Apache Spark 统一内存管理模型详解

本文将对 Spark 的内存管理模型进行分析，下面的分析全部是基于 Apache Spark 2.2.1 进行的。为了让下面的文章看起来不枯燥，我不打算贴出代码层面的东西。文章仅对统一内存管理模块(UnifiedMemoryManager)进行分析，如对之前的静态内存管理感兴趣，请参阅网上其他文章。

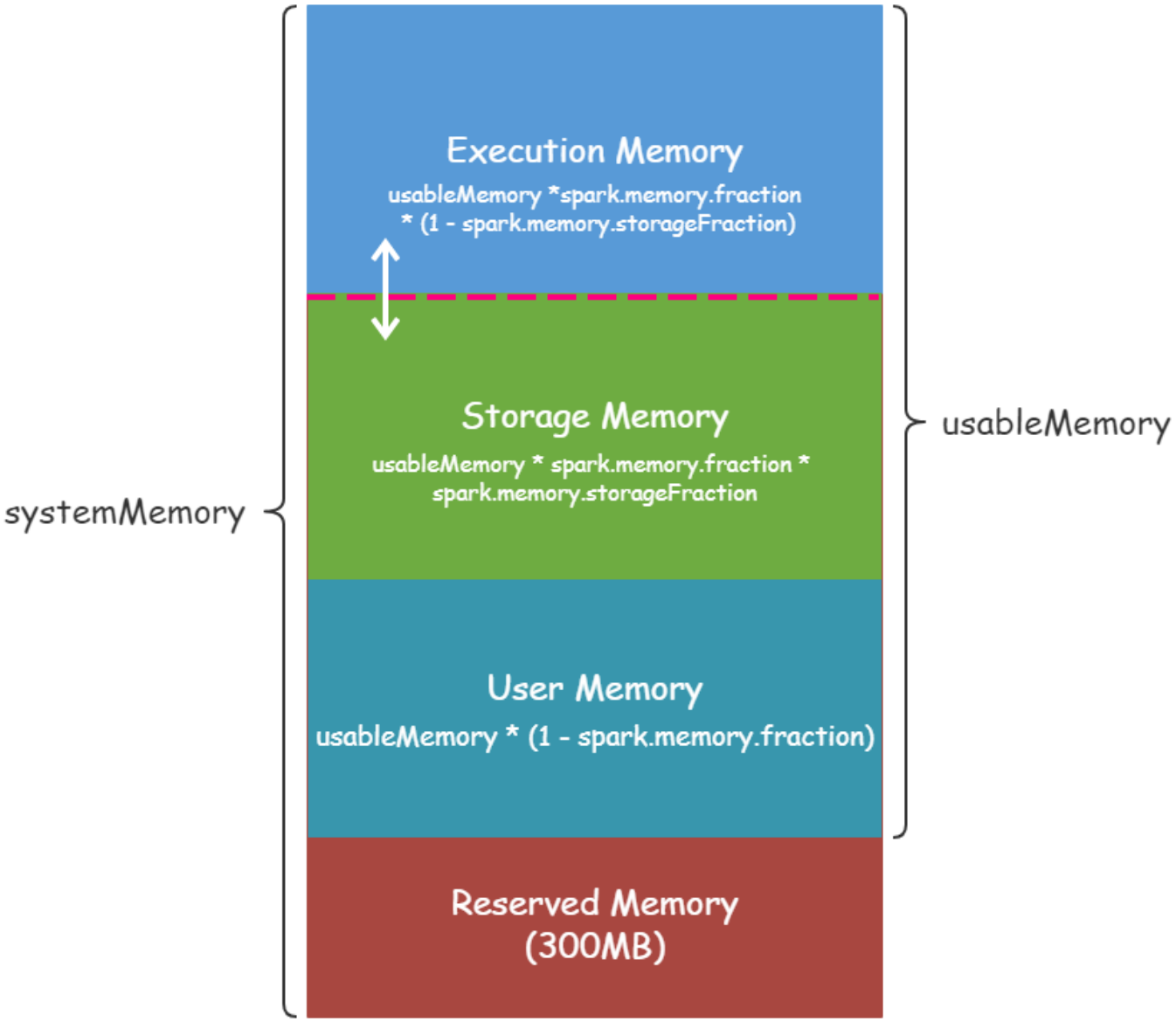
我们都知道 Spark 能够有效的利用内存并进行分布式计算，其内存管理模块在整个系统中扮演着非常重要的角色。为了更好地利用
Spark，深入地理解其内存管理模型具有非常重要的意义，这有助于我们对 Spark 进行更好的调优；在出现各种内存问题时，能够摸清头脑，找到哪块内存区域出现问题。下文介绍的内存模型全部指 Executor 端的内存模型， Driver
端的内存模型本文不做介绍。统一内存管理模块包括了堆内内存(On-heap
Memory)和堆外内存(Off-heap Memory)两大区域，下面对这两块区域进行详细的说明

