PackedInts（一）

为了能节省空间，Lucene使用PackedInts类对long类型的数据进行压缩存储，基于内存使用率（memory-efficient）跟解压速度（读取速度），提供了多种压缩方法，我们先通过类图预览下这些压缩方法。

图1：

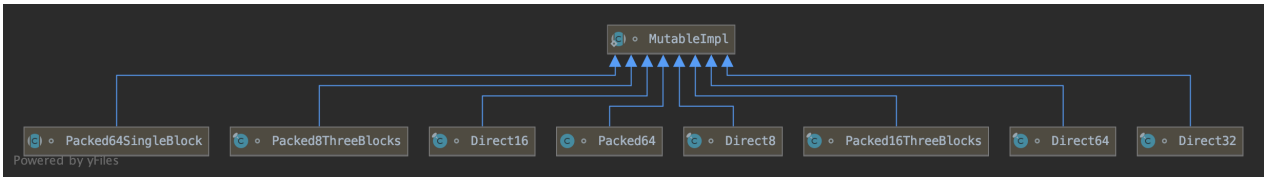
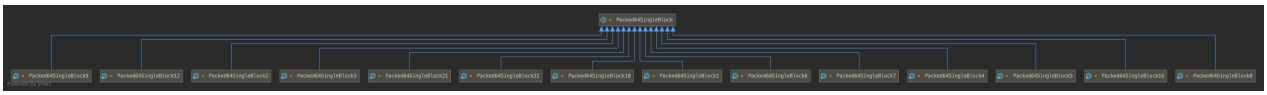


图1中MutableImpl类是PackedInts的内部类，其中Packed64SingleBlock是一个抽象类，它的实现如下所示：

图2：



点击查看大图

预备知识

在介绍PackedInts提供的压缩方法前，我们先介绍下几个预备知识。

数据类型压缩

根据待处理的数据的取值范围，选择占用字节更小的数据类型来表示原数据类型。

数组一：

```
1    long[] oldValues = {10, 130, 7};
```

long类型的数组，每个数组元素都占用8个字节，压缩前存储该数组需要24个字节（只考虑数组元素的大小），即三个数组元素的占用字节总数，它们的二进制表示如下所示：

表一：

数组元素	二进制
10	00000000_00000000_00000000_00000000_00000000_00000000_00000000_00001010
102	00000000_00000000_00000000_00000000_00000000_00000000_00000000_01100110
130	00000000_00000000_00000000_00000000_00000000_00000000_00000000_10000010

从表一可以看出，这三个值有效的数据位最多只在低8位，故可以使用字节数组来存储这三个数据：

数组二：

```
    byte[] newValues = {10, 130, 7};
```

那么压缩后的数组只需要占用3个字节。

固定位数按字节存储

在数据类型压缩的前提下，待处理的数据集使用相同的固定的bit位存储，bit位的个数由数据集中的最大值，它的数据有效bit位决定，例如有如下的数组：

数组三：

```
    long[] oldValues = {10, 290, 7};
```

三个数组元素的二进制表示如下所示：

表二：

数组元素	二进制
10	00000000_00000000_00000000_00000000_00000000_00000000_00000000_00001010
290	00000000_00000000_00000000_00000000_00000000_00000000_00000001_00100010
7	00000000_00000000_00000000_00000000_00000000_00000000_00000000_00000111

上表中，最大的数组元素是290，它的有效数据位为低16位，它是所有数组元素中有效数据位占用bit位最多的，那么在进行数据类型压缩后，新的数据类型既要保证数值290的精度，同时还要使得占用字节最小，故只能选择short类型的数组，并且数组元素10、7都需要跟290一样，需要存储2个字节的大小，尽管它们两个只需要1个字节就可以表示：

数组四：

```
    short[] newValues = {10, 290, 7};
```

压缩后的数组只需要占用6个字节。

为什么要选用固定字节：

能让所有数据用同一种数据类型表示，并且任何数据不会出现精度缺失问题，尽管压缩率不是最高，但是读取速度是非常快的。

固定位数按位存储

上文中，数组三在使用了`固定位数按字节存储`以及`数据类型压缩`之后，生成了数组四，此时三个数组元素的二进制表示如下所示：

表三：

数组元素	二进制
10	00000000_00001010
290	00000001_00100010
7	00000000_00000111

表三中，我们可以发现，在使用`固定位数按字节存储`的前提下，仍然有高7个bit位是无效数据，那么可以通过固定位数按位存储的方式来进一步提高压缩率，该方法不会使用数据类型的压缩，只是将所有数据的有效数据位进行拼接，当拼接后的bit位达到64位时，即生成一个long值，剩余的有效数据继续拼接并生成一个新的long值，例如我们有下面的数组：

