Spark的RDD原理以及2.0特性的介绍

2016-05-19 Hadoop技术博文

编者按：本文由王联辉在高可用架构群分享，**本文转载自高可用架构**「 ArchNotes 」。

> 王联辉，曾在腾讯，Intel 等公司从事大数据相关的工作。2013 年 - 2016 年先后负责腾讯 Yarn 集群和 Spark 平台的运营与研发。曾负责 Intel Hadoop 发行版的 Hive 及 HBase 版本研发。参与过百度用户行为数据仓库的建设和开发，以及淘宝数据魔方和淘宝指数的数据开发工作。给 Spark 社区贡献了 25+ 个 patch，接受的重要特性有 python on yarn-cluster，yarn-cluster 支持动态资源扩缩容以及 RDD Core 中 memory 管理等。

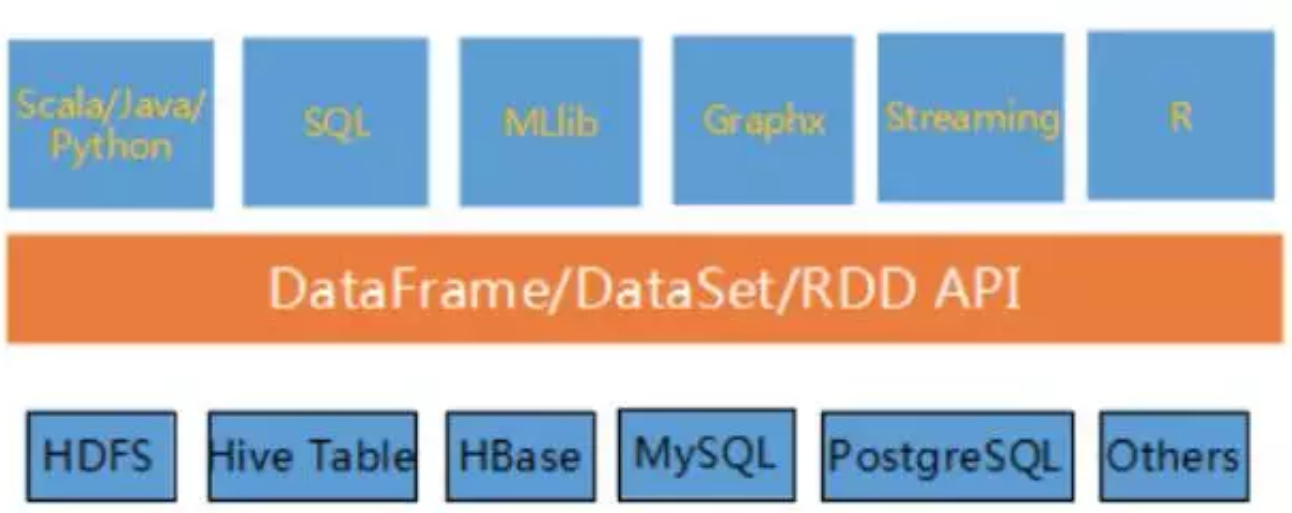
Spark 是什么

Spark 是 Apache 顶级项目里面最火的大数据处理的计算引擎，它目前是负责大数据计算的工作。包括离线计算或交互式查询、数据挖掘算法、流式计算以及图计算等。全世界有许多公司和组织使用或给社区贡献代码，社区的活跃度见 www.github.com/apache/spark。

2013 年开始 Spark开发团队成立 Databricks，来对 Spark 进行运作和管理，并提供 Cloud 服务。Spark 社区基本保持一个季度一个版本，不出意外的话 Spark 2.0 将在五月底发布。

与 Mapreduce 相比，Spark 具备 DAG 执行引擎以及基于内存的多轮迭代计算等优势，在 SQL 层面上，比 Hive/Pig 相比，引入关系数据库的许多特性，以及内存管理技术。另外在 Spark 上所有的计算模型最终都统一基于 RDD 之上运行执行，包括流式和离线计算。Spark 基于磁盘的性能是 MR 的 10 倍，基于内存的性能是 MR 的 100 倍 ❶ （见文后参考阅读❶ ，下同）。

Spark 提供 SQL、机器学习库 MLlib、流计算 Streaming 和图计算 Graphx，同时也支持 Scala、Java、Python 和 R 语言开发的基于 API 的应用程序。



RDD 的原理

> RDD，英文全称叫 Resilient Distributed Datasets。
> an RDD is a read-only, partitioned collection of records❸. 字面意思是只读的分布式数据集。