堆内内存(On-heap Memory)

默认情况下，Spark 仅仅使用了堆内内存。Executor
端的堆内内存区域大致可以分为以下四大块：

- Execution 内存：主要用于存放 Shuffle、Join、Sort、Aggregation
  等计算过程中的临时数据
- Storage 内存：主要用于存储 spark 的 cache 数据，例如RDD的缓存、unroll数据；
- 用户内存（User Memory）：主要用于存储 RDD 转换操作所需要的数据，例如 RDD
  依赖等信息。
- 预留内存（Reserved Memory）：系统预留内存，会用来存储Spark内部对象。

整个 Executor 端堆内内存如果用图来表示的话，可以概括如下：



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

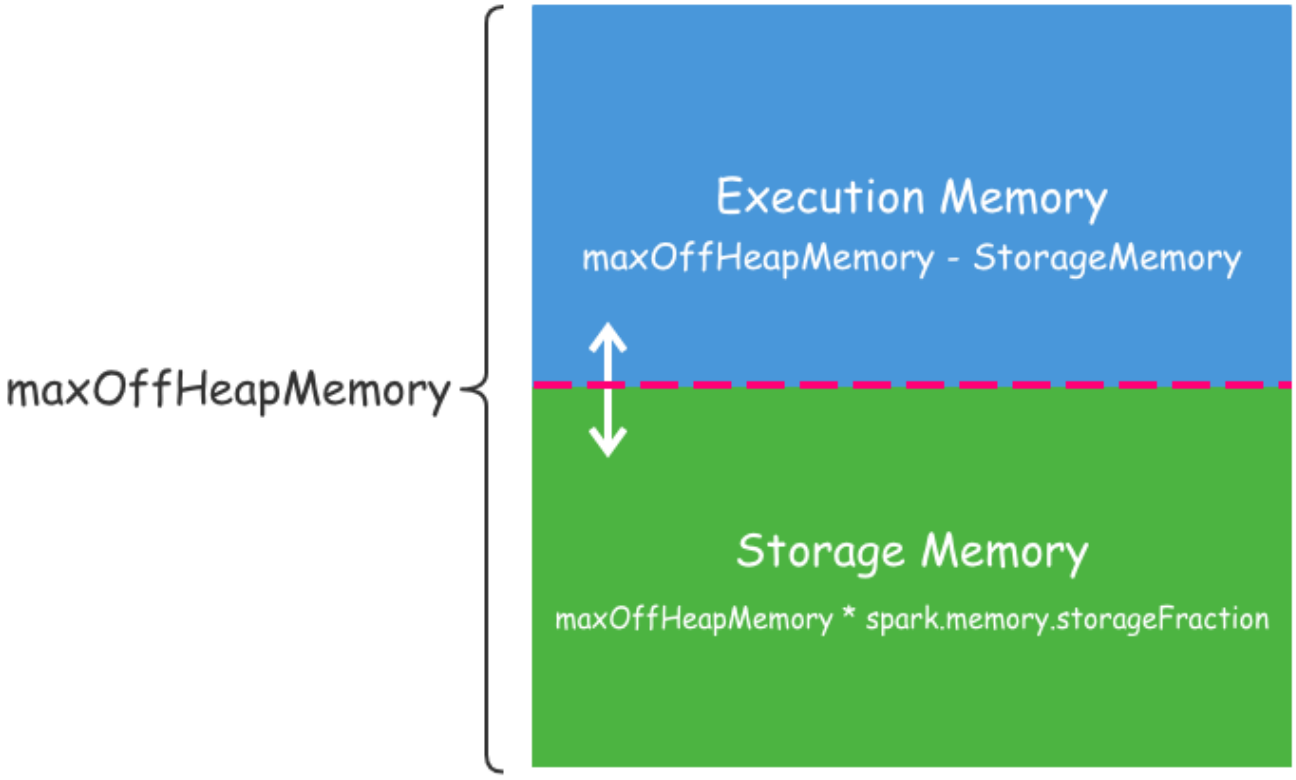
我们对上图进行以下说明：

- systemMemory = Runtime.getRuntime.maxMemory，其实就是通过参数 spark.executor.memory 或 --executor-memory 配置的。
- reservedMemory 在 Spark 2.2.1 中是写死的，其值等于 300MB，这个值是不能修改的（如果在测试环境下，我们可以通过 spark.testing.reservedMemory 参数进行修改）；
- usableMemory = systemMemory - reservedMemory，这个就是 Spark 可用内存；

