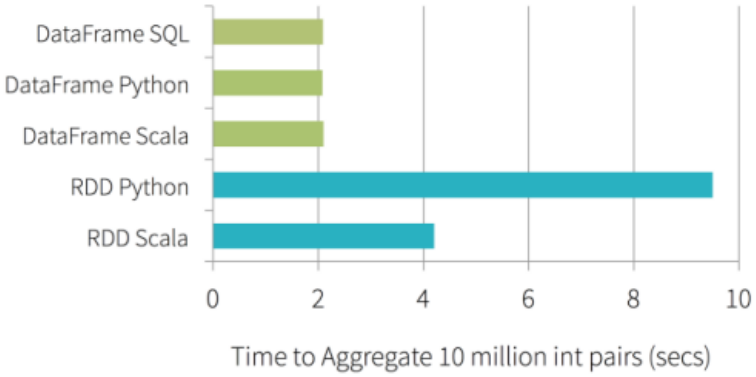
Shark是伯克利实验室spark生态环境的组件之一，它修改了内存管理、物理计划、执行三个模块，并使之能运行在spark引擎上，从而使得SQL查询的速度得到10-100倍的提升；因为Shark依赖hive太重，制约spark各模块的集合，抛弃shark代码后，发展出sparkSQL；

SparkSQL是分布式的即时查询工具，属于SPark系列的一个模块，它利用了数据更多的结构化信息，加入了更多的优化措施，可以对结构化数据进行选择、过滤、聚合等操作。