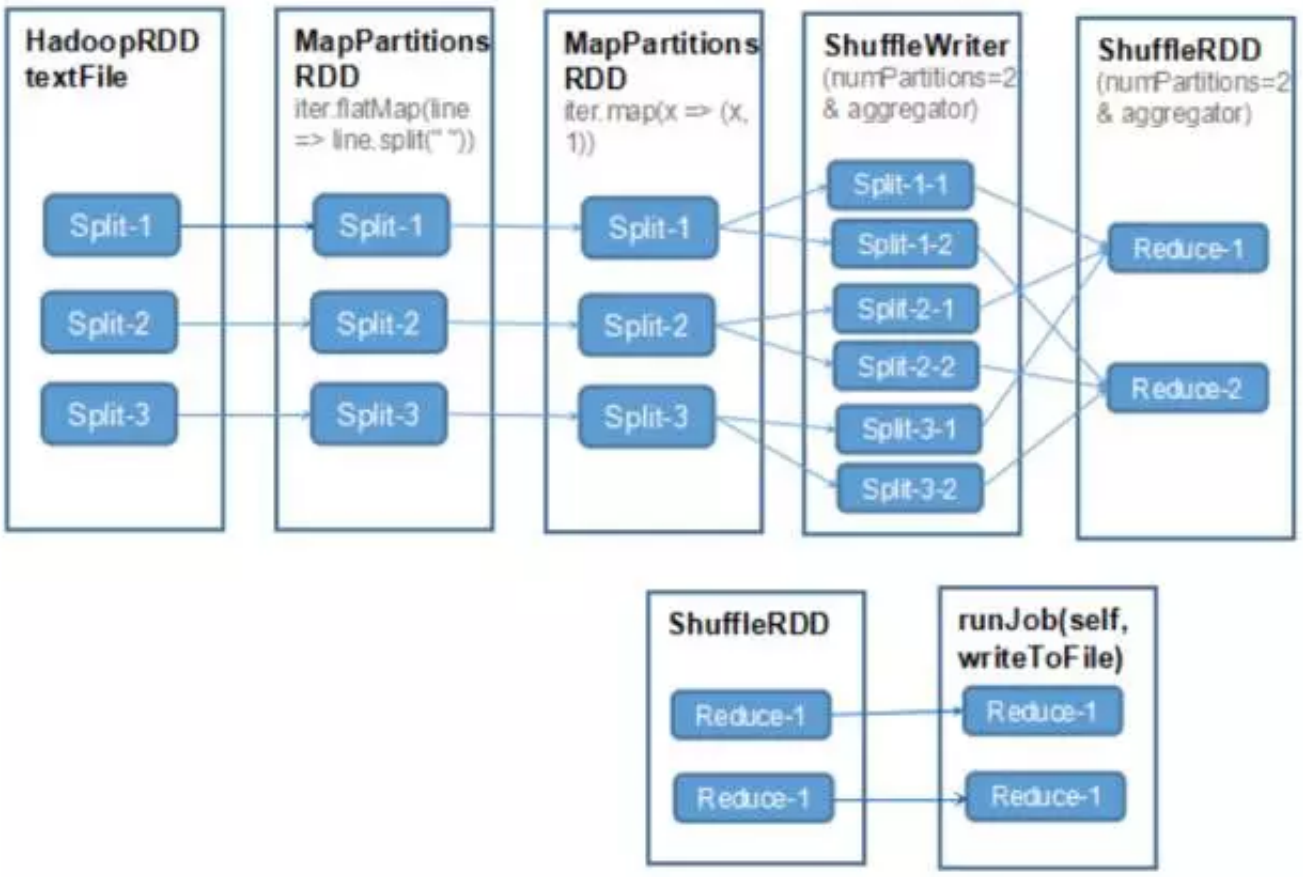
但其实个人觉得可以把 RDD 理解为关系数据库 里的一个个操作，比如 map，filter，Join 等。在 Spark 里面实现了许多这样的 RDD 类，即可以看成是操作类。当我们调用一个 map 接口，底层实现是会生成一个 MapPartitionsRDD 对象，当 RDD 真正执行时，会调用 MapPartitionsRDD 对象里面的 compute 方法来执行这个操作的计算逻辑。但是不同的是，RDD 是 lazy 模式，只有像 count，saveasText 这种 action 动作被调用后再会去触发 runJob 动作。

RDD 分为二类：transformation 和 action。

- transformation 是从一个 RDD 转换为一个新的 RDD 或者从数据源生成一个新的 RDD；
- action 是触发 job 的执行。所有的 transformation 都是 lazy 执行，只有在 action 被提交的时候才触发前面整个 RDD 的执行图。如下

```
val file = sc.textFile(args(0))
val words = file.flatMap(line => line.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _, 2)
wordCounts.saveAsTextFile(args(1))
```

