计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2020.20200109
57(6):1179-1190,2020

基于 Spark 的大数据访存行为跨层分析工具

许丹亚[1] 王 晶[1,2] 王 利[3] 张伟功[2,3]
[1](首都师范大学信息工程学院 北京 100048)
[2](高可靠嵌入式技术北京市工程研究中心(首都师范大学) 北京 100048)
[3](北京成像理论与技术高精尖创新中心(首都师范大学) 北京 100048)
(xudanya@cnu.edu.cn)

A Cross-Layer Memory Tracing Toolkit for Big Data Application Based on Spark

Xu Danya[1], Wang Jing[1,2], Wang Li[3], and Zhang Weigong[2,3]
[1](*Information Engineering College*, *Capital Normal University*, *Beijing* 100048)
[2](*Beijing Engineering Research Center of High Reliable Embedded System* (*Capital Normal University*), *Beijing* 100048)
[3](*Beijing Advanced Innovation Center for Imaging Theory and Technology* (*Capital Normal University*), *Beijing* 100048)

Abstract Spark has been increasingly employed by industries for big data analytics recently, due to its efficient in-memory distributed programming model. Most existing optimization and analysis tool of Spark perform at either application layer or operating system layer separately, which makes Spark semantics separate from the underlying actions. For example, unknowing the impaction of operating system parameters on performance of Spark layer will lead unknowing of how to use OS parameters to tune system performance. In this paper, we propose SMTT, a new Spark memory tracing toolkit, which establishes the semantics of the upper application and the underlying physical hardware across Spark layer, JVM layer and OS layer. Based on the characteristics of Spark memory, we design the tracking scheme of execution memory and storage memory respectively. Then we analyze the Spark iterative calculation process and execution/storage memory usage by SMTT. The experiment of RDD memory assessment analysis shows our toolkit could be effectively used on performance analysis and provide guides for optimization of Spark memory system.

Key words big data; Spark; memory management; cross-layer analysis; memory tracing

摘 要 大数据时代的到来为信息处理带来了新的挑战,内存计算方式的 Spark 显著提高了数据处理的性能.Spark 的性能优化和分析可以在应用层、系统层和硬件层开展,然而现有工作都只局限在某一层,使得 Spark 语义与底层动作脱离,如操作系统参数对 Spark 应用层的性能影响的缺失将使得大量灵活的操作系统配置参数无法发挥作用.针对上述问题,设计了 Spark 存储系统分析工具 SMTT,打通了 Spark 层、JVM 层和 OS 层,建立了上层应用程序的语义与底层物理内存信息的联系.SMTT 针对 Spark 内存特点,分别设计了针对执行内存和存储内存的追踪方式.基于 SMTT 工具完成了对 Spark

收稿日期:2020-02-26;**修回日期**:2020-04-15
基金项目:国家自然科学基金项目(61772350);北京市科技新星计划(Z181100006218093);北京未来芯片技术高精尖创新中心科研基金项目(KYJJ2018008);北京市高水平教师队伍建设计划(CIT&TCD201704082);科技创新服务能力建设-基本科研业务费(科研类)(19530050173)
This work was supported by the National Natural Science Foundation of China (61772350), the Beijing Nova Program (Z181100006218093), the Research Fund from Beijing Innovation Center for Future Chips (KYJJ2018008), the Construction Plan of Beijing High-level Teacher Team (CIT&TCD201704082), and the Capacity Building for Sci-Tech Innovation Fundamental Scientific Research Funds (19530050173).
通信作者:王晶(jwang@cnu.edu.cn)

迭代计算过程内存使用,以及跨越 Spark,JVM 和 OS 层的执行/存储内存使用过程的分析,并以 RDD 为例通过 SMTT 分析了单节点和多节点情况下 Spark 中读和写操作比例,结果表明该工作为 Spark 内存系统的性能分析和优化提供了有力的支持.

关键词 大数据;Spark;内存管理;跨层分析;内存追踪

中图法分类号 TP391

大数据时代的到来,为信息处理技术带来了新的挑战.大数据时代的信息具有数据量大、数据类型多、增长速度快、价值密度低等特点[1].MapReduce 有效地解决了海量数据处理的扩展性问题[2],然而 MapReduce 在处理过程中,将数据和中间结果以文件的形式存放到磁盘上,例如 Map 阶段的计算结果,频繁地访问磁盘,导致磁盘的压力成为计算中的性能瓶颈[3].对此,Spark 以内存计算的方式进行了改进,即数据的存储和处理都位于内存之中.目前 Spark 已经得到了工业界的广泛应用,腾讯、百度、淘宝、亚马逊、网易等企业都在使用 Spark 构建自己的大数据业务[4].内存计算显著地提高了程序的运行速度,但同时也对内存系统造成了巨大的压力,使得内存系统的性能成为影响数据处理速度的瓶颈[5-6].

传统的计算机体系结构和存储系统设计已经无法适应处理大数据的要求.内存计算方式的 Spark 采用了"统一内存管理"模型,将存储系统化分为 Spark 层、JVM 层、操作系统(OS)层和硬件层,完成数据的传输、元数据的管理,以及内存和磁盘等硬件的数据管理.层次化的存储系统给设计和维护带来了便利.然而,上层应用程序对系统内存、磁盘、CPU、网络等资源敏感,需要底层优化以达到性能最优.因此理解底层系统行为与上层应用特征之间的关系,对于优化系统性能十分有意义,而此时层次化设计中底层对上层透明的工作方式给系统优化带来了困难.例如,位于硬件和软件交界面的操作系统从应用层角度常被作为黑盒对待,丰富灵活的系统配置参数对普通用户难以准确理解和使用,进而导致其对上层应用程序的影响不得而知,陷入"不知道"与"不使用"的循环.因此,跨层的理解 Spark 系统的行为是十分重要且必要的.然而,计算机体系结构中"高内聚、低耦合"的分层设计,使得 Spark 系统中跨层的分析存在 4 个挑战:

1) 在 Spark 系统中,Spark 应用层、JVM 层和 OS 层每一层都有自己的内存管理机制和本层特有的语义.从图 1 Spark 系统中各层内存对象的语义及各层语义转换流程可以看出,每层的语义对外暴露较少,因此如何在不破坏其原有工作方式的同时获取其内存对象是 Spark 分析的挑战之一.

2) 现有的性能分析工具大多只工作在某一层,尽管我们可以在每一层利用各种工具来获取程序行为、抓取性能指标,但无法将每一层的结果相互对应起来进行分析,Spark 语义与底层动作脱离,在 OS 层分析 Spark 性能时如何感知 Spark 语义是 Spark 分析的挑战之二.