堆外内存(Off-heap Memory)

Spark 1.6 开始引入了Off-heap memory(详见SPARK-11389)。这种模式不在 JVM 内申请内存，而是调用 Java 的 unsafe 相关 API 进行诸如 C 语言里面的 malloc() 直接向操作系统申请内存，由于这种方式不进过 JVM 内存管理，所以可以避免频繁的 GC，这种内存申请的缺点是必须自己编写内存申请和释放的逻辑。

默认情况下，堆外内存是关闭的，我们可以通过 spark.memory.offHeap.enabled 参数启用，并且通过 spark.memory.offHeap.size 设置堆外内存大小，单位为字节。如果堆外内存被启用，那么 Executor 内将同时存在堆内和堆外内存，两者的使用互补影响，这个时候 Executor 中的 Execution 内存是堆内的 Execution 内存和堆外的 Execution 内存之和，同理，Storage 内存也一样。相比堆内内存，堆外内存只区分 Execution 内存和 Storage 内存，其内存分布如下图所示：



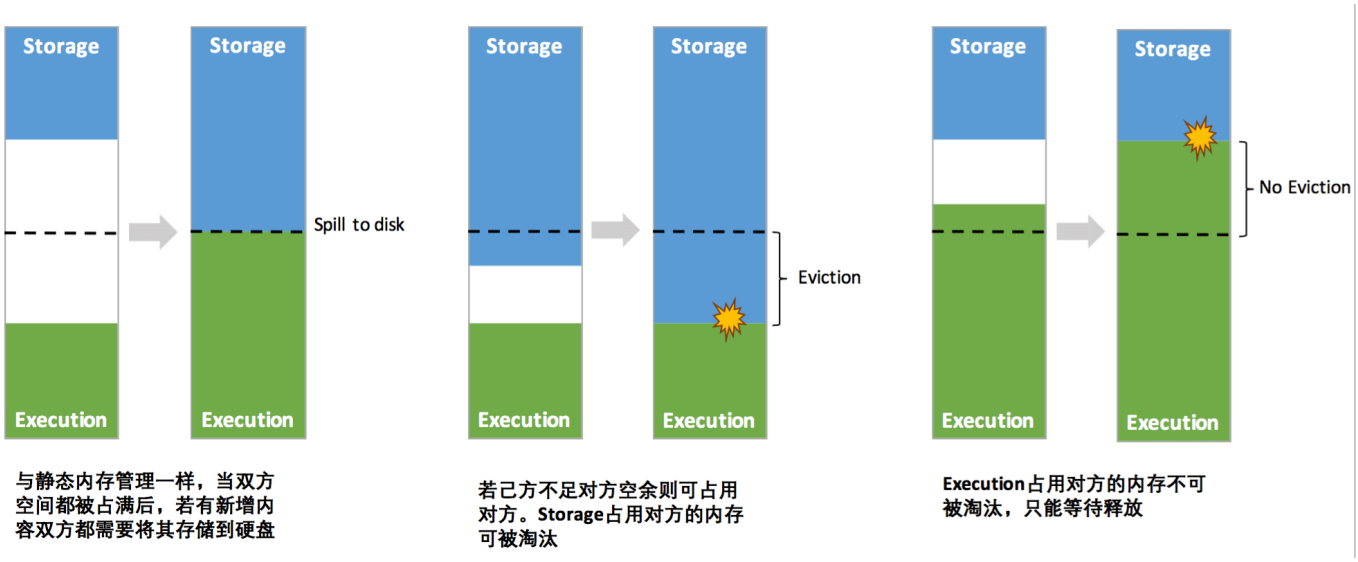
如果想及时了解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

上图中的 maxOffHeapMemory 等于 spark.memory.offHeap.size 参数配置的。

Execution 内存和 Storage 内存动态调整

细心的同学肯定看到上面两张图中的 Execution 内存和 Storage 内存之间存在一条虚线，这是为什么呢？

用过 Spark 的同学应该知道，在 Spark 1.5 之前，Execution 内存和 Storage 内存分配是静态的，换句话说就是如果 Execution 内存不足，即使 Storage 内存有很大空闲程序也是无法利用到的；反之亦然。这就导致我们很难进行内存的调优工作，我们必须非常清楚地了解 Execution 和 Storage 两块区域的内存分布。而目前 Execution 内存和 Storage 内存可以互相共享的。也就是说，如果 Execution 内存不足，而 Storage 内存有空闲，那么 Execution 可以从 Storage 中申请空间；反之亦然。所以上图中的虚线代表 Execution 内存和 Storage 内存是可以随着运作动态调整的，这样可以有效地利用内存资源。Execution 内存和 Storage 内存之间的动态调整可以概括如下：