这段代码生成的 RDD 的执行树是如下图所示：

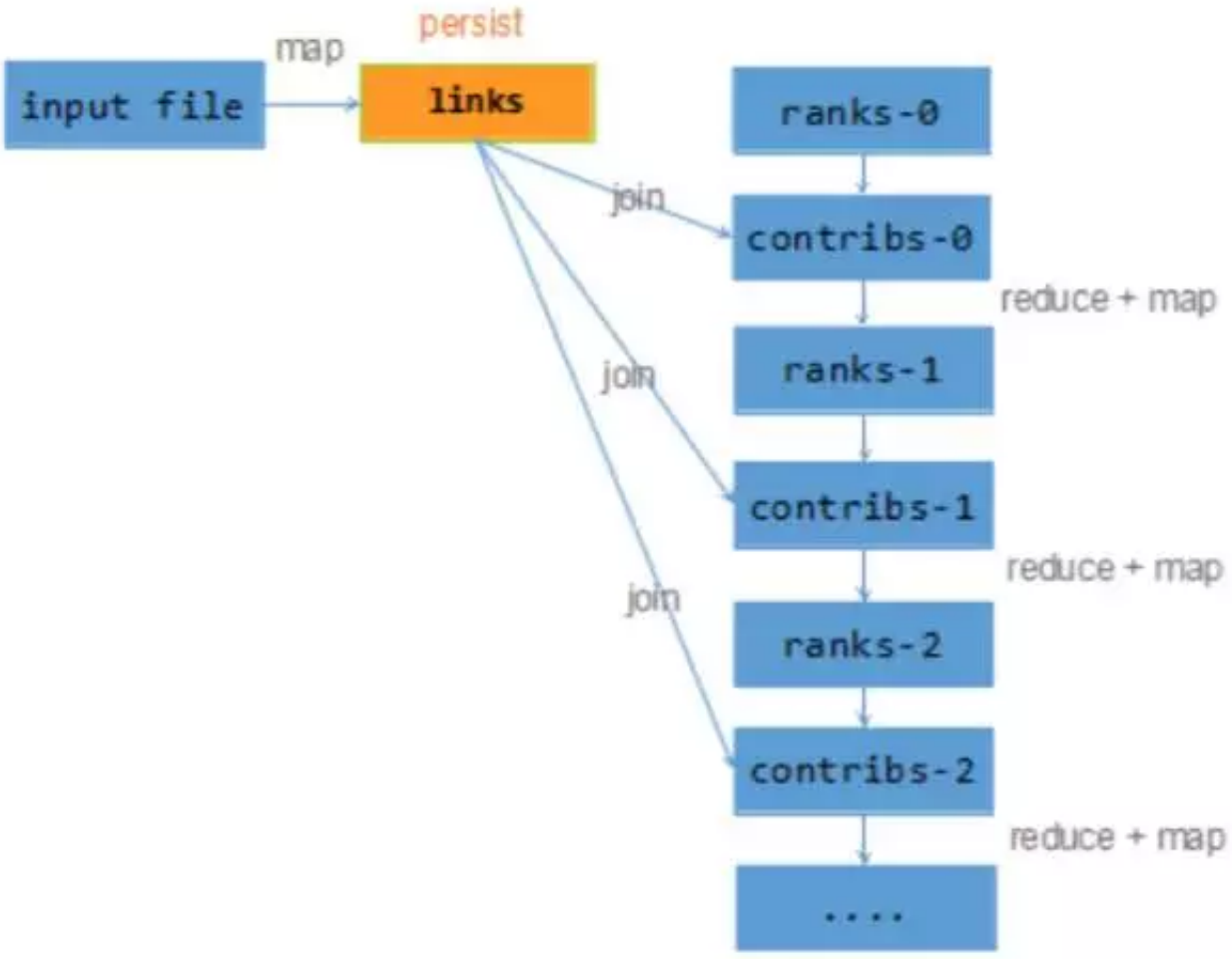


最终在 saveAsTextFile 方法时才会将整个 RDD 的执行图提交给 DAG 执行引擎，根据相关信息切分成一个一个 Stage，每个 Stage 去执行多个 task，最终完成整个 Job 的执行。

还有一个区别就是，RDD 计算后的中间结果是可以被持久化，当下一次需要使用时，可以直接使用之前持久化好的结果，而不是重新计算，并且这些结果被存储在各个结点的 executor 上。下一次使用时，调度器可以直接把 task 分发到存储持久化数据的结点上，减少数据的网络传输开稍。这种场景在数据挖掘迭代计算是经常出现。如下代码

```
val links = spark.textFile(...).map(...).persist() var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
// Build an RDD of (targetURL, float) pairs // with the contributions sent by each page val
contribs = links.join(ranks).flatMap {
(url, (links, rank)) =>
links.map(dest => (dest, rank/links.size)) }
// Sum contributions by URL and get new ranks

ranks = contribs.reduceByKey((x,y) => x+y)
.mapValues(sum => a/N + (1-a)*sum) }
```

以上代码生成的 RDD 执行树如下图所示：



计算 contribs-0 时需要使用 links 的计算逻辑，当 links 每个分片计算完后，会将这个结果保存到本地内存或磁盘上，下一次 contribs-1 计算要使用 links 的数据时，直接从上一次保存的内存和磁盘上读取就可以了。这个持久化系统叫做 **blockManager**，类似于在内部再构建了一个 KV 系统，K 表示每个分区 ID 号，V 表示这个分区计算后的结果。

另外在 streaming 计算时，每个 batch 会去消息队列上拉取这个时间段的数据，每个 Recevier 接收过来数据形成 block 块并存放到 blockManager 上，为了可靠性，这个 block 块可以远程备份，后续的 batch 计算就直接在之前已读取的 block 块上进行计算，这样不断循环迭代来完成流处理。

一个 RDD 一般会有以下四个函数组成。

1. 操作算子的物理执行逻辑

定义为：

def compute(split: Partition, context: TaskContext): Iterator[T]

如在 MapPartitionsRDD 里的实现是如下：

override def compute(split: Partition, context: TaskContext): Iterator[U] = f(context, split.index, firstParent[T].iterator(split, context))

函数定义

f: (TaskContext, Int, Iterator[T]) => Iterator[U]

2. 获取分片信息

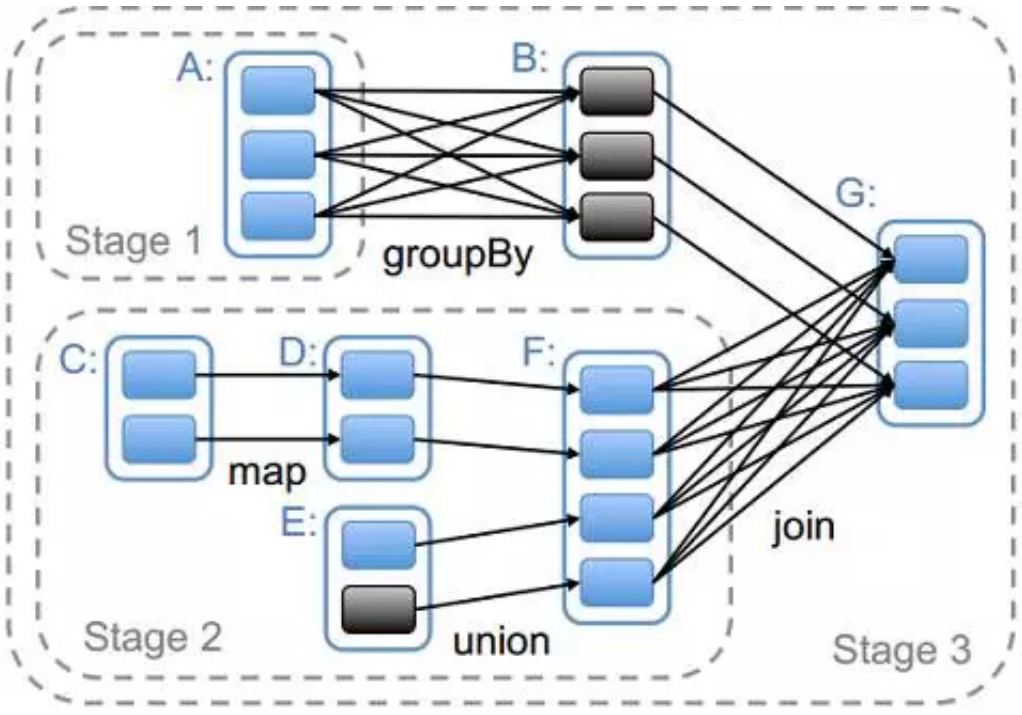
protected def getPartitions: Array[Partition]

即这个操作的数据划分为多少个分 区。跟 mapreduce 里的 map 上的 split 类似的。

3. 获取父 RDD 的依赖关系

protected def getDependencies: Seq[Dependency[_]]