3) OS 层性能表现与上层应用程序特征并不是

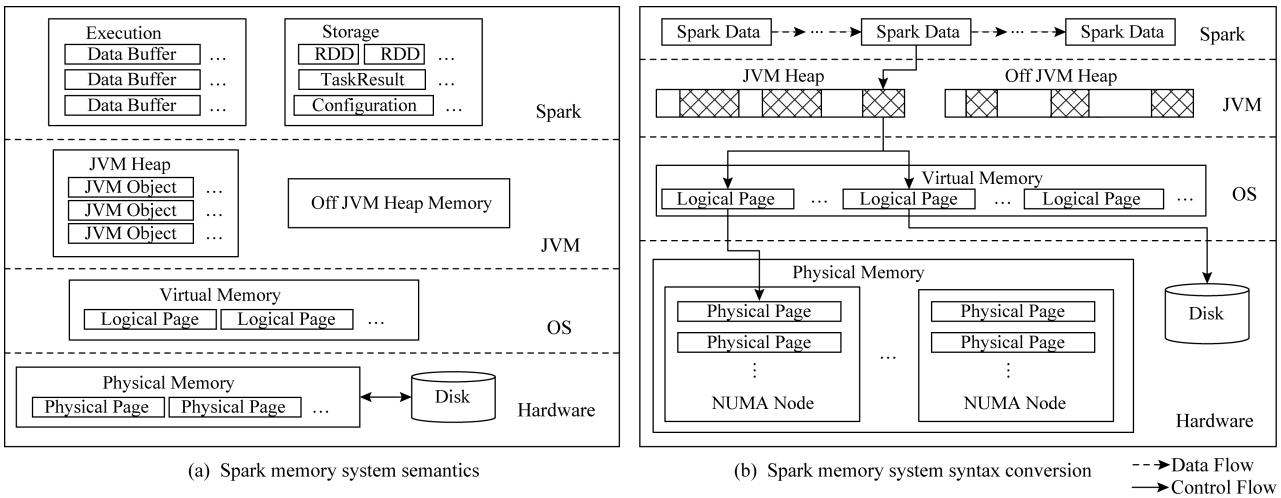

Fig. 1 Spark memory system semantics and syntax conversion

图 1 Spark 系统中各层内存对象的语义及各层语义转换流程

清晰可见的,我们在 OS 层观测到的性能指标,除了体现 Spark 应用程序特征之外,还包含其他因素的影响,如 Spark 框架、JVM 自身运行引发的开销,如何排除逻辑上不相关的参数通过其他机制产生的间接影响是 Spark 分析和实验的挑战之三.

4) 物理存储对上层应用的性能存在影响.当前数据中心里的商用服务器内存通常较大,很多已达到 128 GB 以上.在这类场景下,为了提高服务器内存访问的吞吐率,通常采用多内存条结构,即多个内存条均匀地分布在多个 CPU 内存控制器上,尽可能地利用所有 CPU 的内存控制器.但在对上层类似 Spark 自己管理内存的应用,通常仅利用虚拟地址进行内存管理,无法感知底层的内存分布,容易造成 NUMA 架构下的内存访问不均衡(CPU 访问非本地内存会有较大的时延;多 CPU 同时访问同一个内存通道也有性能瓶颈),从而影响性能.并且上层应用对虚拟地址以外的内存地址无法感知,也没有现有的工具建立 Spark 存储对象的关联关系,因此如何对 Spark 物理页进行追踪是性能分析工具需要解决的挑战之四.

为了解决上述问题,本文设计了跨层的 Spark 内存追踪工具 SMTT(Spark memory tracing toolkit).SMTT 垂直打通了 Spark 层、JVM 层和 OS 层,将上层应用程序的语义与底层物理内存信息建立了联系,从而有助于对应用程序的访存行为进行跟踪和分析.

在 Spark 运行期间,SMTT 会在 Spark 应用层抓取到 Spark 应用程序对数据的访问序列,并记录每层的使用信息.SMTT 针对 Spark 系统中特有的执行内存和存储内存分别设计了不同的追踪方案.对于执行内存,我们将 RDD(resilient distributed datasets)与其对应的虚拟内存地址联系起来.对于存储内存,我们逐层剥去封装数据的外层数据结构,获取存放数据的真正虚拟地址,并将虚拟地址与数据在 Spark 应用层的语义对应起来,从而有效地分析 Spark 的分布式执行和 RDD 的存储访问等重要信息.

Spark 一直处于快速发展之中,伴随着频繁的版本迭代,很多重要的特性也发生了变化.然而,目前为止,本文所关注的“统一内存管理”和“迭代计算”并未发生根本性的改变,因此,本文的研究方法对于较新版本的 Spark 仍有借鉴意义.SMTT 能够提供内存对象的各个层次对应关系,为系统优化提供支持,例如通过 SMTT 得到的物理页存储关系,能够为基于 NUMA 感知的内存调度等优化策略提供指导,实现对分布式内存结构的高效利用,为 Spark 的性能分析和优化奠定了基础.

1 相关工作

针对 Spark 和 Hadoop 的对比分析无论是从日志文件角度[7]、从页排序和分类算法的角度,还是词频统计算法方面[8-9],结果都表明面向云负载,内存计算方式的 Spark 优于 Hadoop.随着 Spark 的广泛应用,其性能优化问题成为研究人员普遍关注的热点.Spark 性能可以通过配置 Spark 参数、重构 Spark 应用代码、优化 JVM 等手段,从调度与分区、内存存储、网络传输、序列化与压缩等方面优化[4,10].然而,无论哪种优化方案,都需要首先对 Spark 行为特征进行分析.如基于 Spark 的弹性分布式数据集设计首先分析了 Spark 屏幕终端日志在迭代算法和迭代数据挖掘应用场景下的效率问题,观察到内存数据管理和存放策略对性能的影响,从而提出了基于共享内存和粗粒度交易的容错系统设计[11].将 Spark 应用程序运行的历史数据建立成数据库,根据对历史数据的分析来自动化配置 Spark 参数,能够大大降低调优的门槛,同时也使调优的效果变得更为可信[12].Spark SQL 分析了应用层的 API 接口、基本操作行为和数据模型,并基于分析提出了新的 Apach Spark 集成模型[13].江涛等人分析了典型算法 Spark 实现的执行时间、每秒钟磁盘读写次数、内存带宽、内存页访问频率等[6].现有性能分析工作对于我们认识 Spark 负载的性能特征具有很大的参考价值,但这些研究大多在系统单一层面开展,缺乏全面、有效和系统的分析方案以及分析工具.

当前,云服务器面临着严峻的存储墙问题,对访存行为的分析和追踪是优化存储系统的重要基础.现有的内存追踪工具分别从操作系统层、应用层和硬件层展开分析.

1) 操作系统层分析工具.perf[14]是广泛使用的系统性能分析工具,通过它应用程序可以利用 PMU,tracepoint 和内核中的特殊计数器来进行性能统计,但 perf 只能看到操作系统层面的行为,无法感知上层逻辑.Simics[15]和 QEMU[16]等软件模拟器能够生成访存踪迹并评测存储系统性能,但现有模拟器大多只能支持桌面应用,难以模拟 Spark 和 Hadoop 这类复杂系统的行为.HMTT 可以跟踪到物理内存的访问轨迹和对应的程序语义[17-18].然而这里的“程序”仅限于 OS 层面的程序,无法感知

Spark 层的语义，如 RDD、分区等，因此不适用于 Spark 的应用场景.

2) 应用层分析工具.Pin[19] 和 Valgrind[20] 等二进制插桩工具能够分析应用程序的虚拟地址，但难以对内核插桩，无法分析内核层面的信息.Spark UI[21] 是针对 Spark 的应用层分析工具，可以跟踪 Spark 的执行情况，同样无法看到其引起的底层动作.由于 Spark 语义与底层动作的关联缺失，无法完整地看到一个 Spark 程序执行的来龙去脉，更无法解释什么样的程序会引起什么样的系统性能变化.应用层的分析工具都把操作系统看成了黑盒、忽略了它的可配置性，导致数以千计的内核参数并没有发挥其该有的作用.