如果想及时了解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

具体的实现逻辑如下：

- 程序提交的时候我们都会设定基本的 Execution 内存和 Storage 内存区域（通过 spark.memory.storageFraction 参数设置）；
- 在程序运行时，如果双方的空间都不足时，则存储到硬盘；将内存中的块存储到磁盘的策略是按照 LRU 规则进行的。若己方空间不足而对方空余时，可借用对方的空间;（存储空间不足是指不足以放下一个完整的 Block）
- Execution 内存的空间被对方占用后，可让对方将占用的部分转存到硬盘，然后"归还"借用的空间
- Storage 内存的空间被对方占用后，目前的实现是无法让对方"归还"，因为需要考虑 Shuffle 过程中的很多因素，实现起来较为复杂；而且 Shuffle 过程产生的文件在后面一定会被使用到，而 Cache 在内存的数据不一定在后面使用。

注意，上面说的借用对方的内存需要借用方和被借用方的内存类型都一样，都是堆内内存或者都是堆外内存，不存在堆内内存不够去借用堆外内存的空间。

Task 之间内存分布

为了更好地使用使用内存，Executor 内运行的 Task 之间共享着 Execution 内存。具体的，Spark

内部维护了一个 HashMap 用于记录每个 Task 占用的内存。当 Task 需要在 Execution 内存区域申请 numBytes 内存，其先判断 HashMap 里面是否维护着这个 Task 的内存使用情况，如果没有，则将这个 Task 内存使用置为0，并且以 TaskId 为 key，内存使用为 value 加入到 HashMap 里面。之后为这个 Task 申请 numBytes 内存，如果 Execution 内存区域正好有大于 numBytes 的空闲内存，则在 HashMap 里面将当前 Task 使用的内存加上 numBytes，然后返回；如果当前 Execution 内存区域无法申请到每个 Task 最小可申请的内存，则当前 Task 被阻塞，直到有其他任务释放了足够的执行内存，该任务才可以被唤醒。每个 Task 可以使用 Execution 内存大小范围为 1/2N ~ 1/N，其中 N 为当前 Executor 内正在运行的 Task 个数。一个 Task 能够运行必须申请到最小内存为 (1/2N * Execution 内存)；当 N = 1 的时候，Task 可以使用全部的 Execution 内存。

比如如果 Execution 内存大小为 10GB，当前 Executor 内正在运行的 Task 个数为5，则该 Task 可以申请的内存范围为 10 / (2 * 5) ~ 10 / 5，也就是 1GB ~ 2GB的范围。

一个示例

为了更好的理解上面堆内内存和堆外内存的使用情况，这里给出一个简单的例子。

只用了堆内内存

现在我们提交的 Spark 作业关于内存的配置如下：

--executor-memory 18g

由于没有设置 spark.memory.fraction 和 spark.memory.storageFraction 参数，我们可以看到 Spark UI 关于 Storage Memory 的显示如下：

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
driver	123.206.77.132:33618	Active	0	0.0 B / 429.1 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump
1	www.iteblog.com:50956	Active	0	0.0 B / 10.1 GB	0.0 B	1	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump

Showing 1 to 2 of 2 entries Previous 1 Next

上图很清楚地看到 Storage Memory 的可用内存是 10.1GB，这个数是咋来的呢？根据前面的规则，我们可以得出以下的计算：

systemMemory = spark.executor.memory
reservedMemory = 300MB
usableMemory = systemMemory - reservedMemory
StorageMemory= usableMemory * spark.memory.fraction * spark.memory.storageFraction

如果我们把数据代进去，得出以下的结果：

systemMemory = 18Gb = 19327352832 字节
reservedMemory = 300MB = 300 * 1024 * 1024 = 314572800
usableMemory = systemMemory - reservedMemory = 19327352832 - 314572800 = 19012780032
StorageMemory= usableMemory * spark.memory.fraction * spark.memory.storageFraction
= 19012780032 * 0.6 * 0.5 = 5703834009.6 = 5.312109375GB

不对啊，和上面的 10.1GB 对不上啊。为什么呢？这是因为 Spark UI 上面显示的 Storage Memory 可用内存其实等于 Execution 内存和 Storage 内存之和，也就是 usableMemory * spark.memory.fraction：