依赖分二种：如果 RDD 的每个分区最多只能被一个 Child RDD 的一个分区使用，则称之为 narrow dependency；若依赖于多个 Child RDD 分区，则称之为 wide dependency。不同的操作根据其特性，可能会产生不同的依赖 ❹。如下图所示



map 操作前后二个 RDD 操作之间的分区是一对一的关系，故产生 narrow dependency，

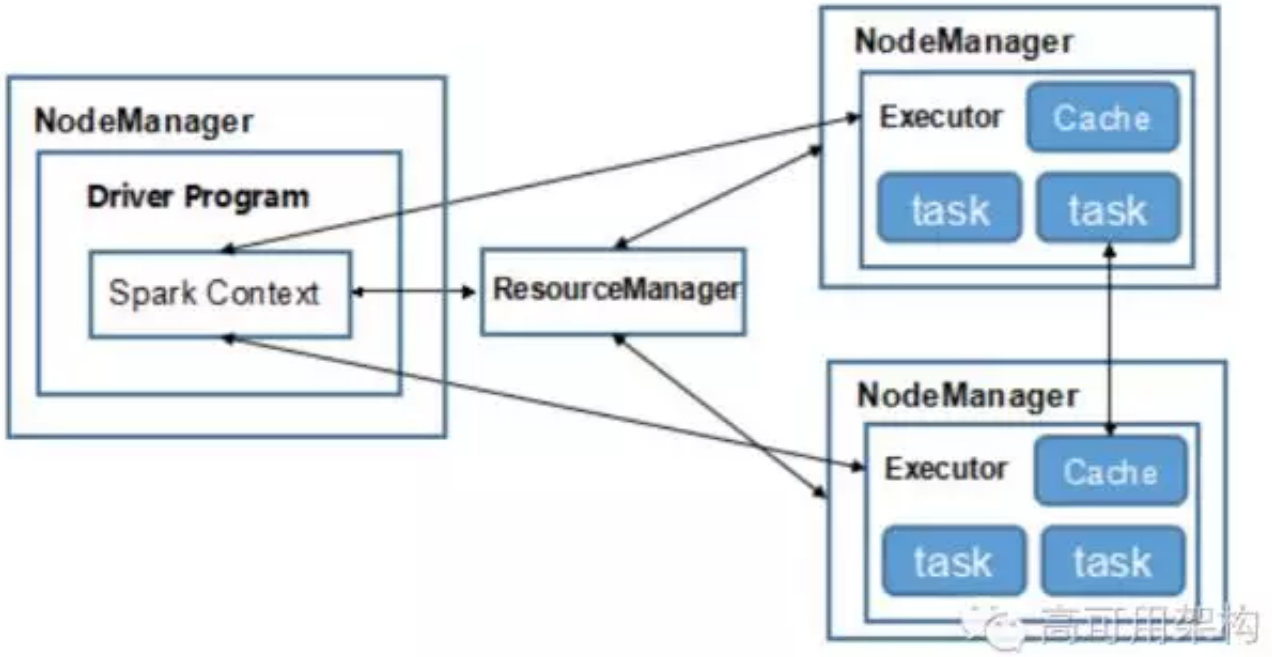
而 join 操作的分区分别对应于它的二个子操作相对应的分区，故产生 wide dependency。当最后要生成具体的 task 运行时，就需要利用这个依赖关系也生成 Stage 的 DAG 图。

4. 获取该操作对应数据的存放位置信息，主要是针对 HDFS 这类有数据源的 RDD。

protected def getPreferredLocations(split: Partition): Seq[String]

Spark 的执行模式

Spark 的执行模式有 local、Yarn、Standalone、Mesos 四类。后面三个分别有 cluster 和 client 二种。client 和 cluster 的区别就是指 Driver 是在程序提交客户端还是在集群的 AM 上。 比如常见的 Yarn-cluster 模式如下图所示：



一般来说，运行简单测试或 UT 用的是 local 模式运行，其实就是用多线程模似分布式执行。 如果业务部门较少且不需要对部门或组之间的资源做划分和优先级调度的话，可以使用 Standalone 模式来部署。