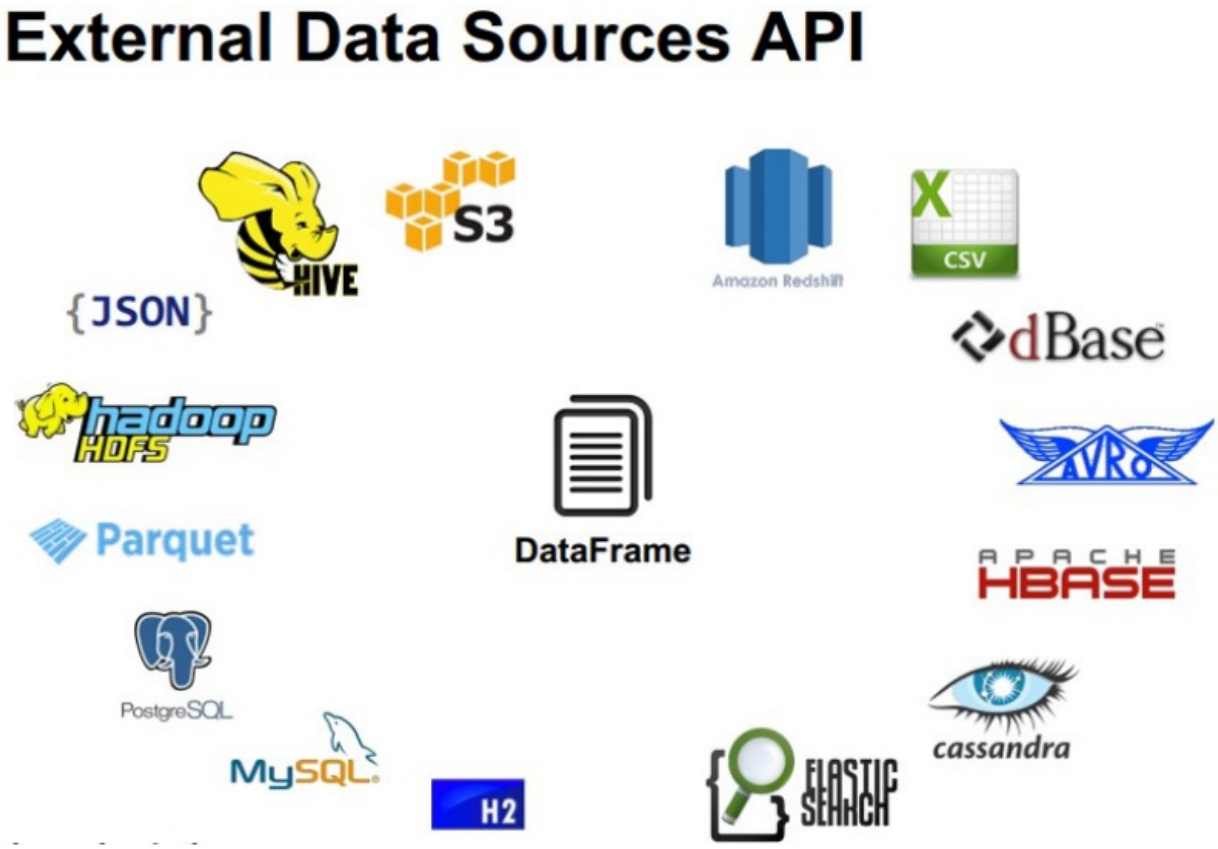


SparkSQL优点

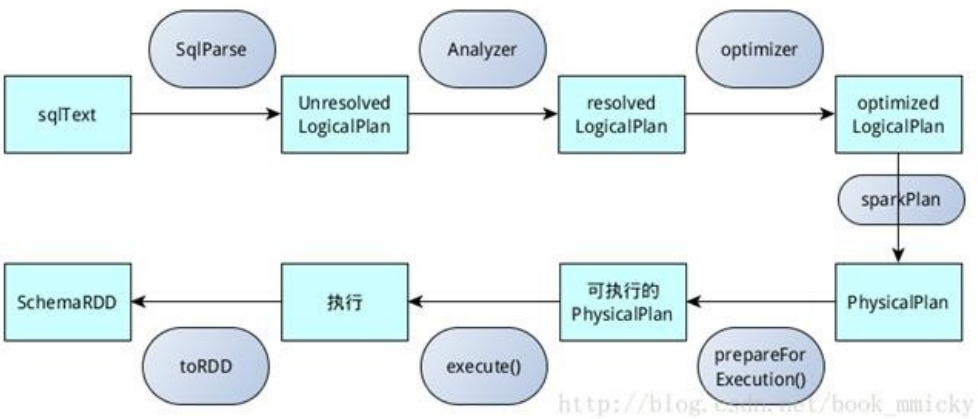
- 性能优化方面 除了采取In-Memory Columnar Storage、byte-code generation等优化技术外、将会引进Cost Model对查询进行动态评估、获取最佳物理计划等等



- 组件扩展方面 无论是SQL的语法解析器、分析器还是优化器都可以重新定义，进行扩展
- Scala编写代码的时候，尽量避免低效的、容易GC的代码
- 数据源兼容丰富：结构化数据文件、Hive数据表、已存在的RDD、外部数据库



SparkSQL执行与API



SparkSQL优化

- spark.sql.codegen 当它设置为true时，Spark SQL会把每条查询的语句在运行时编译为java的二进制代码，大型查询时才开启

Caching Data In Memory Then compression to minimize memory usage and GC pressure

- spark.sql.inMemoryColumnStorage.compressed 自动对内存中的列式存储进行压缩
- spark.sql.inMemoryColumnarStorage.batchSize 列式缓存时的每个批处理的大小；增大可以提高内存使用率与压缩，但容易OOM
- spark.sql.parquet.compressed.codec 压缩算法snappy/gzip/lzo
- spark.sql.tungsten.enabled tungsten优化

代码：

```
/**
  * Created by ding on 2018/1/6.
  */
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.{DataFrame, SQLContext}
import org.apache.spark.storage.StorageLevel

object sql {
  //模式类 默认序列化与支持模式匹配
  case class Person(name:String, age:Int)
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("SQL").setMaster("local[4]")
    val sc = new SparkContext(conf)
    sc.setLogLevel("WARN")
    val wordcount = sc.textFile("C:\\Users\\zxy\\Desktop\\Spark1.txt").flatMap(_.split(" ")).map((_,
1)).reduceByKey(_+_).persist(StorageLevel.MEMORY_AND_DISK)
    wordcount.collect().foreach(println)
    val sqlContext = new SQLContext(sc)
    /*  val spark = SparkSession
        .builder()
        .appName("Spark SQL basic example")
        .config("spark.some.config.option", "some-value")
        .getOrCreate()
    */
    import sqlContext.implicits._
    //创建一个数据集
    val peopleDF: DataFrame = sqlContext.read.json("C:\\Users\\zxy\\Desktop\\people.json")

    println("=======================数据集查看==========================")
    //数据集查看
    peopleDF.show()
    //查看数据表的结构schema
    peopleDF.printSchema()
    //选择列进行查看
```

```
    peopleDF.select("name").show()
    peopleDF.select($"name", $"age" + 1).show()
    //过滤列进行查看
    peopleDF.filter($"age" > 21).show()
    //聚合列
    peopleDF.groupBy("age").count().show()
    //peopleDF.createOrReplaceTempView("people")
    peopleDF.registerTempTable("peopleTable")
    peopleDF.registerTempTable("peopleTable_back")
    sqlContext.sql("select a.*, b.age from peopleTable a join peopleTable_back b on a.name = b.name").show
    peopleDF.join(peopleDF).show
    println("======================程序化查询==========================")
    //程序式的运行SQL查询
    val temporaryPeopleDF1: DataFrame =  peopleDF.select("name")
    temporaryPeopleDF1.show()

    val temporaryPeopleDF2: DataFrame = sqlContext.sql("select age from peopleTable")
    temporaryPeopleDF2.show()
    sqlContext.sql("select max(age) as maxAge from peopleTable").show
    peopleDF.map(_.getAs[String]("name")).foreach(println)
    peopleDF.map(_.getValuesMap[Any](List("name","age"))).foreach(println)

    println("======================反射生成==========================")
    //DataFrame与RDD的互相转换
    //1 使用反射推断schema
    //代码更加简洁 并且你已经确定RDD的结构

    import sqlContext.implicits._
    val people = sc.makeRDD(Seq("july,35", "tine,18")).map(_.split(","))
    val peopleDF_1: RDD[Person] = people.map(p => Person(p(0),p(1).trim.toInt))
    peopleDF_1.toDF().show()
    sqlContext.createDataFrame(peopleDF_1).show()

    println("======================显式指明==========================")
    //2 程序式的指明schema结构
    //代码复杂，数据运行时才能确定结构
    val schemaString = "name age"
    import org.apache.spark.sql.Row
    import org.apache.spark.sql.types.{StructType,StructField,StringType};
    val schema: StructType = StructType(
      schemaString.split(" ").map(p=> StructField(p , StringType, true))
    )
    val rowRDD: RDD[Row] = people.map(p => Row(p(0), p(1)))
    val peopleDF_2: DataFrame = sqlContext.createDataFrame(rowRDD, schema)
    peopleDF_2.show()
    Console.readLine()
    //保存数据
    //peopleDF_2.write.save("")
  }
}
```