数组五：

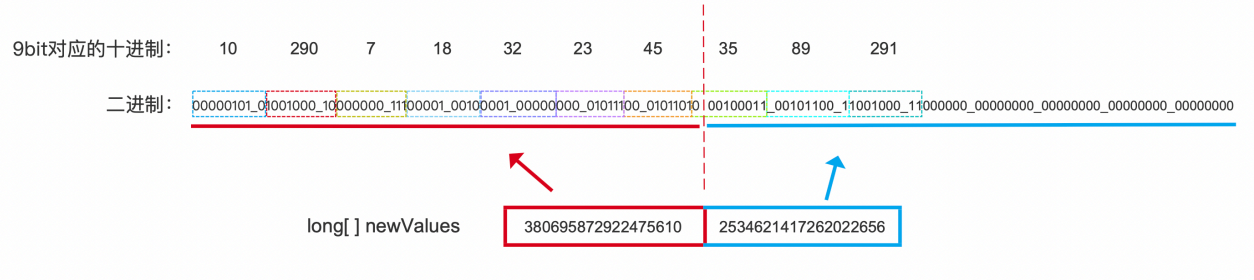
```
long[] oldValues = {10, 290, 7, 18, 32, 23, 45, 35, 89, 291};
```

在使用固定位数按位存储之后，生成的新的数组如下所示：

数组六：

```
long[] newValues = {380695872922475610, 2534621417262022656};
```

图3：



点击查看大图

图3中，由于数组五中的最大值为291，它的有效数据占用9个bit位，所以其他的数组元素也使用9个bit位表示，那么就可以使用一个long类型的来表示7个完整的数值，由于7个数值一共占用7*9 =63个bit位，剩余的1个bit位为数组元素35的最高位0（红色虚线左侧的0），35的低8位只能存放到第二个long值中，故在读取阶段，需要读取两个long才能得到35这个数值，那么原本需要10*8=80个字节存储，现在只需要2*8=16个字节。

block

block在源码中用来描述一块数据区域，在这块数据可以存放一个或多个数值，block使用数组来实现，也就是说数组的一个数组元素称为一个block，并且这个数组元素可以存放一个或多个数值。

例如上文中的数组三，它是一个short类型的数组，它有三个block，并且每一个block中存放了一个数值，而在数组六中，它有两个block，第一个block存放了7个完整的数值，以及1个不完整的数值(7*9 + 1)，第二个block最多可以存放6个完整的数值，以及2个不完整的数值(8 + 6*9 + 2 = 64)。

压缩实现

图1跟图2展示的是PackedInts提供的所有压缩实现，我们先对这些实现进行分类：

表4：

数据分布	是否有填充bit	是否单block单值	实现类
一个block	否	是	Direct8 Direct16 Direct32 Direct64
	是	否	Packed64SingleBlock1 Packed64SingleBlock2 Packed64SingleBlock3 Packed64SingleBlock4 Packed64SingleBlock5 Packed64SingleBlock6 Packed64SingleBlock7 Packed64SingleBlock8 Packed64SingleBlock9 Packed64SingleBlock10 Packed64SingleBlock12 Packed64SingleBlock16 Packed64SingleBlock21 Packed64SingleBlock32
两个block	否	否	Packed64
三个block	否	-	Packed8ThreeBlocks Packed16ThreeBlocks

我们先对表4的内容做一些介绍：

- 数据分布：该列名描述的是一个数值的信息是否可能分布在不同的block中，例如图3中的数值35，它用9个bit位来描述，其中最高位存储在第一个block，而低8位存储在第二个block中，又比如上文中的数组二跟数组四，每个数值都存储在一个block中
- 是否有填充bit：例如图3中的数值35，在第一个block中存储了7个完整的数据后，该block仅剩余1个bit位，如果该bit位不存储数值35的最高位，那么该bit位就是填充bit，也就是说，如果使用了填充bit，那么一个block中不会有不完整的数值，当然内存使用率会降低
- 是否单block单值：该列描述的是一个block中是否只存储一个数值，例如数组4，每一个block只存储一个short类型的数值，例如数组六，第一个block存储了7个完整的数值以及一个不完整的数值