当如果有多个部门或组，且希望每个组织可以限制固定运行的最大资源，另外组或者任务需要有优先级执行的话，可以选择 Yarn 或 Mesos。

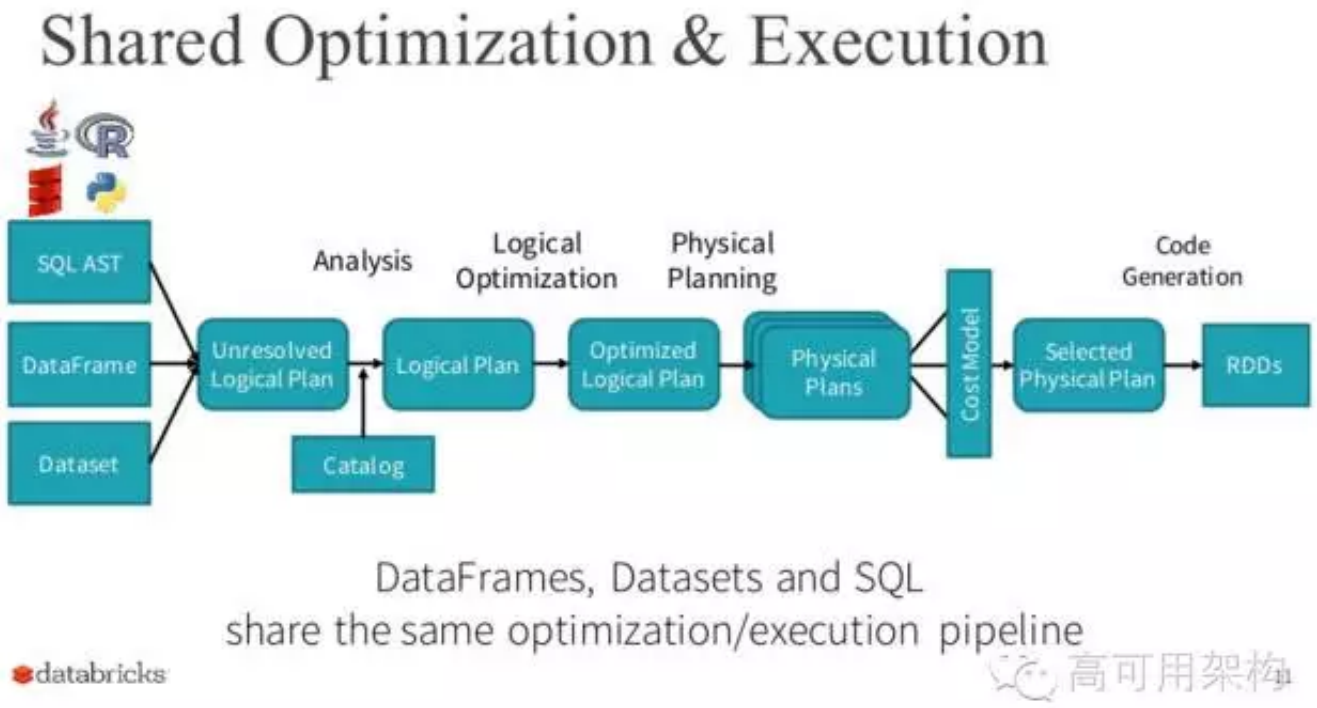
Spark 2.0 的特性

Unifying DataFrames and Datasets in Scala/Java

DataFrame❺ 和 Dataset❽ 的功能是什么?

它们都是提供给用户使用，包括各类操作接口的 API。1.3 版本引入 DataFrame，1.6 版本引入 Dataset，2.0 提供的功能是将二者统一，即保留 Dataset，而把 DataFrame 定义为 Dataset[Row]，即是 Dataset 里的元素对象为 Row 的一种(SPARK-13485)。

在参考资料❺ 中有介绍 DataFrame，它就是提供了一系列操作 API，与 RDD API 相比较，DataFrame 里操作的数据都是带有 Schema 信息，所以 DataFrame 里的所有操作是可以享受 Spark SQL Catalyst optimizer 带来的性能提升，比如 code generation 以及 Tungsten❼ 等。执行过程如下图所示



但是 DataFrame 出来后发现有些情况下 RDD 可以表达的逻辑用 DataFrame 无法表达。比如 要对 group by 或 join 后的结果用自定义的函数,可能用 SQL 是无法表达的。如下代码：

```
case class ClassData(a: String, b: Int)
case class ClassNullableData(a: String, b: Integer)
val ds = Seq(ClassData("a", 1), ClassData("a", 2)).toDS()
val agged = ds.groupByKey(d => ClassNullableData(d.a, null))
.mapGroups {
case (key, values) => key.a + values.map(_.b).sum
}
```

中间处理过程的数据是自定义的类型,并且 groupby 后的聚合逻辑也是自定义的，故用

SQL 比较难以表达，所以提出了 Dataset API。Dataset API 扩展 DataFrame API 支持静态类型和运行已经存在的 Scala 或 Java 语言的用户自定义函数。同时 Dataset 也能享受 Spark SQL 里所有性能 带来的提升。

那么后面发现 Dataset 是包含了 DataFrame 的功能，这样二者就出现了很大的冗余，故在 2.0 时将二者统一，保留 Dataset API，把 DataFrame 表示为 Dataset[Row]，即 Dataset 的子集。

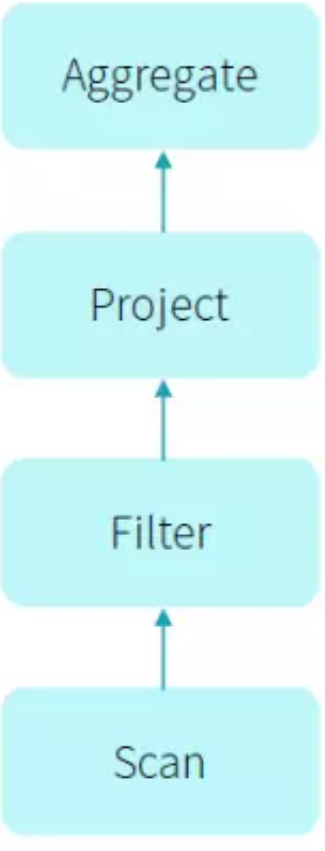
因此我们在使用 API 时，优先选择 DataFrame & Dataset，因为它的性能很好，而且以后的优化它都可以享受到，但是为了兼容早期版本的程序，RDD API 也会一直保留着。后续 Spark 上层的库将全部会用 DataFrame，比如 MLlib、Streaming、Graphx 等。

Whole-stage code generation

在参考资料 9 中有几个例子的代码比较，我们看其中一个例子：

```
select count(*) from store_sales where ss_item_sk = 1000
```

那么在翻译成计算引擎的执行计划如下图：



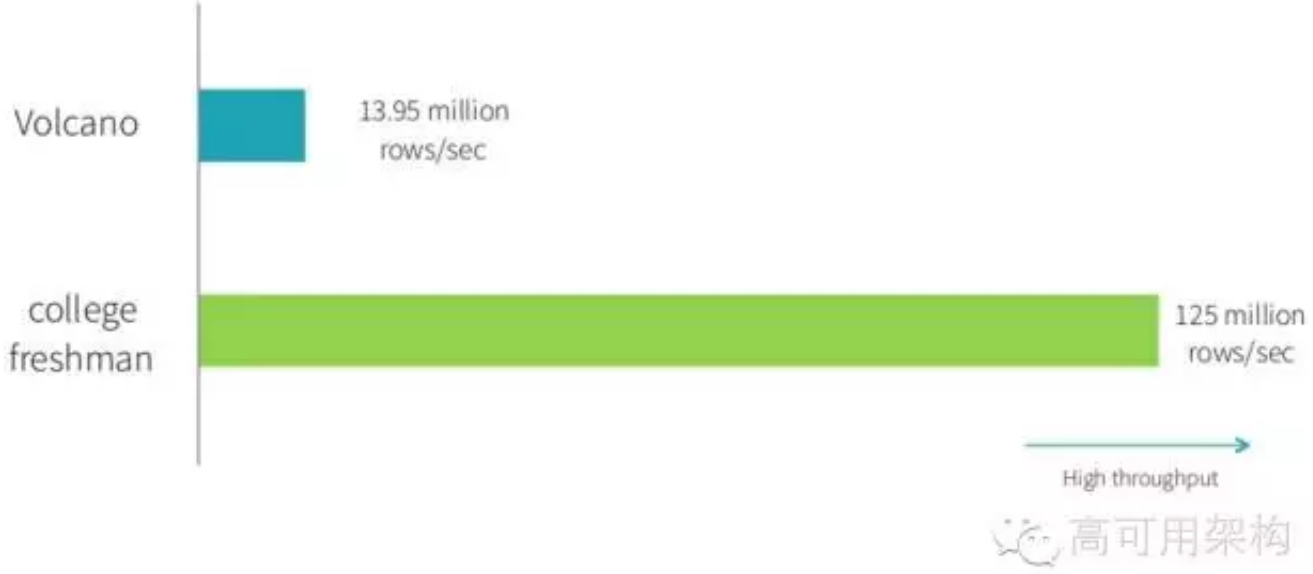
而通常物理计划的代码是这样实现的：

```
class Filter {
def next(): Boolean = {
var found = false
while (!found && child.next()) {
found = predicate(child.fetch())
}
return found
}
def fetch(): InternalRow = {
child.fetch()
}...
}
```