StorageMemory= usableMemory * spark.memory.fraction
= 19012780032 * 0.6 = 11407668019.2 = 10.62421GB

还是不对，这是因为我们虽然设置了 --executor-memory 18g，但是 Spark 的 Executor 端通过 Runtime.getRuntime.maxMemory 拿到的内存其实没这么大，只有 17179869184 字节，所以 systemMemory = 17179869184，然后计算的数据如下：

systemMemory = 17179869184 字节
reservedMemory = 300MB = 300 * 1024 * 1024 = 314572800
usableMemory = systemMemory - reservedMemory = 17179869184 - 314572800 = 16865296384
StorageMemory= usableMemory * spark.memory.fraction
= 16865296384 * 0.6 = 9.42421875 GB

我们通过将上面的 16865296384 * 0.6 字节除于 1024 * 1024 * 1024 转换成 9.42421875 GB，和 UI 上显示的还是对不上，这是因为 Spark UI 是通过除于 1000 * 1000 * 1000 将字节转换成

GB，如下：

systemMemory = 17179869184 字节
reservedMemory = 300MB = 300 * 1024 * 1024 = 314572800
usableMemory = systemMemory - reservedMemory = 17179869184 - 314572800 = 16865296384
StorageMemory= usableMemory * spark.memory.fraction
= 16865296384 * 0.6 字节 = 16865296384 * 0.6 / (1000 * 1000 * 1000) = 10.1GB

现在终于对上了。

具体将字节转换成 GB 的计算逻辑如下(core 模块下面的 /core/src/main/resources/org/apache/spark/ui/static/utils.js)：

```
function formatBytes(bytes, type) {
    if (type !== 'display') return bytes;
    if (bytes == 0) return '0.0 B';
    var k = 1000;
    var dm = 1;
    var sizes = ['B', 'KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'];
    var i = Math.floor(Math.log(bytes) / Math.log(k));
    return parseFloat((bytes / Math.pow(k, i)).toFixed(dm)) + ' ' + sizes[i];
}
```

我们设置了 --executor-memory 18g，但是 Spark 的 Executor 端通过 Runtime.getRuntime.maxMemory 拿到的内存其实没这么大，只有 17179869184 字节，这个数据是怎么计算的？
Runtime.getRuntime.maxMemory 是程序能够使用的最大内存，其值会比实际配置的执行器内存的值小。这是因为内存分配池的堆部分分为 Eden，Survivor 和 Tenured 三部分空间，而这里面一共包含了两个 Survivor 区域，而这两个 Survivor 区域在任何时候我们只能用到其中一个，所以我们可以使用下面的公式进行描述：

ExecutorMemory = Eden + 2 * Survivor + Tenured
Runtime.getRuntime.maxMemory = Eden + Survivor + Tenured

上面的 17179869184 字节可能因为你的 GC 配置不一样得到的数据不一样，但是上面的计算公式是一样的。

用了堆内和堆外内存

现在如果我们启用了堆外内存，情况咋样呢？我们的内存相关配置如下：

spark.executor.memory 18g
spark.memory.offHeap.enabled true
spark.memory.offHeap.size 10737418240

从上面可以看出，堆外内存为 10GB，现在 Spark UI 上面显示的 Storage Memory 可用内存为 20.9GB，如下：

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	(GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
driver	123.206.77.132:48167	Active	0	0.0 B / 11.2 GB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump
1	www.iteblog.com:51908	Active	0	0.0 B / 20.9 GB	0.0 B	1	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
2	www.iteblog.com:56932	Active	0	0.0 B / 20.9 GB	0.0 B	1	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
3	www.iteblog.com:43380	Active	0	0.0 B / 20.9 GB	0.0 B	1	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump

其实 Spark UI 上面显示的 Storage Memory
可用内存等于堆内内存和堆外内存之和，计算公式如下：

堆内
systemMemory = 17179869184 字节
reservedMemory = 300MB = 300 * 1024 * 1024 = 314572800
usableMemory = systemMemory - reservedMemory = 17179869184 - 314572800 = 16865296384
totalOnHeapStorageMemory = usableMemory * spark.memory.fraction
= 16865296384 * 0.6 = 10119177830

堆外
totalOffHeapStorageMemory = spark.memory.offHeap.size = 10737418240

StorageMemory = totalOnHeapStorageMemory + totalOffHeapStorageMemory
= (10119177830 + 10737418240) 字节
= (20856596070 / (1000 * 1000 * 1000)) GB
= 20.9 GB