接下来我们先介绍下每种实现中使用哪种数据类型来存储数据，然后再介绍下为什么Lucene提供这么多的压缩，以及如何选择，最后再介绍几个例子。

Direct*

该系列分别使用不同的数据类型来实现一个block存储一个数值，它们属于上文中提到的`固定位数按字节存储`：

- Direct8：使用byte[]数组存储
- Direct16：使用short[]数组存储
- Direct32：使用int[]数组存储
- Direct64：使用long[]数组存储

Packed8ThreeBlocks、Packed16ThreeBlocks

该系列使用三个block来存储一个数值：

- Packed8ThreeBlocks：使用byte[]数组存储
- Packed16ThreeBlocks：使用short[]数组存储

Packed64SingleBlock*

该系列使用一个block来存储一个或多个数值，并且可能存在填充bit，它们属于上文中提到的`固定位数按位存储`，所有的实现使用long[]数组存储。

Packed64

Packed64使用一个block来存储一个或多个数值，不会存在填充bit，它属于上文中提到的`固定位数按位存储`，使用long[]数组存储。

为什么Lucene提供这么多的压缩实现：

这些压缩实现分别提供不同的内存使用率以及解压速度（读取速度），下面几张图是Lucene的核心贡献者Adrien Grand提供的测试数据，它使用了三台机器来测试压缩实现的压缩性能以及解压性能：

测试一：

图4：

pc254: Intel 4 3.20 GHz, 2MB cache

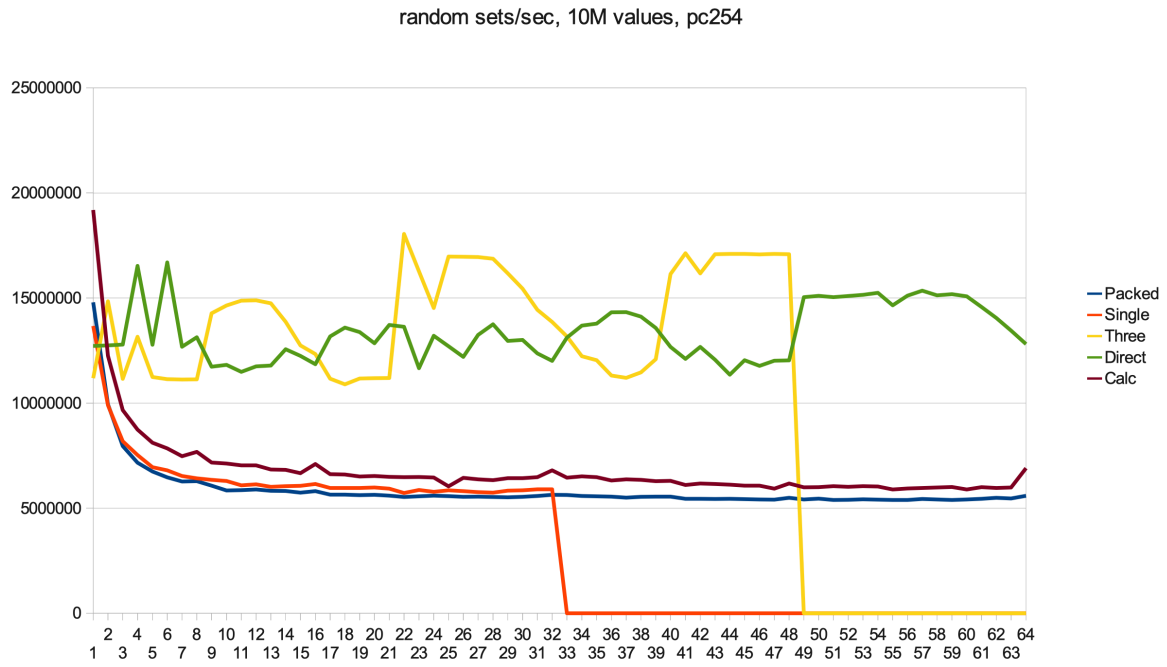
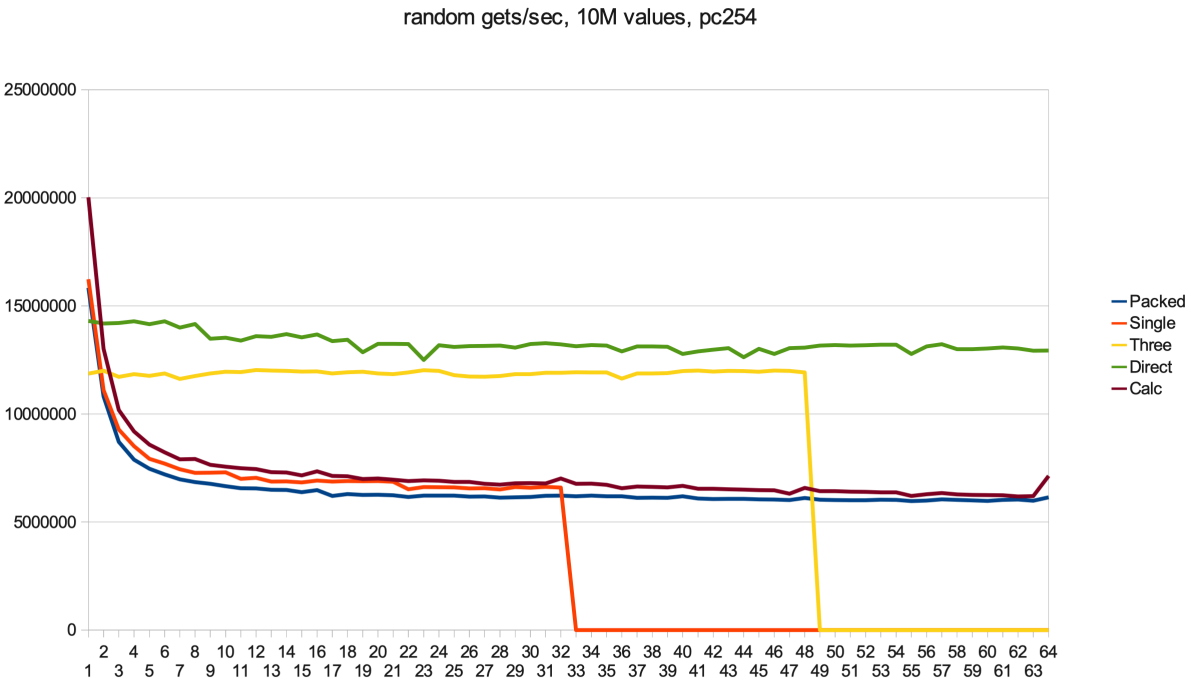


