脱离JVM？ Hadoop生态圈的挣扎与演化

新世纪以来，互联网及个人终端的普及，传统行业的信息化及物联网的发展等产业变化产生了大量的数据，远远超出了单台机器能够处理的范围，分布式存储与处理成为唯一的选项。从2005年开始，Hadoop从最初Nutch项目的一部分，逐步发展成为目前最流行的大数据处理平台。Hadoop生态圈的各个项目，围绕着大数据的存储，计算，分析，展示，安全等各个方面，构建了一个完整的大数据生态系统，并有Cloudera，HortonWorks，MapR等数十家公司基于开源的Hadoop平台构建自己的商业模式，可以认为是最近十年来最成功的开源社区。

Hadoop的成功固然是由于其顺应了新世纪以来互联网技术的发展趋势，同时其基于JVM的平台开发也为Hadoop的快速发展起到了促进作用。Hadoop生态圈的项目大都基于Java，Scala，Clojure等JVM语言开发，这些语言良好的语法规范，丰富的第三方类库以及完善的工具支持，为Hadoop这样的超大型项目提供了基础支撑。同时，作为在程序员中普及率最高的语言之一，它也降低了更多程序员使用，或是参与开发Hadoop项目的门槛。同时，基于Scala开发的Spark，甚至因为项目的火热反过来极大的促进了Scala语言的推广。但是随着Hadoop平台的逐步发展，Hadoop生态圈的项目之间的竞争加剧，越来越多的Hadoop项目注意到了这些JVM语言的一些不足之处，希望通过更有效率的处理方式，提升分布式系统的执行效率与健壮性。本文主要以Spark和Flink项目为例，介绍Hadoop社区观察到的一些因为JVM语言的不足导致的问题，以及相应的解决方案与未来可能的发展方向。
注：本文假设读者对Java和Hadoop系统有基本了解。

背景

目前Hadoop生态圈共有MapReduce，Tez，Spark及Flink等分布式计算引擎，分布式计算引擎项目之间的竞争也相当激烈。MapReduce作为Hadoop平台的第一个分布式计算引擎，具有非常良好的可扩展性，Yahoo曾成功的搭建了上万台节点的MapReduce系统。但是MapReduce只支持Map和Reduce编程范式，使得复杂数据计算逻辑需要分割为多个Hadoop Job，而每个Hadoop Job都需要从HDFS读取数据，并将Job执行结果写回HDFS，所以会产生大量额外的IO开销，目前MapReduce正在逐渐被其他三个分布式计算引擎替代。Tez,Spark和Flink都支持图结构的分布式计算流，可在同一Job内支持任意复杂逻辑的计算流。Tez的抽象层次较低，用户不易直接使用，Spark与Flink都提供了抽象的分布式数据集以及可在数据集上使用的操作符，用户可以像操作Scala数据集合类似的方式在Spark/FLink中的操作分布式数据集，非常的容易上手，同时，Spark与Flink都在分布式计算引擎之上，提供了针对SQL，流处理，机器学习和图计算等特定数据处理领域的库。

随着各个项目的发展与日益成熟，通过改进分布式计算框架本身大幅提高性能的机会越来越少。同时，在当前数据中心的硬件配置中，采用了越来越多更先进的IO设备，例如SSD存储，10G甚至是40Gbps网络，IO带宽的提升非常明显，许多计算密集类型的工作负载的瓶颈已经取决于底层硬件系统的吞吐量，而不是传统上人们认为的IO带宽，而CPU和内存的利用效率，则很大程度上决定了底层硬件系统的吞吐量。所以越来越多的项目将眼光投向了JVM本身，希望通过解决JVM本身带来的一些问题，提高分布式系统的性能或是健壮性，从而增强自身的竞争力。