但是真正如果我们用 hard code 写的话，代码是这样的：

```
var count = 0
for (ss_item_sk in store_sales) {
if (ss_item_sk == 1000) {
count += 1
}
}
```

发现二者相关如下图所示：



那么如何使得计算引擎的物理执行速度能达到 hard code 的性能呢？这就提出了 whole-

stage code generation，即对物理执行的多次调用转换为代码 for 循环，类似 hard code 方式，减少中间执行的函数调用次数，当数据记录多时，这个调用次数是很大。 最后这个优化带来的性能提升如下图所示：

cost per row (single thread)

primitive	Spark 1.6	Spark 2.0
filter	15ns	1.1ns
sum w/o group	14ns	0.9ns
sum w/ group	79ns	10.7ns
hash join	115ns	4.0ns
sort (8-bit entropy)	620ns	5.3ns
sort (64-bit entropy)	620ns	40ns
sort-merge join	750ns	700ns

从 benchmark 的结果可以看出，使用了该特性后各操作的性能都有很大的提升。

Structured Streaming

Spark Streaming 是把流式计算看成一个一个的离线计算来完成流式计算，提供了一套 Dstream 的流 API，相比于其他的流式计算，Spark Streaming 的优点是容错性和吞吐量上要有优势⑩，关于 Spark Streaming 的详细设计思想和分析，可以到 https://github.com/lw-lin/CoolplaySpark 进行详细学习和了解。

在 2.0 以前的版本，用户在使用时，如果有流计算，又有离线计算，就需要用二套 API 去编写程序，一套是 RDD API，一套是 Dstream API。而且 Dstream API 在易用性上远不如 SQL 或 DataFrame。

为了真正将流式计算和离线计算在编程 API 上统一，同时也让 Streaming 作业能够享受 DataFrame/Dataset 上所带来的优势：性能提升和 API 易用，于是提出了 Structured Streaming。最后我们只需要基于 DataFrame/Dataset 可以开发离线计算和流式计算的程序，很容易使得 Spark 在 API 跟业界所说的 DataFlow 来统一离线计算和流式计算效果一

样。

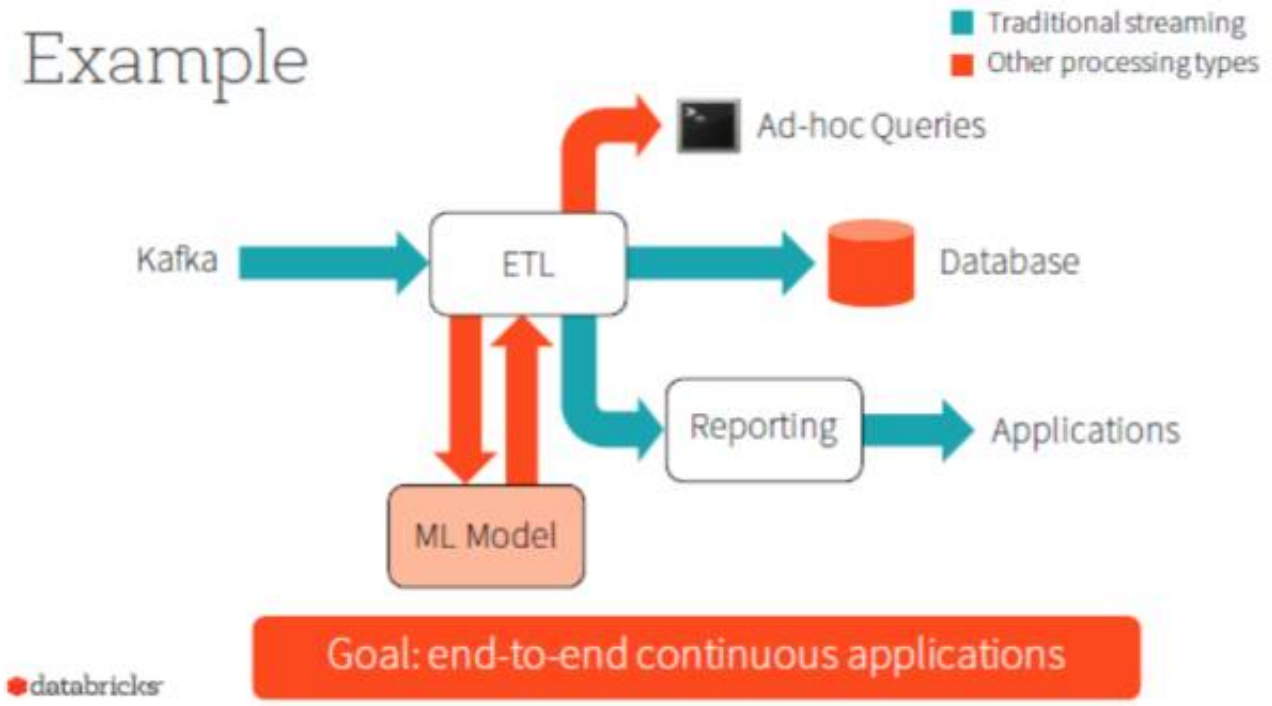
比如在做 Batch Aggregation 时我们可以写成下面的代码

```
logs  = ctx.read.format("json").open("s3://logs")

logs.groupBy(logs.user_id).agg(sum(logs.time))
    .write.format("jdbc")
    .save("jdbc:mysql//...")
```