图5：



测试二：

图6：

te-prime: Intel i7 Q 820 1.73GHz, 8MB cache

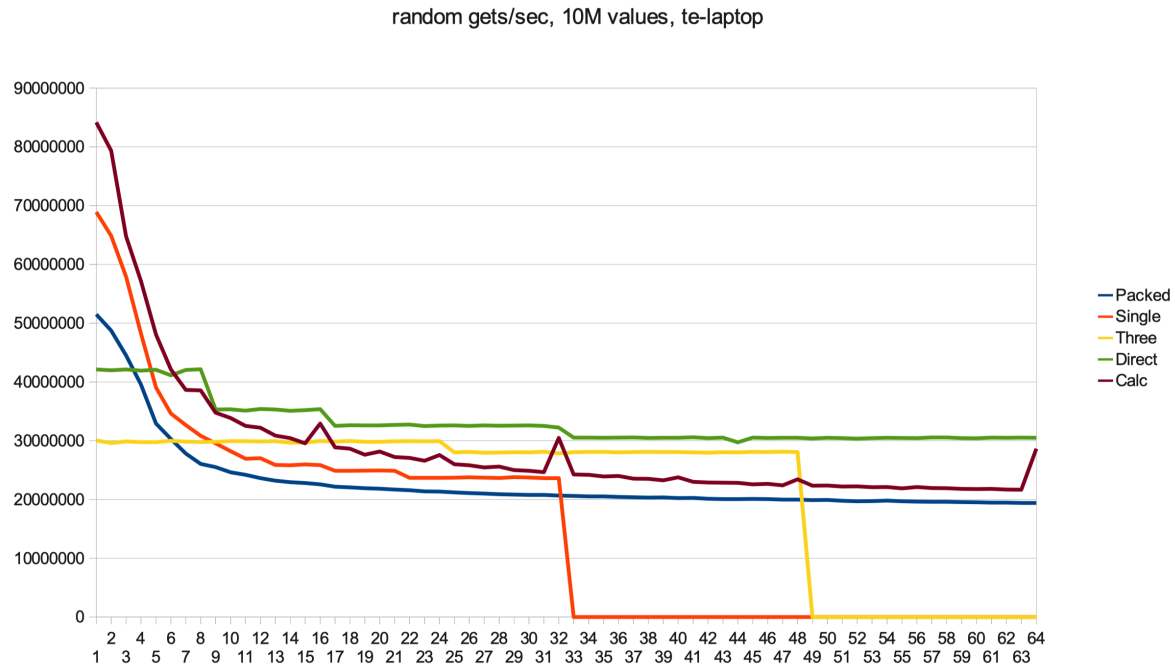
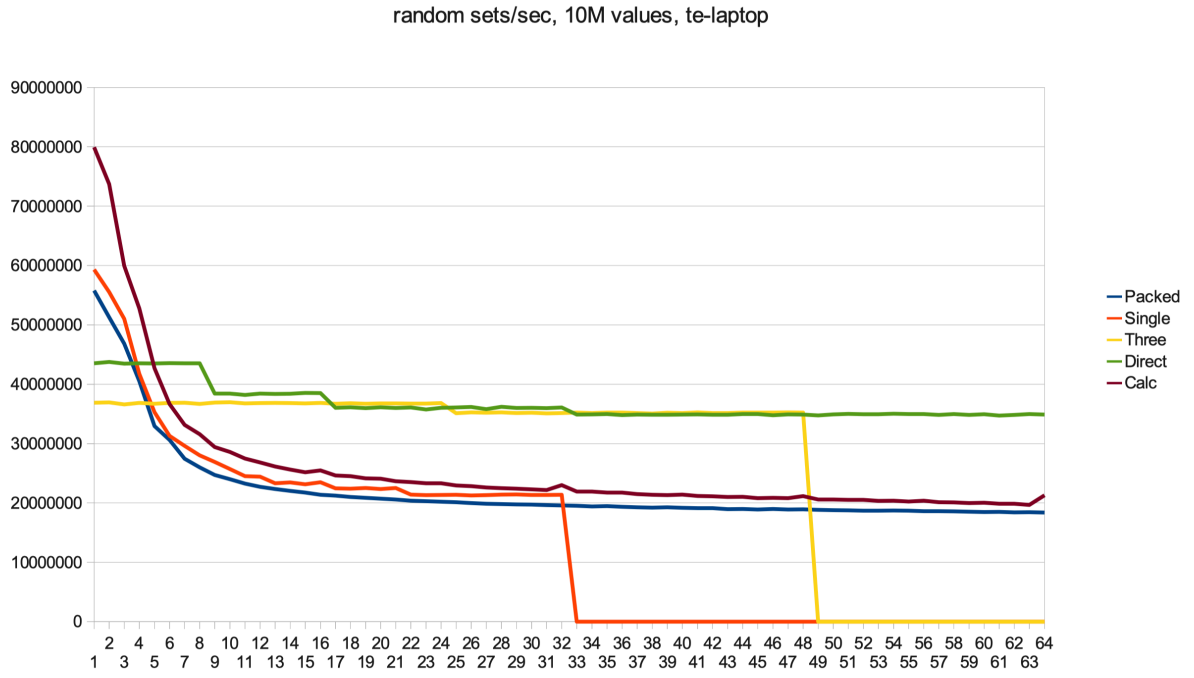