3) 硬件层分析工具.Awan 等人通过双端口服务器的硬件计数器分析了批处理和流处理负载的体系结构特征，发现了同时多线程、Cache 预取和 NUMA 结构对于性能的影响[22].这类基于硬件计数器的方法只能看到系统中的一部分硬件事件，无法对所有访存行为进行追踪.

针对上述单一层面分析工具的问题，本文设计了打通 Spark，JVM，OS 三层的内存跟踪工具 SMTT，以满足跨层内存跟踪的需要.SMTT 从 OS 层对 Spark 应用程序进行性能分析，在不丢失 Spark 语义的情况下，分析 Spark 负载的行为特征与底层系统性能之间的关系，为 Spark 应用的调优提供依据.此外，SMTT 在执行节点层对访存进行追踪，能够为 Spark Streaming[23] 和 Spark SQL[12] 等上层框架提供细粒度的内存追踪结果.

2 SMTT 整体设计

内存计算是 Spark 的重要特征，即数据的存储和处理都在内存之中.与 JVM 层一样，Spark 层也有着自己的内存管理机制.Spark 层采用统一内存管理模型，将所管理的内存分为保留内存、用户内存和 Spark 内存这 3 部分[24].保留内存由系统保留，对 Spark 应用程序是透明的，其中存放了许多 Spark 内部对象.用户内存由 Spark 应用程序使用，比如：应用程序如果在内存中保留了大量用户数据，那么将从这一部分中分配空间用于存储.Spark 在集群层次的主要工作是进行任务调度，其具体的对象级内存管理过程属于节点级，主要体现在 executor 节点上的内存管理.因此，本文针对目标为 Spark 任务执行节点内存状态进行追踪，对应 Spark 整体集群的内存管理状态，可以通过对单节点采集信息的聚合得到.Spark 内存由 Spark 层进行管理，按照 Spark 的工作方式来存储应用程序数据.Spark 内存又分为存储内存和执行内存 2 部分.其中，存储内存用于存储持久化的 RDD 数据、广播变量等；执行内存用于存储运行期间的中间数据，如洗牌过程中在映射端的缓存.

Spark 所管理的内存可以来自 JVM 堆，也可以直接从操作系统获得.对 Spark 应用进行性能分析时，Spark 层、JVM 层和 OS 层都有追踪工具可以利用，但无法将各层追踪到的结果建立联系.例如，Spark 层的评测系统可以看到被存储的 RDD 以及它们的大小，但无法知道它们位于内存何处；JVM 工具可以看到当前堆的状态，却无法知道堆内具体存储什么数据对象，导致我们无法讨论上层应用程序行为与系统性能的关系；OS 层无法跨越 JVM 层将性能指标与 Spark 应用程序行为建立联系，导致底层观测丢失了 Spark 语义.尽管 JVM 层也有对象的创建、回收和复用机制，但 Spark 的数据有很大一部分不来源于 JVM 堆，对于这类数据 JVM 分析工具无能为力.SMTT 将 Spark 语义延伸至硬件，整体架构如图 2 所示.逻辑上 Spark 分为计算引擎和存

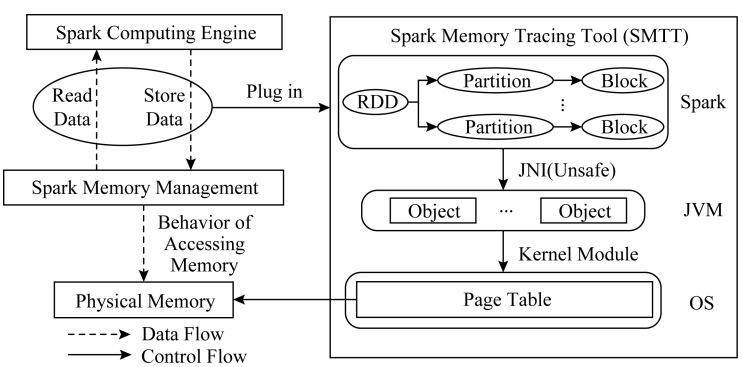

Fig. 2 Framework of SMTT

图 2 SMTT 整体架构

储系统 2 部分.计算引擎负责对数据的处理;存储系统负责从底层(JVM 或 OS)获取内存,对计算过程中的内存进行动态地分配和回收.代码层面上,Spark 存储系统相对独立,统一封装对外的接口.计算引擎在执行过程中调用存储系统的接口来分配和释放内存.

SMTT 首先在不破坏原有执行逻辑的情况下对这些接口进行了代码插桩,获取当前操作的数据信息、数据所在的数据结构和数据访问序列.其中,代码插桩只监听 Spark 的数据结构,不改变其固有的接口与逻辑,因此在修改源码和重新编译后不影响 Spark 与其他组件,如 Yarn 和 Flink 等的集成.然后,SMTT 分别对 JVM 堆内和堆外的数据进行处理,获得虚拟地址:对于从 JVM 堆内获取的内存,找到数据所对应的 JVM 对象,并将对象转换为 OS 层的虚拟地址.对于从 JVM 堆外获取的内存,SMTT 找到起始地址,此时这个起始地址直接就是 OS 层的虚拟地址.最后,SMTT 将虚拟地址转换成物理地址,并获得物理页号、是否被交换到外存、所在 NUMA 节点等数据在物理硬件上的分布信息.经过这样的流程,Spark 数据在物理硬件上的分布一目了然,便于从性能的根源分析 Spark 应用程序的数据访问特性与系统性能的关系.

SMTT 按照数据被访问的顺序给出访问序列,记录 Spark 语义与底层地址信息的对应关系,其结果格式如图 3 所示,包括:

1) 访问时间.何时发起的本次访问.

2) 访问类型.对数据的访问是读还是写.

3) Spark 语义.被访问数据在 Spark 应用层的意义,如所属 RDD 和分区.

4) 虚拟地址.被访问数据在 OS 中分配的虚拟地址.

5) 虚拟页信息.被访问数据属于哪些虚拟页、数据是否跨页了、页是在内存中还是被交换到了外存等影响性能的重要因素.

6) 物理地址.被访问数据在物理内存中的地址.

7) 物理页信息.被访问数据所属物理页、物理页的热度,以及在 NUMA 系统中该物理页是否跨节点等.

由于执行内存和存储内存的用途不同、工作原理不同,它们在 Spark 源码中的存取接口也完全不一样.因此,SMTT 针对这 2 种内存分别设计了追踪方案.

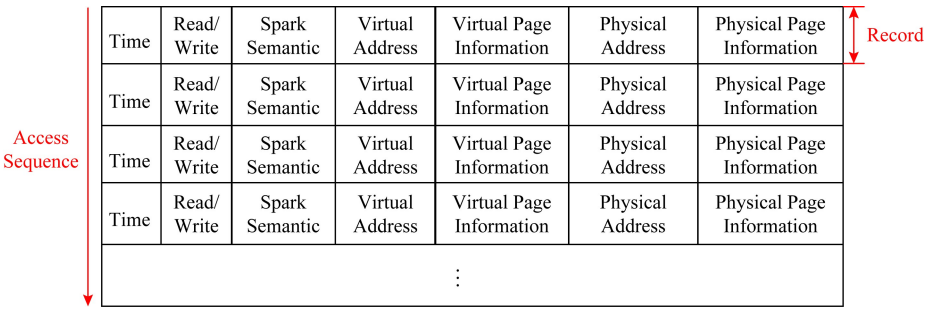

Fig. 3 Output formation of SMTT

图 3 SMTT 结果格式

2.1 执行内存追踪方案

执行内存在 Spark 任务执行期间按需分配,例如,当数据需要排序、合并等整理操作时,系统会在执行内存中分配空间进行处理,使用之后立即释放.