JVM本身作为一个各种类型应用执行的平台，其对Java对象的管理也是基于通用的处理策略

，其垃圾回收器通过估算Java对象的生命周期对Java对象进行有效率的管理。针对不同类型的应用，用户可能需要针对该类型应用的特点，配置针对性的JVM参数更有效率的管理Java对象，从而提高性能。这种JVM调优的黑魔法需要用户对应用本身以及JVM的各参数有深入的了解，极大的提高了分布式计算平台的调优门槛（例如这篇文章中对Spark的调优Tuning Java Garbage Collection for Spark Applications
）。然而类似Spark或是Flink的分布式计算框架，框架本身了解计算逻辑每个步骤的数据传输，相比于JVM垃圾回收器，其了解更多的Java对象生命周期，从而为更有效率的管理Java对象提供了可能。

JVM存在的问题

1. Java对象开销

相对于c/c++等更加接近底层的语言，Java对象的存储密度相对偏低，例如【1】，“abcd”这样简单的字符串在UTF-8编码中需要4个字节存储，但Java采用UTF-16编码存储字符串，需要8个字节存储“abcd”，同时Java对象还对象header等其他额外信息，一个4字节字符串对象，在Java中需要48字节的空间来存储。对于大部分的大数据应用，内存都是稀缺资源，更有效率的内存存储，则意味着CPU数据访问吞吐量更高，以及更少的磁盘落地可能。

2. 对象存储结构引发的cache miss

为了缓解CPU处理速度与内存访问速度的差距【2】，现代CPU数据访问一般都会有多级缓存。当从内存加载数据到缓存时，一般是以cache line为单位加载数据，所以当CPU访问的数据如果是在内存中连续存储的话，访问的效率会非常高。如果CPU要访问的数据不在当前缓存所有的cache line中，则需要从内存中加载对应的数据，这被称为一次cache miss。当cache miss非常高的时候，CPU大部分的时间都在等待数据加载，而不是真正的处理数据。Java对象并不是连续的存储在内存上，同时很多的Java数据结构的数据聚集性也不好，在Spark的性能调优中，经常能够观测到大量的cache miss。Java社区有个项目叫做Project
Valhalla，可能会部分的解决这个问题，有兴趣的可以看看这儿OpenJDK: Valhalla。

3. 大数据的垃圾回收

Java的垃圾回收机制，一直让Java开发者又爱又恨，一方面它免去了开发者自己回收资源的步骤，提高了开发效率，减少了内存泄漏的可能，另一方面，垃圾回收也是Java应用的一颗不定时炸弹，有时秒级甚至是分钟级的垃圾回收极大的影响了Java应用的性能和可用性。在当前的数据中心中，大容量的内存得到了广泛的应用，甚至出现了单台机器配置TB内存的情况，同时，大数据分析通常会遍历整个源数据集，对数据进行转换，清洗，处理等步骤。在这个过程中，会产生海量的Java对象，JVM的垃圾回收执行效率对性能有很大影响。通过JVM参数调优提高垃圾回收效率需要用户对应用和分布式计算框架以及JVM的各参数有深入的了解，而且有时候这也远远不够。

4. OOM问题

OutOfMemoryError是分布式计算框架经常会遇到的问题，当JVM中所有对象大小超过分配给JVM的内存大小时，就会fOutOfMemoryError错误，JVM崩溃，分布式框架的健壮性和性能都

会受到影响。通过JVM管理内存，同时试图解决OOM问题的应用，通常都需要检查Java对象的大小，并在某些存储Java对象特别多的数据结构中设置阈值进行控制。但是JVM并没有提供官方的检查Java对象大小的工具，第三方的工具类库可能无法准确通用的确定Java对象的大小【6】。侵入式的阈值检查也会为分布式计算框架的实现增加很多额外的业务逻辑无关的代码。

解决方案

为了解决以上提到的问题，高性能分布式计算框架通常需要以下技术：

1. 定制的序列化工具。显式内存管理的前提步骤就是序列化，将Java对象序列化成二进制数据存储在内存上（on heap或是off-heap）。通用的序列化框架，如Java默认的java.io.Serializable将Java对象以及其成员变量的所有元信息作为其序列化数据的一部分，序列化后的数据包含了所有反序列化所需的信息。这在某些场景中十分必要，但是对于Spark或是Flink这样的分布式计算框架来说，这些元数据信息可能是冗余数据。定制的序列化框架，如Hadoop的org.apache.hadoop.io.Writable，需要用户实现该接口，并自定义类的序列化和反序列化方法。这种方式效率最高，但需要用户额外的工作，不够友好。

