腾讯Elasticsearch压缩编码优化实践

毕杰山 （Jaison Bi）

腾讯 – 云架构平台部

目录

主流场景/矛盾

主流场景：日志类、时序类存储与分析

海量数据规模： **百PB**级别

存储周期： 7天~30天不等

存储介质： **SSD**为主，满足高性能读写需求

代表客户： CLS、云监控、B站等

主要矛盾：用户希望存储更长的周期，但存储成本太高

常规应急手段：清理旧索引；降低副本数；扩容

几乎所有的扩容性需求都源自存储空间不足

使用SATA盘或者对象存储服务来满足更长的存储周期

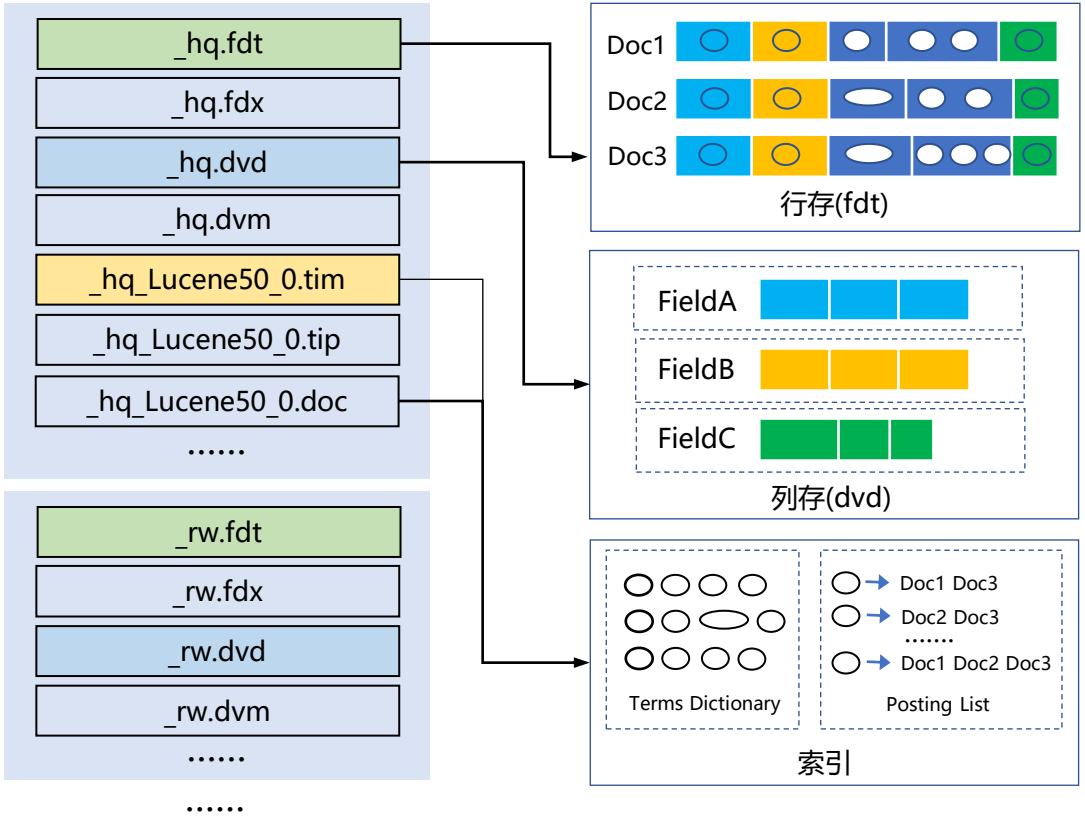
存储成本优化的可行思路

01 机型/存储介质优化

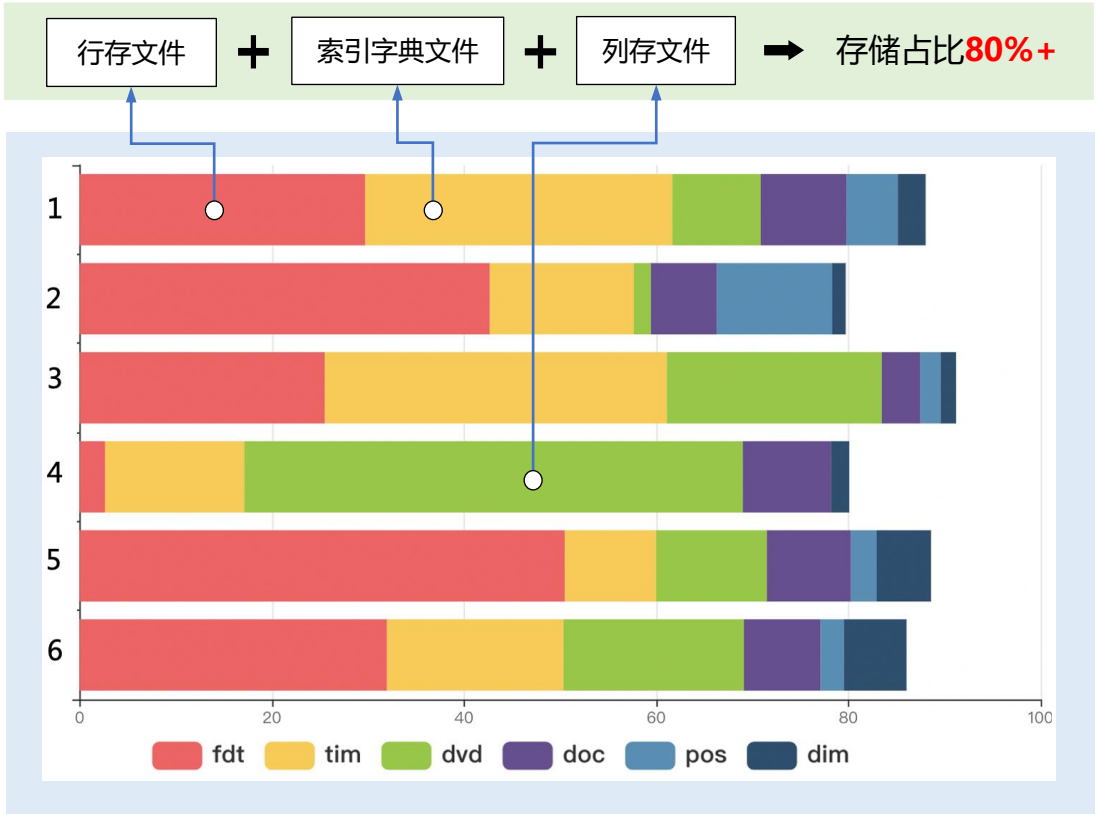
02 底层文件格式优化：**采用更高效的压缩编码**

03 架构优化： 计算与存储分离

Lucene索引文件构成概述

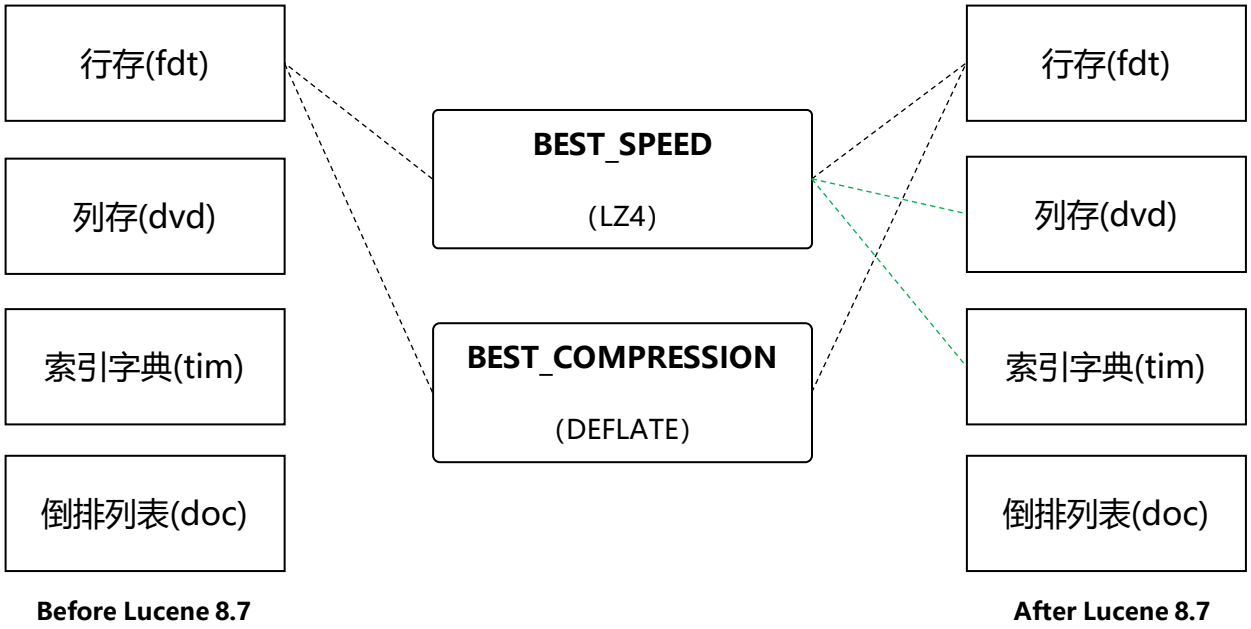