对于 RDD 中的每一个分区,Spark 会启动一个 Task 线程处理其中的数据.但是,Task 只包含 RDD 信息,却无法获得该 RDD 内数据对应的虚拟内存地址;Sorter 中包括虚拟内存地址,但无法获得该地址中存放的数据所对应的 RDD 信息.为了打通各部分的语义,我们设计了如图 4 所示的针对执行内存的追踪方案,其中箭头表示追踪流程:

1) 在 Task 中,将所用 Writer 的 Hash 码(Hashcode)和 RDD 信息传给 SMTT,SMTT 将其写入一张以 Writer 的 Hash 码为键(key)、RDD 信息为值(value)的 Hash 表;

2) 在 Writer 中,将所用 Sorter 的 Hash 码和当前 Writer 的 Hash 码传给 SMTT,SMTT 将其写入一张以 Sorter 的 Hash 码为键、Writer 的 Hash 码为值的 Hash 表;

3) 在 Sorter 中,将当前 Sorter 的 Hash 码,以及数据申请到的虚拟内存地址传给 SMTT;

4) 根据 Sorter 的 Hash 码找到 Writer 的 Hash

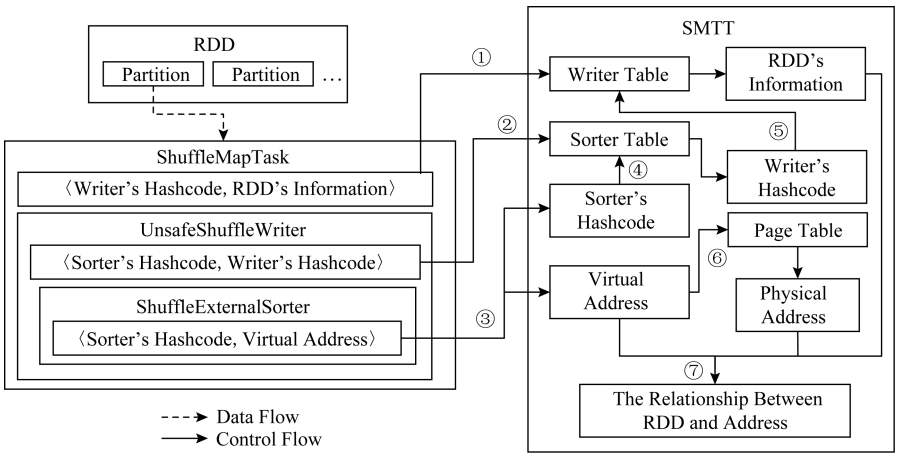

Fig. 4 Tracing scheme of execution memory

图 4 执行内存的追踪方案

码,根据 Writer 的 Hash 码找到 RDD 信息;

5) 通过访问 OS 页表,根据传进来的虚拟内存地址得到对应的物理内存信息;

6) 将 RDD 信息、虚拟地址、物理地址信息对应起来,作为一条记录保存至文件.

7) 缓存中的数据和溢出文件中的数据会被合并到一个文件中.

通过上述步骤,我们将 Spark 应用层 RDD 的信息,同 JVM 层的信息,与底层物理内存信息联系起来,获得了一一对应的关系,实现了任意层观测到的信息与其他层特征和行为之间的因果关系分析.

2.2 存储内存追踪方案

Spark 迭代计算节约了内存,但牺牲了时间.为了改善性能,Spark 提供了缓存机制:被缓存在内存中的 RDD 数据,后续被访问时可直接从内存中获取,而无需从头计算,缓存 RDD 的内存来自于存储内存.对于持久化到内存中的 RDD,Spark 的 MemoryStore 对象提供了统一的存/取接口.其内部维护了一个以 Spark 的 BlockID 对象为键、以 MemoryEntry 对象为值的 Hash 表.其中 BlockID 对象的值就是 RDD 的 ID 与分区的 ID 按照特定格式的组合.MemoryEntry 对象用于描述被存储的数据,其内部有一个 Java 对象 ByteBuffer 数组,而每一个 ByteBuffer 对象内部有一个存放数据的字节数组.

对于存储内存的追踪,我们主要解决的问题是如何逐层剥去封装数据的外层数据结构,获得虚拟

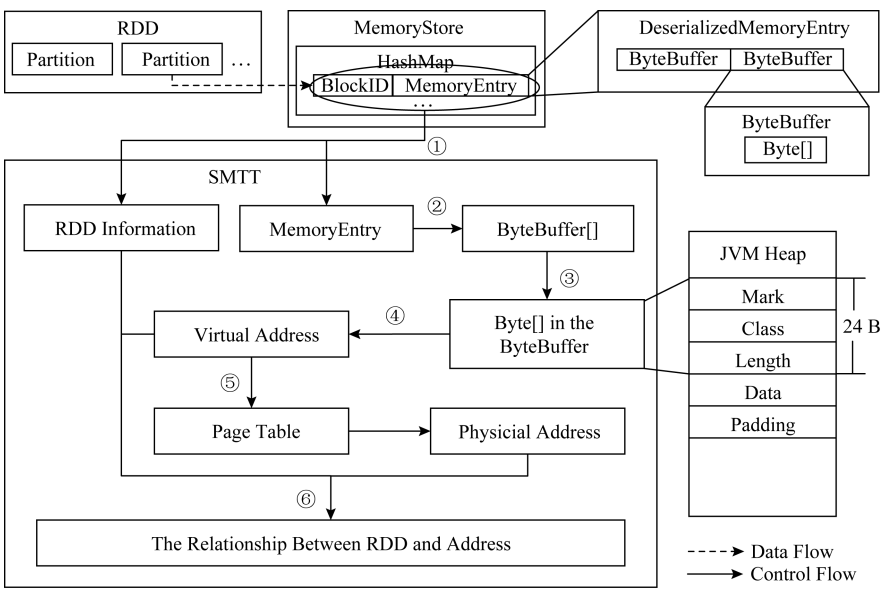

Fig. 5 Tracing scheme of storage memory

图 5 存储内存的追踪方案

地址,并将其与数据在 Spark 层的语义对应起来.对此,我们的设计方案如图 5 所示,箭头表示追踪流程:

1) 在 MemoryStore 对象内部,当对 Hash 表进行存/取时,将此时的 BlockID 对象和 MemoryEntry 对象传给 SMTT;

2) SMTT 获取 MemoryEntry 对象的成员变量 ByteBuffer 数组;

3) 对于每一个 ByteBuffer 实例,通过 Unsafe 对象提供的接口获取其中的成员变量字节数组在 JVM 堆中的起始地址;

4) 获得字节数组内数据的起始地址,在不使用指针压缩的情况下,数组在 JVM 堆中前 24 B 为头信息,其后紧跟着数据,所以根据头信息的长度获得数据的虚拟地址,然后使用 JNI 技术直接访问内存地址;

5) 通过访问操作系统页表,将虚拟地址转换成物理地址;

6) 将传入 SMTT 的 RDD 信息、数据虚拟地址信息、物理地址信息对应起来,保存至文件.

通过上述步骤,完成了存储内存中不同层语义之间的关联,为访存行为分析建立了通路.