2. 显式的内存管理。一般通用的做法是批量申请和释放内存，每个JVM实例有一个统一的内存管理器，所有的内存的申请和释放都通过该内存管理器进行。这可以避免常见的内存碎片问题，同时由于数据以二进制的方式存储，可以大大减轻垃圾回收的压力。

3. 缓存友好的数据结构和算法。只将操作相关的数据连续存储，可以最大化的利用L1/L2/L3缓存，减少Cache miss的概率，提升CPU计算的吞吐量。以排序为例，由于排序的主要操作是对Key进行对比，如果将所有排序数据的Key与Value分开，对Key连续存储，则访问Key时的Cache命中率会大大提高。

定制的序列化工具

分布式计算框架可以使用定制序列化工具的前提是要处理的数据流通常是同一类型，由于数据集对象的类型固定，对于数据集可以只保存一份对象Schema信息，节省大量的存储空间。同时，对于固定大小的类型，也可通过固定的偏移位置存取。当我们需要访问某个对象成员变量的时候，通过定制的序列化工具，并不需要反序列化整个Java对象，而是可以直接通过偏移量，只是反序列化特定的对象成员变量。如果对象的成员变量较多时，能够大大减少Java对象的创建开销，以及内存数据的拷贝大小。Spark与Flink数据集都支持任意Java或是Scala类型，通过自动生成定制序列化工具，Spark与Flink既保证了API接口对用户的友好度（不用像Hadoop那样数据类型需要继承实现org.apache.hadoop.io.Writable接口），同时也达到了和Hadoop类似的序列化效率。

Spark的序列化框架

Spark支持通用的计算框架，如Java Serialization和Kryo。其缺点之前也略有论述，总结如下：

1. 占用较多内存。Kryo相对于Java Serialization更高，它支持一种类型到Integer的映射机制，序列化时用Integer代替类型信息，但还不及定制的序列化工具效率。
2. 反序列化时，必须反序列化整个Java对象。

3. 无法直接操作序列化后的二进制数据。

Project Tungsten 提供了一种更好的解决方式，针对于DataFrame API（Spark针对结构化数据的类SQL分析API，参考Spark DataFrame Blog），由于其数据集是有固定Schema的Tuple（可大概类比为数据库中的行），序列化是针对每个Tuple存储其类型信息以及其成员的类型信息是非常浪费内存的，对于Spark来说，Tuple类型信息是全局可知的，所以其定制的序列化工具只存储Tuple的数据，如下图所示