图7：



测试三：

图8：

mars: Intel Xeon E5-2670 @ 2.60 GHz, 20MB cache

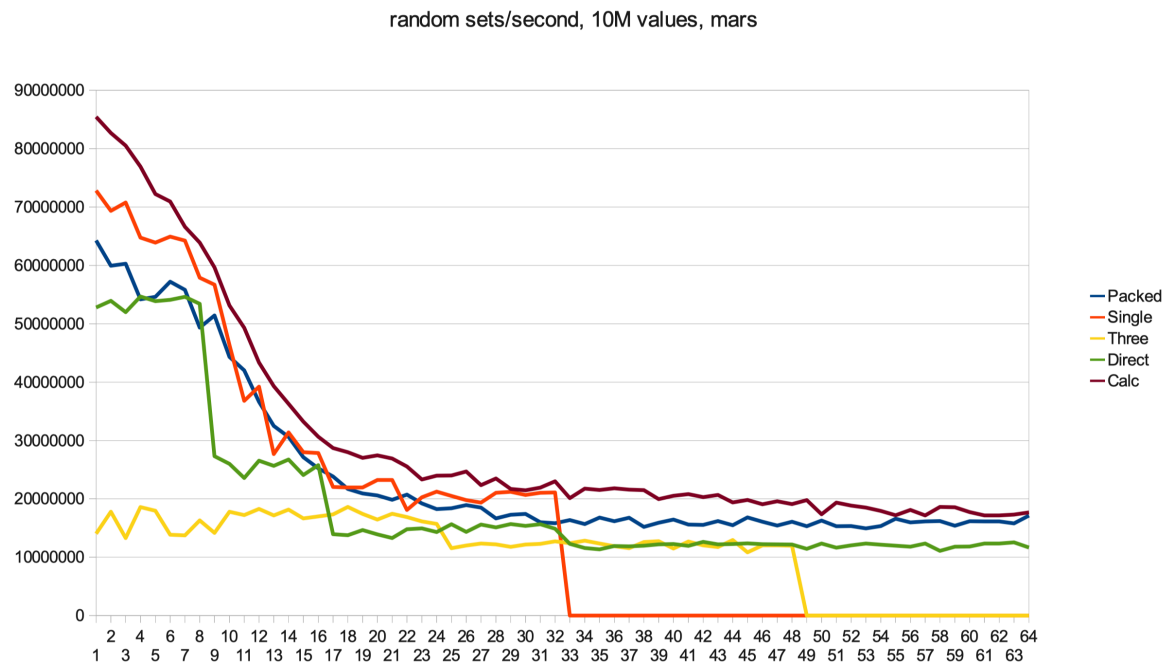
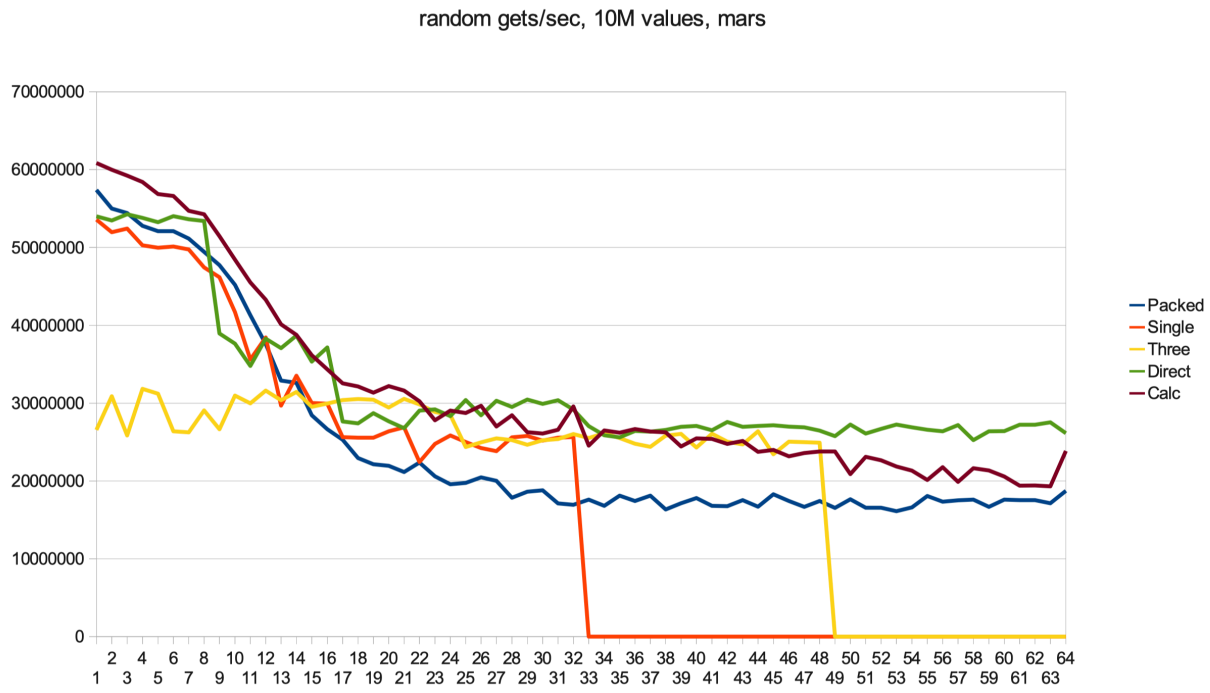