3 基于 SMTT 的访存行为分析

SMTT 给出了跨层的追踪方式,建立了不同层的语义.本节借助于 SMTT 工具对 Spark 的迭代计算过程中内存的访问过程,以及执行内存和存储内存的使用过程进行详细的跟踪,解释了 Spark 中 RDD 的数据对象从应用层开始,经 JVM 层和 OS 层对应到物理内存的流程.

3.1 Spark 计算过程追踪

本节以统计词频的 Spark 应用程序为例,分析其分布式计算的完整过程.如图 6 所示,一个 RDD 访问完整过程可视为一个有向无环图,分成 2 个阶段:第 1 个阶段 Stage0 完成 Shuffle 写,即 Map 映射;第 2 个阶段 Stage1 完成 Shuffle 读,即 Reduce 化简.每个阶段的执行起点是当前阶段的最后一个 RDD,这个 RDD 的每一个分区会交给一个 Spark 创建的独立 Task 线程进行处理.Task 会调用最后一个 RDD(即 RDD3)的 *iterator*()方法获得其负责处理的分区数据的迭代器,在 RDD 没有被持久化的情况下,这个 *iterator*()方法会调用当前 RDD 的 *compute*()方法.这样一直递归调用到第 1 个 RDD(即 RDD0)的 *compute*()方法,这时的 *compute*()会返回一个文本数据的迭代器给它的下一个 RDD(即 RDD1),下一个 RDD 会新创建一个迭代器对象 Iter1,并重写其中的 *next*()方法,用自身生成时的转换函数 *f*0 包装前一个 RDD 传过来的迭代器 Iter0 的 *next*()方法,然后将这个新创建的迭代器返回给它的下一个 RDD;它的下一个 RDD 还会采取相同的操作,一直到最后一个 RDD(即 RDD3)的 *compute*()返回给 Task.此时 Task 拿到迭代器之后,每调用一次 *next*()方法,就会从文件系统读取一条记录并经过逐层转换函数的加工返回给它.

值得注意的是,第 1 个 RDD 从文件系统中读取

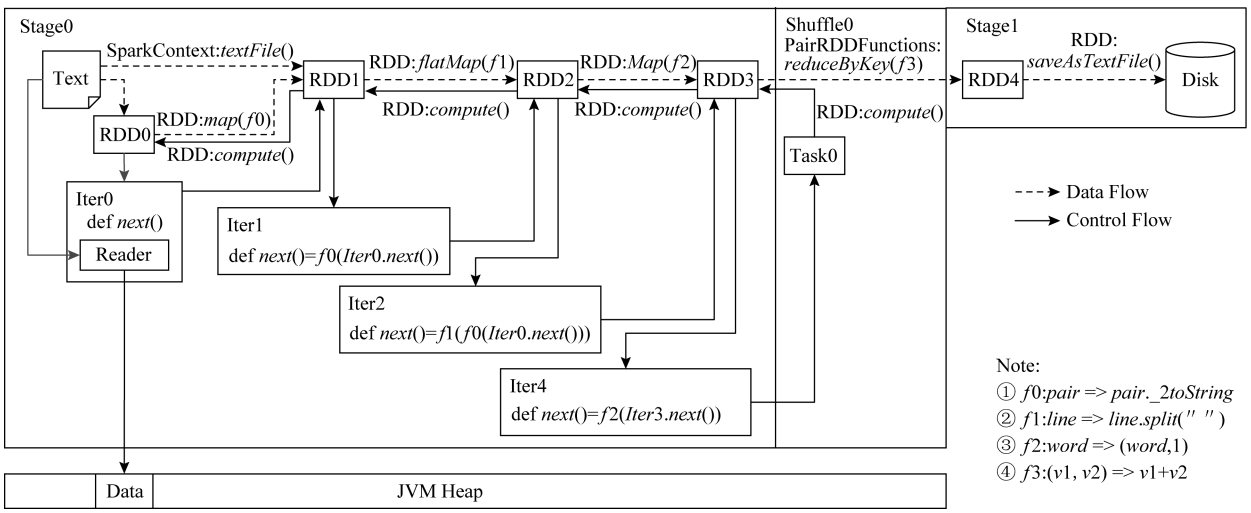

Fig. 6 Distributed computing process of Spark for word count algorithm

图 6 词频统计算法中 Spark 分布式计算的过程

数据的时候使用了一块缓存,即图 6 中 JVM 堆中的 Data 部分,这块缓存在迭代器后续的数据读取过程中是被重复使用的.从上述的迭代过程可以看出,如果不做持久化,RDD 的数据是不会全部存在内存中.这就解释了,尽管一个 Task 需要处理大量的数据,但其实际内存占用量并不大的原因.有效的复用提高了存储空间的利用率.

3.2 执行内存使用过程追踪

在 Spark 中执行内存通常在洗牌过程以 Unsafe 方式、Sort 方式或 Bypass 方式使用.图 7 描述了使用 SMTT 工具对 Unsafe 方式和 Sort 方式的工作过程进行追踪的过程,其中①～⑦表示当前任务通过迭代计算获取它所负责分区的数据迭代器,然后将数据逐条写到执行内存分配的缓存空间中,然后进行排序、合并等操作.每向缓存中插入一条记录,当前任务都需要判断缓存是否够大,如果不够则通过 *acquirExecutionMemory*()方法向 Spark 申请新的执行内存空间,如图 7 中的步骤⑧⑨所示.写缓存期间,如果所占内存超过一定阈值,当前任务便将数据写回到硬盘文件,这个过程称为溢出(Spill).最后,步骤⑩缓存中的数据和溢出文件中的数据会被合并到一个文件中以供 Stage2 读取.对于 Bypass 方式,与 Unsafe 方式和 Sort 方式不同之处在于,不将数据写到缓存,而是直接写到硬盘文件.