那么对于流式计算时，我们仅仅是调用了 DataFrame/Dataset 的不同函数代码，如下：

```
logs  = ctx.read.format("json").stream("s3://logs")

logs.groupBy(logs.user_id).agg(sum(logs.time))
    .write.format("jdbc")
    .stream("jdbc:mysql//...")
```

最后，在 DataFrame/Dataset 这个 API 上可以完成如下图所示的所有应用：



其他主要性能提升

1. 采用 vectorized Parquet decoder 读取 parquet 上数据。以前是一行一行的读取，然后处理。现在改为一次读取 4096 行记录，不需要每处理一行记录去调用一次 Parquet 获取记录的方法，而是改为一批去调用一次(SPARK-12854)。加上 Parquet 本身是列式存储，这个优化使得 Parquet 读取速度提高 3 倍。
2. 采有 radix sort 来提高 sort 的性能(SPARK-14724)。在某些情况下排序性能可以提高 10-20 倍。
3. 使用 VectorizedHashmap 来代替 Java 的 HashMap 加速 group by 的执行(SPARK-14319)。
4. 将 Hive 中的 Window 函数用 Native Spark Window 实现，因为 Native Spark Window 在内存管理上有优势(SPARK-8641)。
5. 避免复杂语句中的逻辑相同部分在执行时重复计算(SPARK-13523)。
6. 压缩算法默认使用 LZ4(SPARK-12388)。

语句的增强

1. 建立新的语法解析(SPARK-12362)满足所有的 SQL 语法，这样即合并 Hive 和标准 SQL 的语句解析，同时不依赖 Hive 的语法解析 jar(SPARK-14776)。之前版本二者的语法解析是独立的，这样导致在标准 SQL 中无法使用窗口函数或者 Hive 的语法，而在使用 Hive 语法时无法使用标准 SQL 的语法，比如 In/Exists 子句等。在 SQL 编写时，

没法在一个 Context 把二者的范围全部支持，然而有了这个特性后，使得 SQL 语句表达更强大，后续要增加任何语法，只需要维护这一个语法解析即可。当然缺点是后续 Hive 版本的新语法，需要手动添加进来。

2. 支持 intersect/except(SPARK-12542)。如 select * from t1 except select * from t2 或者 select * from t1 intersect select * from t2。
3. 支持 uncorrelated scalar subquery(SPARK-13417)。如 select (select min(value) from testData where key = (select max(key) from testData) - 1)。
4. 支持 DDL/DML(SPARK-14118)。之前 DDL/DML 语句是调用 Hive 的 DDL/DML 语句命令来完成，而现在是直接在 Spark SQL 上就可以完成。
5. 支持 multi-insert(SPARK-13924)。
6. 支持 exist(SPARK-12545)和 NOT EXISTS(SPARK-10600)，如 select * from (select 1 as a union all select 2 as a) t where exists (select * from (select 1 as b) t2 where b = a and b < 2)。
7. 支持 subqueries 带有 In/Not In 子句(SPARK-4226)，如 select * from (select 1 as a union all select 2 as a) t where a in (select b as a from t2 where b < 2)。
8. 支持 select/where/having 中使用 subquery(SPARK-12543)，如 select * from t where a = (select max(b) from t2) 或 select max(a) as ma from t having ma = (select max(b) from t2)。
9. 支持 LeftSemi/LeftAnti(SPARK-14853)。
10. 支持在条件表达式 In/Not In 里使用子句(SPARK-14781)，如 select * from l where l.a in (select c from r) or l.a in (select c from r where l.b < r.d)。
11. 支持所有的 TPCDS 语句(SPARK-12540)。

与以前版本兼容(SPARK-11806)

1. 不支持运行在 Hadoop 版本 < 2.2 上(SPARK-11807)。
2. 去掉 HTTPBroadcast(SPARK-12588)。
3. 去掉 HashShuffleManager(SPARK-14667)。
4. 去掉 Akka RPC。
5. 简化与完善 accumulators and task metrics(SPARK-14626)。
6. 将 Hive 语法解析以及语法移至 Core 里(SPARK-14825)，在没有 Hive 元数据库和 Hive 依赖包时，我们可以像之前版本使用标准 SQL 一样去使用 HiveQL 语句。

1.6 版本严重问题的解决

在 http://geek.csdn.net/news/detail/70162 提到的 1.6 问题中 Spillable 集合内存溢出问题在 SPARK-4452 里已解决，BlockManager 死锁问题在 SPARK-12757 里已解决。

最后 2.0 版本还有一些其他的特性，如：

1. 用 SparkSession 替换掉原来的 SQLContext and HiveContext。
2. mllib 里的计算用 DataFrame-based API 代替以前的 RDD 计算逻辑。
3. 提供更多的 R 语言算法。
4. 默认使用 Scala 2.11 编译与运行。

参考资料

1. http://spark.apache.org/
2. https://databricks.com/blog/2016/05/11/spark-2-0-technical-preview-easier-faster-and-smarter.html
3. http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-12.pdf
4. http://www.infoq.com/cn/articles/spark-core-rdd
5. https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html
6. http://www.slideshare.net/databricks/spark-summit-eu-2015-spark-dataframes-simple-and-fast-analysis-of-structured-data
7. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html
8. https://databricks.com/blog/2016/01/04/introducing-spark-datasets.html
9. https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/6122906529858466/293651311471490/5382278320999420/latest.html
10. http://www.csdn.net/article/2014-01-28/2818282-Spark-Streaming-big-data
11. http://www.slideshare.net/rxin/the-future-of-realtime-in-spark