图9：



图中的曲线对应的压缩实现如下所示：

- 蓝色曲线Packed：Packed64
- 红色曲线Single：Packed64SingleBlock*
- 黄色曲线Single：Packed8ThreeBlocks、Packed16ThreeBlocks
- 黄色曲线Single：Direct*

我们先观察下Direct*与Packed64的读取速度，这两种压缩实现无论在哪一台都表现出两个极端，即Packed64读取最慢，而Direct*读取最快，而从表4我们可以看出，Packed64使用连续的bit位存储数据，待存储的数据如果没有突兀的数据，那么相对于Direct*能有很高的压缩率，例如存储连续递增的文档号，并且只存储文档差值，那么只需要1个bit位就能表示一个文档号。

在Lucene 4.0.0版本之前，只有Direct8、Direct16、Direct32、Direct64、Packed64、Packed32（该压缩算实现考虑的是对32位平台，我们这里不作介绍）可供选择，在这个版本中，当计算出待处理的数据集中最大值的数据有效位的bit个数后，我们称之为bitsPerValue，如果bitsPerValue为8，那么选择Direct8，如果bitsPerValue为32，那么选择Direct32，以此类推，即如果bitsPerValue不是8、16、32、64，那么就选择Packed64。

也就说在Lucene 4.0.0版本之前，只有bitsPerValue是8、16、32、64时，才能使用Direct*，我们考虑这么一种假设，如果bitsPerValue的值是21，并且使用Direct32存储，那么我们就能获得较好的性能，但是这么做会造成（32-21）/ 32 = 35%的空间浪费，故Adrien Grand使用long类型数组来存储，那么64个bit位的long类型可以存放3个bitsPerValue为21的数值，并且剩余的一个bit位作为填充bit，该填充bit不存储有效数据，那么我们只要读取一个block，就能读取/写入一个数值，而Packed64需要读取两个block，因为读取/写入的数值可能分布在两个block中，那么自然性能比Packed64好，而这正是空间换时间的设计，每存储3个bitsPerValue为21的数值，需要1个bit额外的空间开销，即每个数值需要1/3个bit的额外开销。

上述的原内容见：https://issues.apache.org/jira/browse/LUCENE-4062 。

上述方式即Packed64SingleBlock*压缩实现，bitsPerValue为21的数值即对应Packed64SingleBlock21，同样Adrien Grand基于这种方式做了测试，他通过10000000个bitsPerValue为21的数值使用Packed64SingleBlock21进行压缩存储，相比较使用Packed64，额外空间开销为2%，但是速度提升了44%，故使用了Packed64SingleBlock*之后，bitsPerValue不为8、16、32、64的部分bitsPerValue通过空间换时间的思想，提高了性能，在图4~图9中也能通过曲线看出来。

为什么只有部分bitsPerValue实现了Packed64SingleBlock*压缩：

从表4中可以发现，只有bitsPerValue为1，2，3，4，5，6，7，8，9，10，12，16，21，32实现了Packed64SingleBlock*压缩，我们通过一个block可存放的数值个数来介绍原因：

- 1个数值：bitsPerValue为64才能使得额外空间开销最小，每个数值的空间开销为（64-64*1）/（1）= 0，它其实就是Direct64
- 2个数值：bitsPerValue为32才能使得额外空间开销最小，每个数值的空间开销为（64-32*2）/（2）= 0
- 3个数值，bitsPerValue为21才能使得额外空间开销最小，每个数值的空间开销为（64-21*3）/（3）= 0.33
- 4个数值，bitsPerValue为16才能使得额外空间开销最小，每个数值的空间开销为（64-16*4）/（4）= 0
- 5个数值，bitsPerValue为12才能使得额外空间开销最小，每个数值的空间开销为（64-12*5）/（5）= 0.8
- 6个数值，bitsPerValue为10才能使得额外空间开销最小，每个数值的空间开销为（64-10*6）/（6）= 0.66
- 7个数值，bitsPerValue为9才能使得额外空间开销最小，每个数值的空间开销为（64-9*7）/（7）= 0.14
- 8个数值，bitsPerValue为8才能使得额外空间开销最小，每个数值的空间开销为（64-8*8）/（8）= 0
- 。。。。。。

可以看出那些实现了Packed64SingleBlock*压缩的bitsPerValue都是基于空间开销下的最优解。

如何选择压缩方式：

基于篇幅，剩余的内容将在下一篇文章中展开。

结语

无

点击下载附件