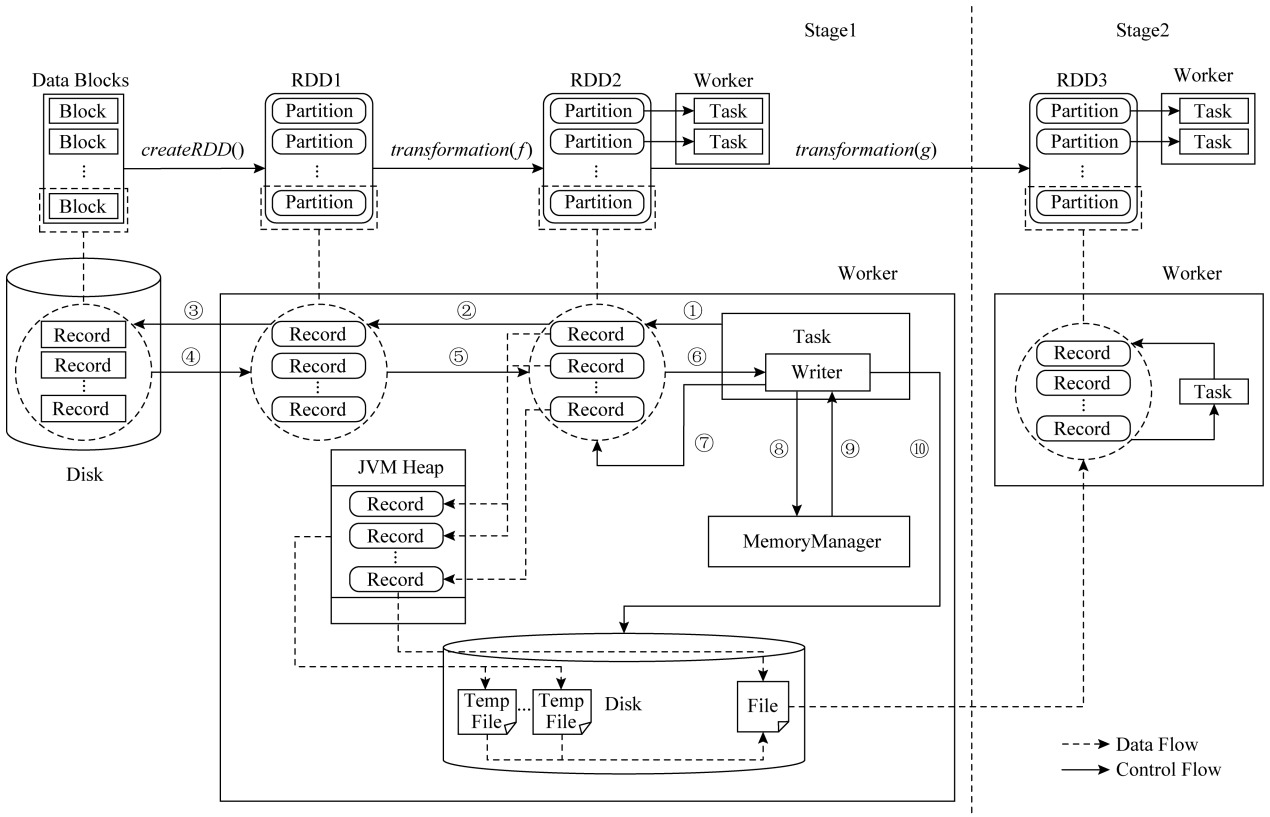

Fig. 7 Memory access process of execution memory in Unsafe mode and Sort mode

图 7 Unsafe 和 Sort 方式中执行内存使用过程

3.3 存储内存使用过程追踪

我们使用 SMTT 对 Spark 中存储内存的使用方式进行了追踪和分析,如图 8 的步骤①～⑫表示 Task 递归获取迭代器的过程.存储内存的使用过程跟执行内存相似,区别是从步骤⑦开始,被存储的 RDD(RDD2)的迭代器不返回给它的下一个 RDD,而是交给 BlockManager 对象.随后,BlockManager 对象使用获得的迭代器将数据逐条存入到内存.每向内存中写入一条记录,Task 都会检查内存是否够用,如果不够则通过 *acquirStorageMemory*()方法向 Spark 申请新的存储内存空间.在全部数据写入内存之后,被持久化的 RDD 会创建一个遍历这块内存的迭代器并返回给下一个 RDD,从而完成 Task 递归地获取迭代器的过程.由此可见,访问持久化的 RDD 数据实际上访问的是内存的某一段空间.

存储内存除了存放持久化的 RDD 数据之外,还存放反序列化过程中“unroll”过程的数据,以及用于全局的广播变量等.无论何种数据,存储内存同

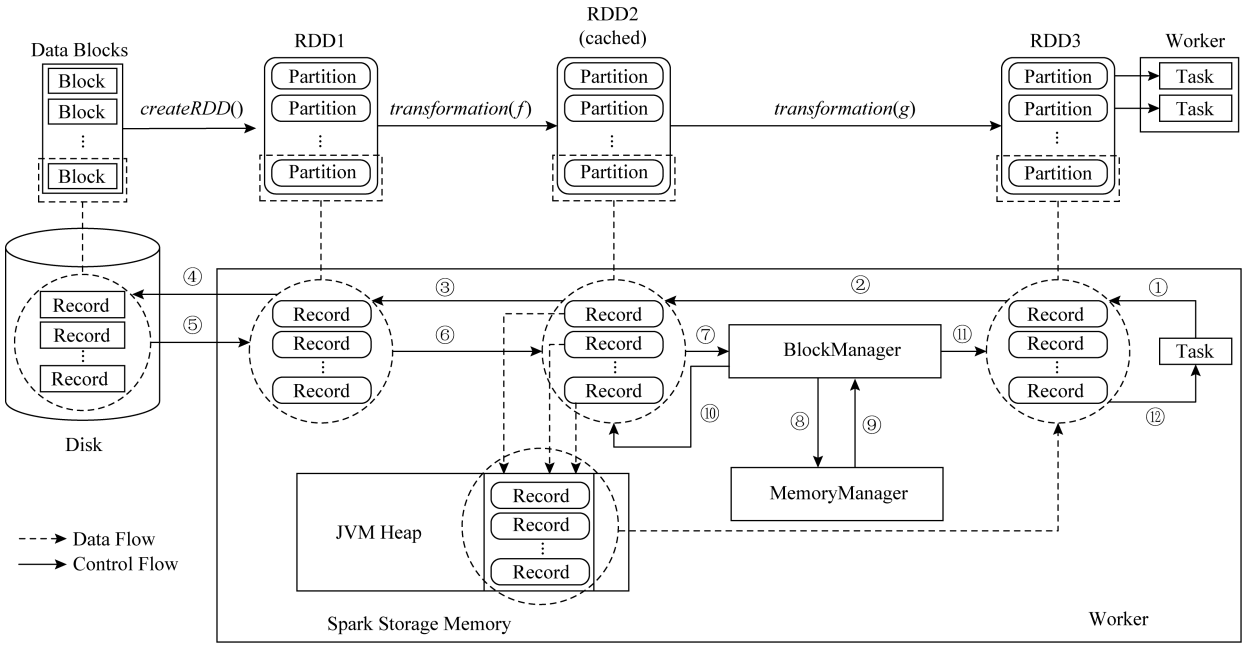

Fig. 8 Memory access process of storage memory

图 8 存储内存使用过程

执行内存一样,既可以从堆上分配,也可以从堆外分配,返回的都是连续的地址空间,从而保证了当前线程访存的局部性.

4 实验评测

4.1 实验环境

本文采用如图 9 所示的 Spark 运行模式,主要组件包括:1 个驱动器节点、1 个主节点、多个从节点.其中,1 个主节点和多个从节点组成集群,主节点用于管理计算资源,而从节点用于执行具体的计算;驱动器节点位于应用程序端,负责向集群提交任务.

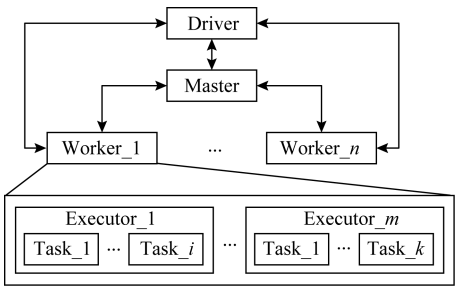

Fig. 9 The development model of Spark

图 9 Spark 部署模型

在实际的生产环境中,集群的情况往往比较复杂.首先,同一个物理集群上运行的除了 Spark,可能还有 HDFS,Yarn,Presto 等其他分布式程序;其次,针对不同的需求,集群的规模可能从几台到几千台不等;此外,对于不同的业务,运行在集群上的应用程序也多种多样.然而,优化 Spark 的前提是对 Spark 自身原理和特性的分析和理解,为了避免复杂环境的影响,本文的实验将焦点聚集在 Spark 本身.考虑到计算节点的访存行为是由 Spark 本身决定的,不受集群规模影响,因此本文选择了代表 Spark 的 Standalone 部署模式的实验环境,采用 3 台物理机组成 HDFS 集群和 Spark 集群,为避免相互干扰进而使观测和分析变得复杂,本文将 HDFS 集群和 Spark 集群的工作节点尽量分开,物理机的角色如表 1 所示:

Table 1 The role of Physical Machines

表 1 实验所用物理机角色

Machine	HDFS Role	Spark Role
1	NameNode SecondaryNameNode DataNode	Master
2		Worker
3		Worker

每台物理机有 2 个 Intel® Xeon® E5620CPU 和 32 GB 物理内存;每个 CPU 主频为 2.4 GHz,包含 4 个 Core,每个 Core 支持双线程.CPU 采用 IA32e 模式,支持 4 KB,2 MB 和 1 GB 大小的页.