Q&A

1. Hive 迁移到 Spark SQL 怎么迁移？有哪些坑？

王联辉：一般 hive 执行有二种模式，一种是客户端，一种是 HiveServer 模式。

- 对于客户端命令行的话，直接把以前的 Hive 命令，改成 spark-sql 命令行 + SQL 语句即可。
- 对于 HiveServer 的方式，Spark SQL 里提供 Hive thriftServer，可以使用这种方式。

2. Spark 未来在内存管理上的优化应该是什么方向？有 RDD sharing on multiple application 计划吗？用 Spark 感觉内存调优需要花很大功夫。

王联辉：从 1.5 的 Tungsten 项目到 1.6，Spark 最大的一个亮点就是内存优化，所以 1.6 开始的版本不管数据量多大都完全没问题，而且性能也比 MR 要快几倍。

未来应该也不会在 Spark 内部去做 RDD sharing on multiple application，因为这其实是 Job server 去做的事情。1.6 的动态内存管理出来后，内存调优稍微简单化了，因为 executor 和 storage 的内存可以动态调整，当然如果用 RDD 的 API，可能内存占用会大一些，因此后续推荐 DataFrame > Dataset > RDD api。

3. Spark Streaming 在 checkpoint 后自动恢复，怎样做到 exactly once 而且恢复速度快？

王联辉：在出现故障时，Spark Streaming 本身是没办法做到完全 exactly once，这是一个事务性的问题。说白了，就是跟 Storm 做比较，Streaming 的优势是吞吐量高，也就是一段间隔内执行时间快。

为什么？首先 Storm 如果要保证消息一定要被处理的话，每一条消息都需要有一个 ack 的附加消息，这个会导致网络压力大，而且消息之间的时序性是乱的。

而在 Spark Streaming，每次是小批量数据处理，中间过程不需要 ack，一旦失败，则重新执行这个小批量数据。

具体可以看一下这篇文章：http://www.csdn.net/article/2014-01-28/2818282-Spark-Streaming-big-data

4. Spark 的应用场景有哪些？可以举些实际应用场景例子吗？

王联辉：Spark 的理念是在一个 RDD 计算框架之上可以满足各种应用，比如 HiveQL，MapReduce。除此之外，还有机器学习挖掘，流计算和图计算。

其实应该是你有什么样的计算场景，然后首先看 Spark 可不可以满足，再来考虑别的计算模式。

5. 请问搭建 Spark + HDFS 这样的集群一般硬件怎么选择？CPU，内存，磁盘……

王联辉：Spark 其实跟 HDFS 没有啥依赖关系，所以逻辑是可以独立搭建，但是物理上是在一台机器，CPU 跟内存是按比例搭配的，一般比如一个核是 4 - 6G，一台机器有 12 虚拟核的话，内存就可以配 64G，当然配 128G 内存也行，这样一个核的内存用的更多，作业跑的就更快。

6: RDD 在迭代的过程中，已经计算完成的 RDD 是何时释放？

王联辉：这个分二部分，一个是 RDD 的元信息，一个是 RDD 的数据。

- 元信息的话，当没有人引用它了，在每个 job 执行完后就会自动释放。
- RDD 的数据，非 shuffle 的数据如果被持久化了，是需要用户调用 unpersist 手动释放。
- shuffle 数据在 executor 丢失时会自动册除，这个一般 Yarn/mesos 会去完成这个工作。

7. Spark 2.0 中的新特性 Structured Streaming ，它的设计初衷是什么？使用场景是什么？

王联辉：设计初衷就是把 Spark SQL 跟 Streaming 统一，让 Streaming 任务可以用 Dataset 来表达，同时底下的执行可以用 Spark SQL 引擎。这样在易用性和性能上都有提升。

之前的 Dstreaming API 能做的事情，都可以用 Dataset 去写 Streaming 任务，最后离线计算和流式计算都用 Dataset 去编写，达到 dataflow 的效果。

8. Spark Streaming 作业在运行的过程中出现错误（比如，集群挂掉半小时后恢复），在重试 N 次后不能自动恢复运行？

王联辉：首先可以需要解决的是集群为啥会挂掉半小时，是 HDFS 还是 yarn 的问题。假设不是 Spark Streaming 的问题，那么运行中出现错误，一般在二个地方会出错，一个是 Receiver 采集数据时，一个是每个小 batch 计算时。

- 第一种情况，可能是你集群的 storage 不够了，把前面没有计算完的 block 删除了，这种一般建议将 block 存储加上磁盘，这样即时内存不够，可以刷磁盘。
- 第二种情况的话，就跟离线计算一样，是任务执行出错了，是不是数据倾斜了导致内存占用太高了，那么建议 partition 数设大一些。

9. Hive on Spark 生产环境稳定性如何？Spark Streaming 滑动窗口大于 1 小时以上的发现性能很低。大神那边是怎么处理的或者有啥好的解决方案优化方案？

王联辉：我之前没有在生产使用 Hive on Spark，所以不做回答。

第二个问题，滑动窗口大于 1 小时以上，需要看具体的业务，因为你这个窗口时间比较大，可能数据也比较大，可能会导致集群的内存无法存放，建议的话，还不如先 5 分钟计算一次，然后到一个小时再前面 5 分钟的数据做个合并。这个在计算一个小时内 distinct 值时无法满足。

10. Spark job 调度管理有啥好的方案吗？azkaban 支持度如何？还有好的方式推荐吗？

王联辉：在上层作业调度来看，其实 Spark 作业跟 MR 作业没有太大的区别，调度一个 Spark 任务跟调度一个 MR 是一样的。因为据我所知，很多公司都会自已开发一套作业管理系统。所以我对这方面的开源系统也不是很了解。

11. Spark Streaming 和 Storm 比较，各自应用场景和优缺点是什么？

王联辉：应用场景我就不回答，我觉得应该反过来问，比如我有这样的应用场景，Spark Streaming 或 Storm 哪个更优？

优缺点的话，Spark Streaming 社区活跃度要高很多，遇到问题查找答案要容易，相反 Storm 这个优势会弱一点，特别是 clojure 语言。

当然 Storm 在实时性方面比 Spark Streaming 要好，比如你要求在 1 秒或者毫秒内计算出结果，Storm 可能会比较容易做到，而 Spark Streaming 各方面的开稍使得当前可能达不到这个要求。

阅读原文