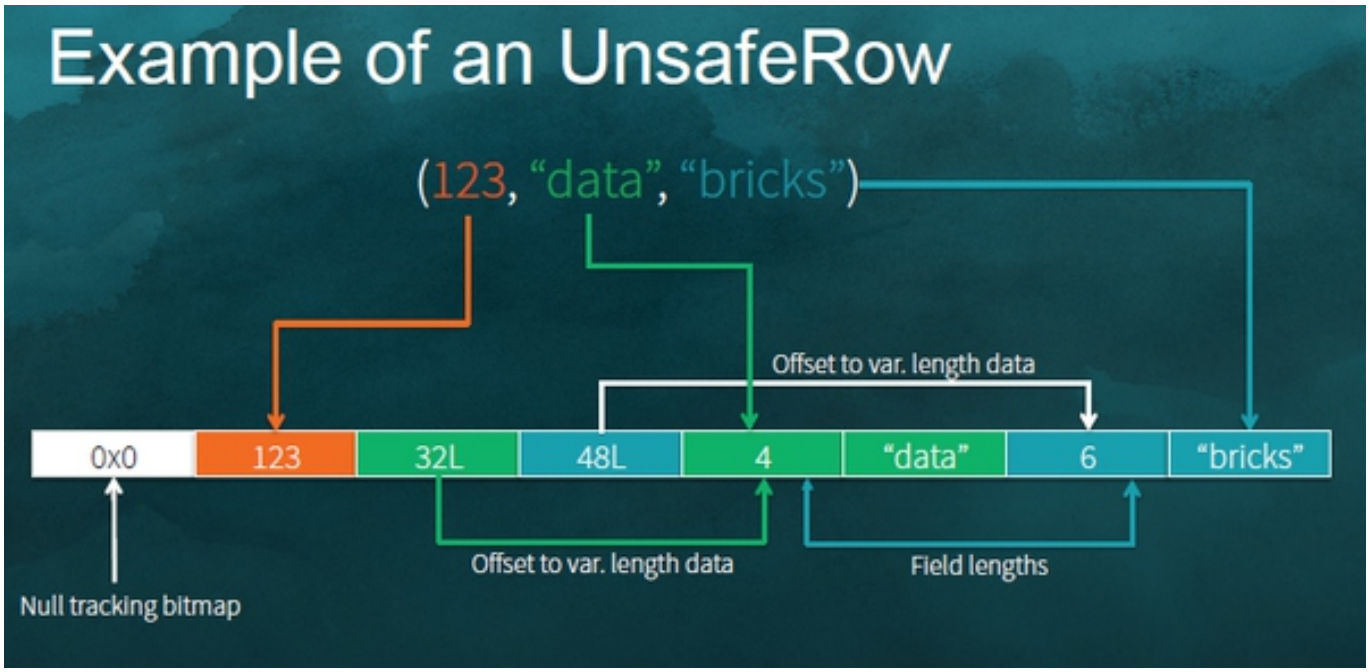

图1 Spark off-heap object layout

对于固定大小的成员，如int，long等，其按照偏移量直接内联存储。对于变长的成员，如String，其存储一个指针，指向真正的数据存储位置，并在数据存储开始处存储其长度。通过这种存储方式，保证了在反序列化时，当只需访问某一个成员时，只需根据偏移量反序列化这个成员，并不需要反序列化整个Tuple。

Project Tungsten的定制序列化工具应用在Sort，HashTable，Shuffle等很多对Spark性能影响最大的地方。比如在Shuffle阶段，定制序列化工具不仅提升了序列化的性能，而且减少了网络传输的数据量，根据DataBricks的Blog介绍，相对于Kryo，Shuffle800万复杂Tuple数据时，其性能至少提高2倍以上。此外，Project Tungsten也计划通过Code generation技术，自动生成序列化代码，将定制序列化工具推广到Spark Core层，从而使得更多的Spark应用受惠于此优化。

Flink的序列化框架

Flink在系统设计之初，就借鉴了很多传统RDBMS的设计，其中之一就是对数据集的类型信息进行分析，对于特定Schema的数据集的处理过程，进行类似RDBMS执行计划优化的优化。同时，数据集的类型信息也可以用来设计定制的序列化工具。和Spark类似，Flink支持任意的Java或是Sc

ala类型，Flink通过Java Reflection框架分析基于Java的Flink程序UDF(User Define Function)的返回类型的类型信息，通过Scala Compiler分析基于Scala的Flink程序UDF的返回类型的类型信息。类型信息由TypeInformation类表示，这个类有诸多具体实现类，例如（更多详情参考Flink官方博客Apache Flink: Juggling with Bits and Bytes）：

1. BasicTypeInfo: 任意Java基本类型（装包或未装包）和String类型。

2. BasicArrayTypeInfo: 任意Java基本类型数组（装包或未装包）和String数组。

3. WritableTypeInfo: 任意Hadoop's Writable接口的实现类.

4. TupleTypeInfo: 任意的Flink tuple类型(支持Tuple1 to Tuple25). Flink tuples是固定长度固定类型的Java Tuple实现。

5. CaseClassTypeInfo: 任意的 Scala CaseClass(包括 Scala tuples).

6. PojoTypeInfo: 任意的POJO (Java or Scala)，例如，Java对象的所有成员变量，要么是public修饰符定义，要么有getter/setter方法。

7. GenericTypeInfo: 任意无法匹配之前几种类型的类。 ）

前6种类型数据集几乎覆盖了绝大部分的Flink程序，针对前6种类型数据集，Flink皆可以自动生成对应的TypeSerializer定制序列化工具，非常有效率的对数据集进行序列化和反序列化。对于第7中类型，Flink使用Kryo进行序列化和反序列化。此外，对于可被用作Key的类型，Flink还同时自动生成TypeComparator，用来辅助直接对序列化后的二进制数据直接进行compare，hash等之类的操作。对于Tuple，CaseClass，Pojo等组合类型，Flink自动生成的TypeSerializer，TypeComparator同样是组合的，并把其成员的序列化/反序列化代理给其成员对应的TypeSerializer，TypeComparator，如下图所示：