为了更好地评测大数据处理程序的性能,很多

基准测试程序应运而生,常用的有 BigDataBench[25], CloudSuite[26], SparkBench[27]等.Spark 拥有活跃的社区,贡献着各种类型的计算模型,目前提供 4 种类型的应用:机器学习、SQL 查询、图计算、流计算[28-29].表 2 列出了本文中所用的 SparkBench 的负载程序及输入大小.本文基于这 4 种类型的应用,以读写 RDD 的评测结果为例,评测借助 SMTT 追踪分析不同程序的不同特征的效果.

Table 2 Description and Input of Workloads

表 2 实验负载及输入

Category	Workload	Input
Machine Learning	LogisticRegression (LR)	73 GB
	SVM	60 GB
	MatrixFactorization (MF)	53 MB
Graph Computation	PageRank (PR)	5 GB
	SVDPlusPlus (SVDPP)	547 MB
	TriangleCount (TC)	2 GB
SQL Query	RDDRelation (RR)	4 GB
Streaming Application	PageViewStream (PVS)	Random

4.2 实验结果

我们追踪了每一个 Spark 负载中 RDD 的读写开销情况,由于不同负载业务类型不同,因此在 RDD 访问方面也呈现出不同的特性,这个特性与负载的具体程序实现方式紧密相关.

图 10 显示在只有一个主节点和一个从节点的单节点集群上 RDD 引发的页遍历读写操作占比.RDD 的读操作比例平均为 2.20%,写操作比例平均为 2.55%,平均有 4.75%的周期用于 RDD 访问.其中,应用程序 TC 的比例最大:读操作比例为 4.87%、写操作为 5.21%,累计占到总执行周期的 10.08%.PR 和 TC 相似,读操作占 4.55%、写操作占 4.91%,总共有 9.46%的周期在进行 RDD 读写.此外,PVS 读写占比 7.22%、RR 读写占比 6.59%、RR 读写占比 5.38%,也都高于平均开销.比例最低的负载为 MF,读操作比例为 0.34%、写操作比例为 0.46%,总共花费的执行周期为 0.80%.而 LR 读操作为 1.63%、写操作为 1.91%,SVM 读操作为 1.72%、写操作为 2.03%,这 2 个负载虽然读写比例也低于平均值,但其所占总执行周期的比例分别为 3.54%和 3.75%,相对居中.

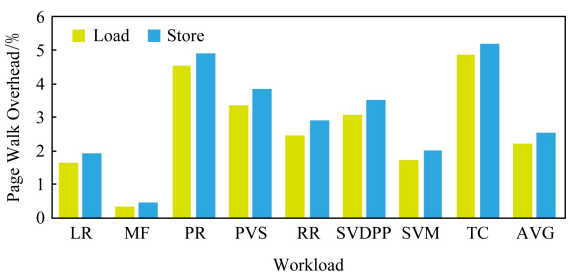

Fig. 10 Page walk overhead of RDD of single node mode

图 10 单节点 RDD 访问引发的页遍历开销

由图 10 可得结论:写操作开销略高于读操作开销,这是因为 RDD 被设计为只读的,常用的 RDD 会被持久化到内存中,此后可以多次重复读取,但是"写"只有一次,即它被创建的时候.

图 11 给出了基于 SMTT 获得的 1 个主节点、2 个从节点的多节点集群上各 Spark 负载由 RDD 访问所花费的周期占总执行周期的比例.平均情况,读操作所占周期比例为 2.15%、写操作所占周期比例为 2.52%,所以总的 RDD 读写周期比例为 4.67%.首先,开销较大的负载为:TC 操作为 5.29%、写操作为 5.61%,PR 读操作为 4.18%、写操作为 4.55%,SVDPP 读操作为 3.49%、写操作为 3.96%,总的周期比例分别为 10.90%,8.73%,7.45%.其次,开销居中的是 PVS 读操作为 2.49%、写操作为 2.93%,RR 读操作为 2.27%、写操作为 2.67%,SVM 读操作为 1.74%、写操作为 2.08%,LR 读操作为 1.57%、读操作为 1.90%,所耗周期比例分别为 5.42%,4.94%,3.82%,3.47%.最后,MF 的开销最小,读操作为 0.38%、写操作为 0.53%,总比例仅为 0.91%.

Fig. 11 Page walk overhead of RDD of multiple node mode

图 11 多节点 RDD 访问引发的页遍历开销

由图 11 可得结论:无论是单节点还是多节点,Spark 访存引发的页遍历开销普遍较低,这主要是因为 Spark 迭代计算、连续分配执行内存和存储内存的设计.迭代计算使程序读取初始数据时尽可能复用内存,从而减少内存占用.并且 Spark 每次从执行内存和存储内存中都分配连续的空间,保证了程序访存的局部性,保持了较高的 TLB 命中率和较低的页表遍历开销.

单节点和多节点情况下不同负载 RDD 所占内存比例如图 12 所示,不同负载的内存占用率差别较大.单节点时,内存使用率的平均值为 40.74%.其中,LR 使用率为 80.10%,SVM 使用率为 76.55%,SVDPP 使用率为 71.32%,TC 使用率为 71.17%,相对较高.PR 使用率为 56.52%,RR 使用率为 35.52%,接近平均值.而 PVS 使用率为 13.91%,MF 使用率为 8.73%,远远低于平均值.

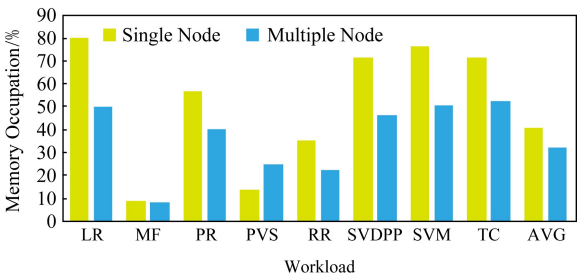

Fig. 12 Memory occupation of RDD

图 12 单节点和多节点下 RDD 内存占用率

多节点时,内存占用率普遍不高,平均值为 32.01%.其中,TC 使用率为 52.51%,SVM 使用率为 50.80%,LR 使用率为 49.73%,SVDPP 使用率为 46.17%,PR 使用率为 40.47%,相对较高.PVS 使用率为 24.58%,RR 使用率为 22.12%,MF 使用率为 8.17%,内存利用率最低.

结论 1. 各负载的 RDD 内存占用率差别较大,其中,LR,SVM,SVDPP,TC 内存占用率较高,PR,RR,PVS 内存占用率较低,MF 内存占用率最低.

结论 2. 相比单节点集群,多节点内存利用率较低.

结论 3. PVS 负载在单节点和多节点情况下的行为差异最大.