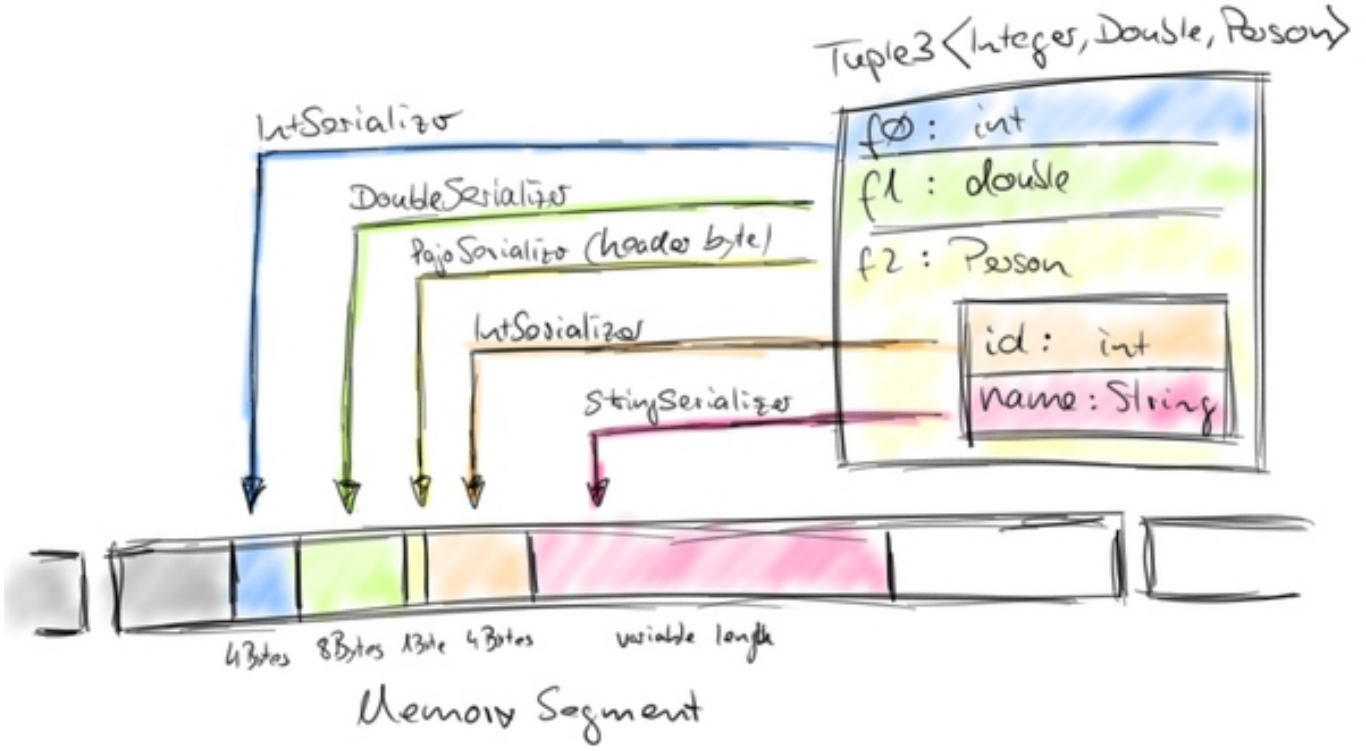

图2 Flink组合类型序列化

此外，如有需要，用户可通过集成TypeInformation接口，定制实现自己的序列化工具。

显式的内存管理

垃圾回收的JVM内存管理回避不了的问题，JDK8的G1算法改善了JVM垃圾回收的效率和可用范围，但对于大数据处理的实际环境中，还是远远不够。这也和现在分布式框架的发展趋势有冲突，越来越多的分布式计算框架希望尽可能多的将待处理的数据集放在内存中，而对于JVM垃圾回收来说，内存中Java对象越少，存活时间越短，其效率越高。通过JVM进行内存管理的话，OutOfMemoryError也是一个很难解决的问题。同时，在JVM内存管理中，Java对象有潜在的碎片化存储问题（Java对象所有信息可能不是在内存中连续存储），也有可能在所有Java对象大小没有超过JVM分配内存时，出现OutOfMemoryError问题。