5 结论与展望

本文针对大数据计算环境下 Spark 底层行为与上层应用程序特征之间的关联缺失问题,设计了 SMTT 工具,打通了 Spark 层、JVM 层和 OS 层,实现了上层应用程序的语义与底层物理内存信息的对应.针对 Spark 内存计算的特点,分别针对执行内存和存储内存设计了不同的追踪方案.本文使用 SMTT 分析了内存使用方式,并分析不同负载用于 RDD 读/写的开销和内存占用情况,结果显示本文所设计的工具能够有效支持 Spark 的存储系统分析,为 Spark 的性能优化奠定了基础.

参 考 文 献

[1] Lin Ziyu. Principles and Applications of Big Data Technology-Big Data Conception, Storage, Processing, Analysis, and Application [M]. 2nd ed. Beijing: Posts & Telecom Press, 2017 (in Chinese)
(林子雨. 大数据技术原理与应用:概念、存储、处理、分析与应用[M]. 第 2 版. 北京:人民邮电出版社, 2017)

[2] White T. Hadoop: The Definitive Guide [M]. Sebastopol, CA: O'Reilly Media, 2009

[3] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters [C] //Proc of the 6th Symp on Operating System Design & Implementation (OSDI). Berkeley, CA: USENIX Association, 2004: 107-113

[4] Gao Yanjie. Data Processing with Spark: Technology, Application, and Performance Optimization [M]. Beijing: China Machine Press, 2014 (in Chinese)
(高彦杰. Spark 大数据处理:技术、应用与性能优化[M]. 北京:机械工业出版社, 2014)

[5] Dhiman G, Ayoub R, Rosing T. PDRAM: A hybrid PRAM and DRAM main memory system [C] //Proc of the 46th ACM/IEEE Design Automation Conf (DAC). Piscataway, NJ: IEEE, 2009: 664-669

[6] Jiang Tao, Zhang Qianlong, Hou Rui, et al. Understanding the behavior of in-memory computing workloads [C] //Proc of the 14th IEEE Int Symp on Workload Characterization (IISWC). Piscataway, NJ: IEEE, 2014: 22-30

[7] Mavridis I, Karatza H. Performance evaluation of cloud-based log file analysis with Apache Hadoop and Apache Spark [J]. Journal of Systems and Software, 2017, 125: 133-151

[8] Singh J, Panda S N, Kaushal R. Performanceevaluation of big data frameworks: MapReduce and Spark [J]. Intelligent Communication, Control and Devices, 2018, 624: 1611-1619

[9] Pan S. The performance comparison of Hadoop and Spark [D]. Minnesota: St. Cloud State University, 2016: 19-31

[10] Lu Kezhong, Zhu Jinbin, Li Zhengmin, et al. Design of RDD persistence method in Spark for SSDs [J]. Journal of Computer Research and Development, 2017, 54(6): 1381-1390 (in Chinese)
(陆克中, 朱金彬, 李正民, 等. 面向固态硬盘的 Spark 数据持久化方法设计[J]. 计算机研究与发展, 2017, 54(6): 1381-1390)

[11] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing [C] //Proc of the 9th USENIX Symp on Networked Systems Design and Implementation (NSDI). Berkeley, CA: USENIX Association, 2012: 15-28

[12] Xu Jungang, Wang Guolu, Liu Shengyuan, et al. A novel performance evaluation and optimization model for big data System [C] //Proc of the 15th Int Symp on Parallel and Distributed Computing (ISPDC). Piscataway, NJ: IEEE, 2016: 121-130

[13] Armbrust M, Xin R S, Lian Cheng, et al. Spark SQL: Relational data processing in Spark [C] //Proc of the 2015 ACM SIGMOD Int Conf on Management of Data (SIGMOD). New York: ACM, 2015: 1383-1394

[14] Aleksandar M. Perf tool: Performance analysis tool for Linux [EB/OL]. 2012 [2020-02-01]. http://lacasa.uah.edu/portal/Upload/tutorials/perf.tool/PerfTool.pdf

[15] Magnusson P S, Christensson M, Eskilson J, et al. Simics: A full system simulation platform [J]. Computer, 2002, 35 (2): 50-58

[16] Bellard F. QEMU, a fast and portable dynamic translator [C] //Proc of the Annual Conf on USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2005: 41-46

[17] Huang Yongbing, Chen Licheng, Cui Zehan, et al. HMTT: A hybrid hardware/software tracing system for bridging the DRAM access trace's semantic gap [J]. ACM Transactions on Architecture and Code Optimization, 2014, 11 (1): Article No.7

[18] Bao Yungang, Chen Mingyu, Ruan Yuan, et al. HMTT: A platform independent full-system memory trace monitoring system [C] //Proc of the 2008 ACM SIGMETRICS Int Conf on Measurement and Modeling of Computer Systems (SIGMETRICS). New York: ACM, 2008: 229-240

[19] Levi O. Pin—A dynamic binary instrumentation tool [EB/OL]. 2012 [2020-04-01]. http://pintool.org/

[20] Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation [C] //Proc of the ACM SIGPLAN 2007 Conf on Programming Language Design and Implementation (PLDI). New York: ACM, 2007: 89-100

[21] Zaharia M. Spark monitoring and instrumentation [EB/OL]. 2019 [2020-04-01]. http://spark.apache.org/docs/latest/monitoring.html

[22] Awan A J, Brorsson M, Vlassov V, et al. Architectural impact on performance of in-memory data analytics: Apache spark case study [J]. arXiv preprint arXiv:1604.08484, 2016

[23] Chintapalli S, Dagit D, Evans B, et al. Benchmarking streaming computation engines: Storm, Flink and Spark streaming [C] //Proc of the Int Parallel and Distributed Processing Symp. Chicago: IEEE, 2016: 1789-1798

[24] Grishchenko A. Sparkmemory management [EB/OL]. 2016 [2020-02-01]. https://0x0fff.com/spark-memory-management/

[25] Wang Lei, Zhan Jianfeng, Luo Chunjie, et al. BigDataBench: A big data benchmark suite from Internet services [C] //Proc of the 20th Int Symp on High Performance Computer Architecture (HPCA). Piscataway, NJ: IEEE, 2014: 488-499

[26] Ferdman M, Adileh A, Kocberber O, et al. Clearing the clouds: A study of emerging scale-out workloads on modern hardware [J]. ACM SIGPLAN Notices, 2012, 47(4): 37-48

[27] Li Min, Tan Jian, Wang Yangdong, et al. SparkBench: A comprehensive benchmarking suite for in memory data analytic platform Spark [C] //Proc of the 12th ACM Int Conf on Computing Frontiers (CF). New York: ACM, 2015: 2575-2589

[28] Zhang Feng, Liu Min, Gui Feng, et al. A distributed frequent itemset mining algorithm using Spark for big data analytics [J]. Cluster Computing, 2015, 18(4): 1493-1501

[29] Zhu Jia, Xu Chuanhua, Li Zhixiu, et al. An examination of on-line machine learning approaches for pseudo-random generated data [J]. Cluster Computing, 2016, 19(3): 1309-1321

Xu Danya, born in 1996. Master candidate. Her main research interests include computer architecture, high performance computing.

Wang Jing, born in 1982. PhD, associate professor. Her main research interests include computer architecture, energy efficient computing, high-performance computing, hardware reliability and variability.

Wang Li, born in 1987. Received his ME degree in computer science and technology from the Capital Normal University in 2019. His main research interests include computer architecture and fault tolerant design.

Zhang Weigong, born in 1967. PhD, professor. Member of CCF. His main research interests include computer architecture, high reliable embedded system design.