Flink的内存管理

Flink将内存分为三个部分，每个部分都有不同的用途：

1. Network buffers: 一些以32KB Byte数组为单位的buffer，主要被网络模块用于数据的网络传输。

2. Memory Manager pool: 大量以32KB Byte数组为单位的内存池，所有的运行时算法（例如Sort/Shuffle/Join）都从这个内存池申请内存，并将序列化后的数据存储其中，结束后释放回内存池。

3. Remaining (Free) Heap: 主要留给UDF中用户自己创建的Java对象，由JVM管理。

Network buffers在Flink中主要基于Netty的网络传输，无需多讲。Remaining Heap用于UDF中用户自己创建的Java对象，在UDF中，用户通常是流式的处理数据，并不需要很多内存，同时Flink也不鼓励用户在UDF中缓存很多数据，因为这会引起前面提到的诸多问题。Memory Manager pool（以后以内存池代指）通常会配置为最大的一块内存，接下来会详细介绍。

在Flink中，内存池由多个MemorySegment组成，每个MemorySegment代表一块连续的内存，底层存储是byte[]，默认32KB大小。MemorySegment提供了根据偏移量访问数据的各种方法，如get/put int，long，float，double等，MemorySegment之间数据拷贝等方法，和java.nio.ByteBuffer类似。对于Flink的数据结构，通常包括多个向内存池申请的MemeorySegment，所有要存入的对象，通过TypeSerializer序列化之后，将二进制数据存储在MemorySegment中，在取出时，通过TypeSerializer反序列化。数据结构通过MemorySegment提供的set/get方法访问具体的二进制数据。

Flink这种看起来比较复杂的内存管理方式带来的好处主要有：

1. 二进制的数据存储大大提高了数据存储密度，节省了存储空间。

2. 所有的运行时数据结构和算法只能通过内存池申请内存，保证了其使用的内存大小是固定的，不会因为运行时数据结构和算法而发生OOM。而对于大部分的分布式计算框架来说，这部分由于要缓存大量数据，是最有可能导致OOM的地方。

3. 内存池虽然占据了大部分内存，但其中的MemorySegment容量较大(默认32KB)，所以内存池中的Java对象其实很少，而且一直被内存池引用，所有在垃圾回收时很快进入持久代，大大减轻了JVM垃圾回收的压力。

4. Remaining Heap的内存虽然由JVM管理，但是由于其主要用来存储用户处理的流式数据，生命周期非常短，速度很快的Minor GC就会全部回收掉，一般不会触发Full GC。

Flink当前的内存管理在最底层是基于byte[]，所以数据最终还是on-heap，最近Flink增加了off-heap的内存管理支持，将会在下一个release中正式出现。Flink off-heap的内存管理相对于on-heap的优点主要在于（更多细节，请参考Apache Flink: Off-heap Memory in Apache Flink and the curious JIT compiler）：

1. 启动分配了大内存(例如100G)的JVM很耗费时间，垃圾回收也很慢。如果采用off-heap，剩下的Network buffer和Remaining heap都会很小，垃圾回收也不用考虑MemorySegment中的Java对象了。

2. 更有效率的IO操作。在off-heap下，将MemorySegment写到磁盘或是网络，可以支持zeor-copy技术，而on-heap的话，则至少需要一次内存拷贝。

3. off-heap可用于错误恢复，比如JVM崩溃，在on-heap时，数据也随之丢失，但在off-

heap下，off-heap的数据可能还在。此外，off-heap上的数据还可以和其他程序共享。

Spark的内存管理

Spark的off-heap内存管理与Flink off-heap模式比较相似，也是通过Java UnSafe API直接访问off-heap内存，通过定制的序列化工具将序列化后的二进制数据存储与off-heap上，Spark的数据结构和算法直接访问和操作在off-heap上的二进制数据。Project Tungsten是一个正在进行中的项目，想了解具体进展可以访问：[SPARK-7075] Project Tungsten (Spark 1.5 Phase 1)， [SPARK-9697] Project Tungsten (Spark 1.6)。

缓存友好的计算

磁盘IO和网络IO之前一直被认为是Hadoop系统的瓶颈，但是随着Spark，Flink等新一代的分布式计算框架的发展，越来越多的趋势使得CPU/Memory逐渐成为瓶颈，这些趋势包括：

1. 更先进的IO硬件逐渐普及。10GB网络和SSD硬盘等已经被越来越多的数据中心使用。

2. 更高效的存储格式。Parquet，ORC等列式存储被越来越多的Hadoop项目支持，其非常高效的压缩性能大大减少了落地存储的数据量。

3. 更高效的执行计划。例如Spark DataFrame的执行计划优化器的Fliter-Push-Down优化会将过滤条件尽可能的提前，甚至提前到Parquet的数据访问层，使得在很多实际的工作负载中，并不需要很多的磁盘IO。

由于CPU处理速度和内存访问速度的差距，提升CPU的处理效率的关键在于最大化的利用L1/L2/L3/Memory，减少任何不必要的Cache miss。定制的序列化工具给Spark和Flink提供了可能，通过定制的序列化工具，Spark和Flink访问的二进制数据本身，因为占用内存较小，存储密度比较大，而且还可以在设计数据结构和算法时，尽量连续存储，减少内存碎片化对Cache命中率的影响，甚至更进一步，Spark与Flink可以将需要操作的部分数据（如排序时的Key）连续存储，而将其他部分的数据存储在其他地方，从而最大可能的提升Cache命中的概率。

Flink中的数据结构

以Flink中的排序为例，排序通常是分布式计算框架中一个非常重的操作，Flink通过特殊设计的排序算法，获得了非常好了性能，其排序算法的实现如下：

1. 将待排序的数据经过序列化后存储在两个不同的MemorySegment集中。数据全部的序列化值存放于其中一个MemorySegment集中。数据序列化后的Key和指向第一个MemorySegment集中其值的指针存放于第二个MemorySegment集中。

2. 对第二个MemorySegment集中的Key进行排序，如需交换Key位置，只需交换对应的Key+Pointer的位置，第一个MemorySegment集中的数据无需改变。 当比较两个Key大小时，TypeComparator提供了直接基于二进制数据的对比方法，无需反序列化任何数据。

3. 排序完成后，访问数据时，按照第二个MemorySegment集中Key的顺序访问，并通过Pinter值找到数据在第一个MemorySegment集中的位置，通过TypeSerializer反序列化成Java对象返回

。

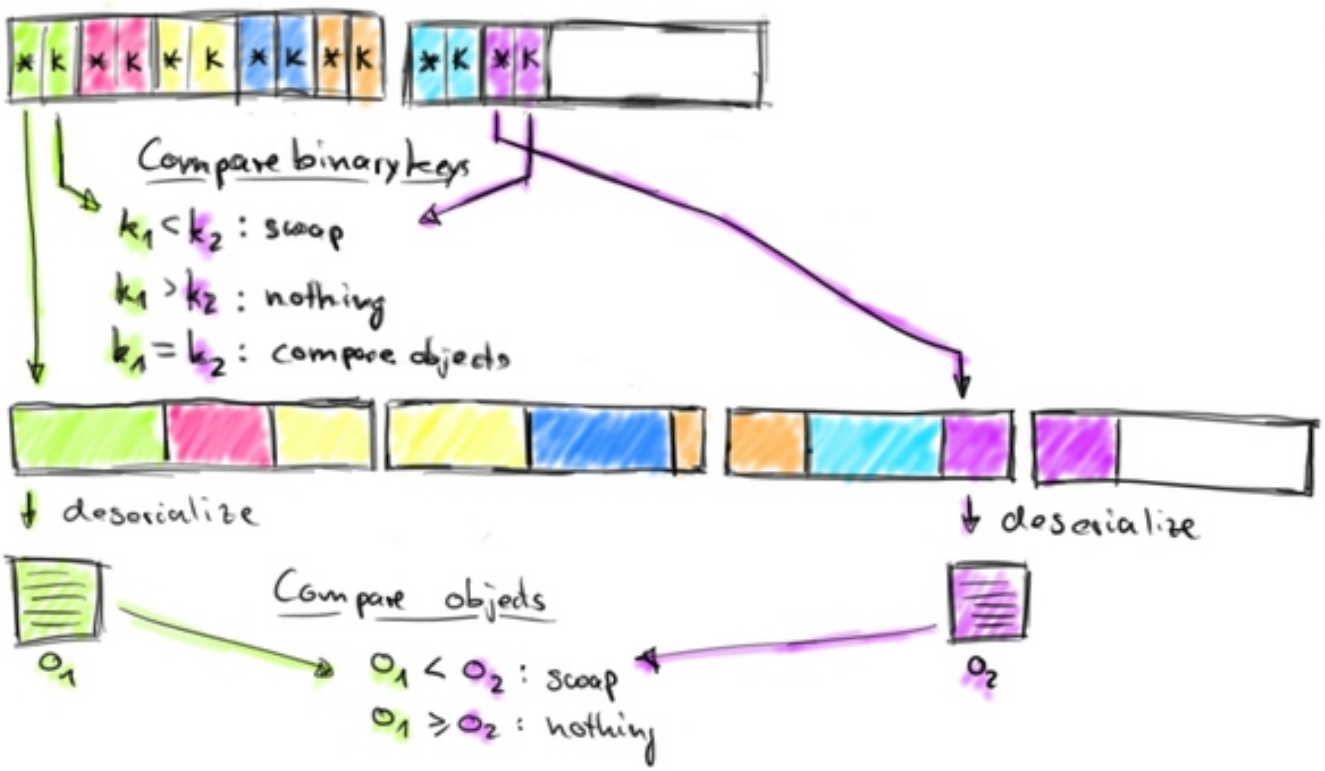

图3 Flink排序算法

这样实现的好处有：

1. 通过Key和Full data分离存储的方式，尽量将被操作的数据最小化，提高Cache命中的概率，从而提高CPU的吞吐量。

2. 移动数据时，只需移动Key+Pointer，而无须移动数据本身，大大减少了内存拷贝的数据量。

3. TypeComparator直接基于二进制数据进行操作，节省了反序列化的时间。

Spark的数据结构

Spark中基于off-
heap的排序
与Flink几乎一模一样，在
这里就不多做介绍了，感兴趣的话，请参考：https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

总结

本文主要介绍了Hadoop生态圈的一些项目遇到的一些因为JVM内存管理导致的问题，以及社区是如何应对的。基本上，以内存为中心的分布式计算框架，大都开始了部分脱离JVM，走上了自己管理内存的路线，Project Tungsten甚至更进一步，提出了通过LLVM，将部分逻辑编译成本地

代码，从而更加深入的挖掘SIMD等CPU潜力。此外，除了Spark，Flink这样的分布式计算框架，HBase（HBASE-11425），HDFS（HDFS-7844）等项目也在部分性能相关的模块通过自己管理内存来规避JVM的一些缺陷，同时提升性能。

参考

1. project tungsten：project-tungsten-bringing-spark-closer-to-bare-metal, http://www.slideshare.net/SparkSummit/deep-dive-into-project-tungsten-josh-rosen

2. The "Memory Wall": Modern Microprocessors

3. flink memory management: Apache Flink: Juggling with Bits and Bytes

4. java GC：Tuning Java Garbage Collection for Spark Applications

5. Project Valhalla: OpenJDK: Valhalla

6. java object size: dweiss/java-sizeof · GitHub

7. Big Data Performance Engineering

本文转载自：http://zhuanlan.zhihu.com/hadoop/20228397

本博客文章除特别声明，全部都是原创！
原创文章版权归过往记忆大数据（过往记忆）所有，未经许可不得转载。
本文链接:【】（）