大集群索引文件构成调研



目录

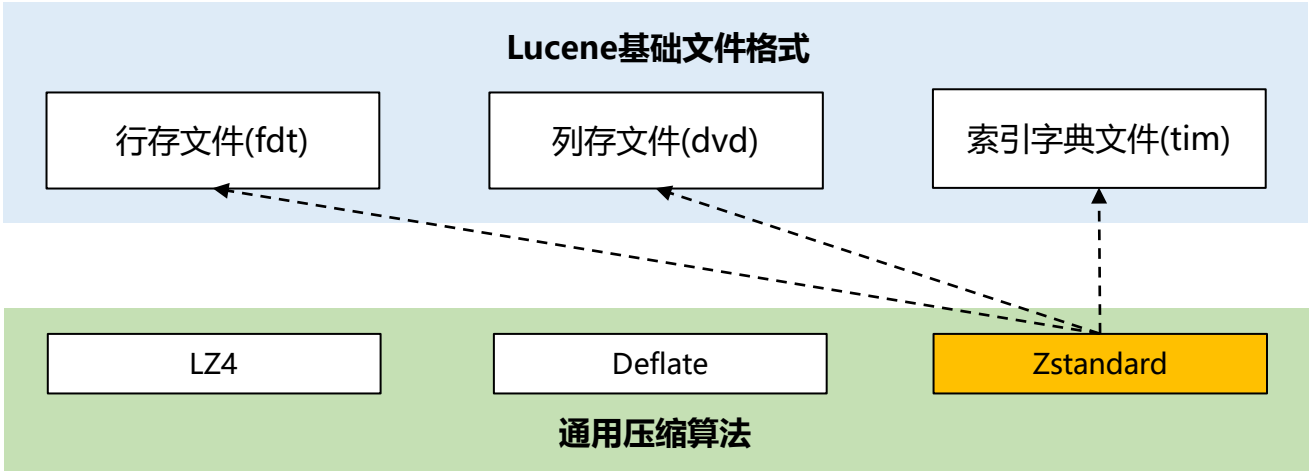
当我们谈论Lucene压缩编码时，我们在谈论什么？



压缩算法仅针对行存数据

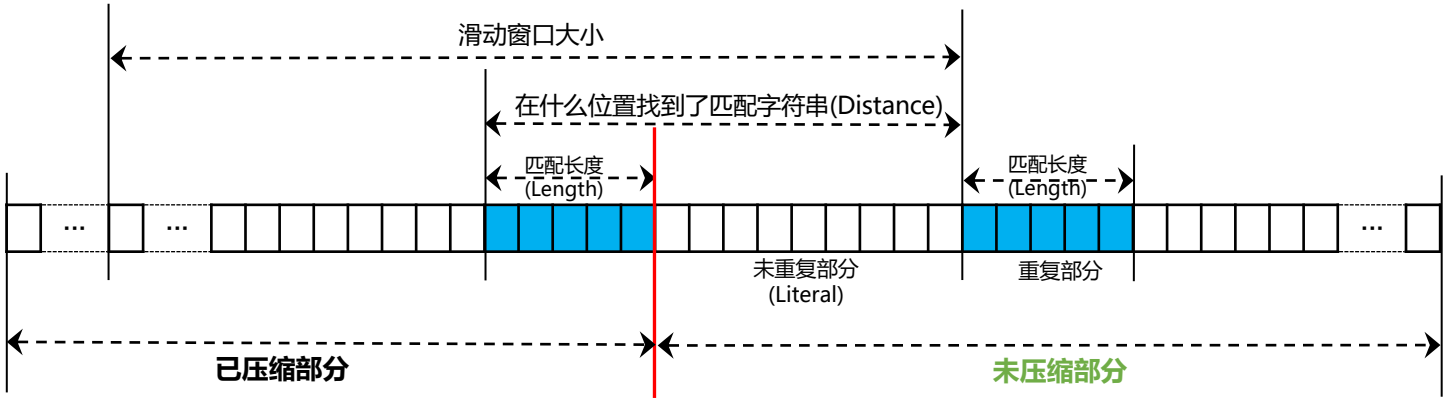
压缩算法应用于更多的文件类型

基础压缩算法的拓展



在现有压缩算法框架中新引入了Zstandard压缩算法， 作为底层通用压缩算法

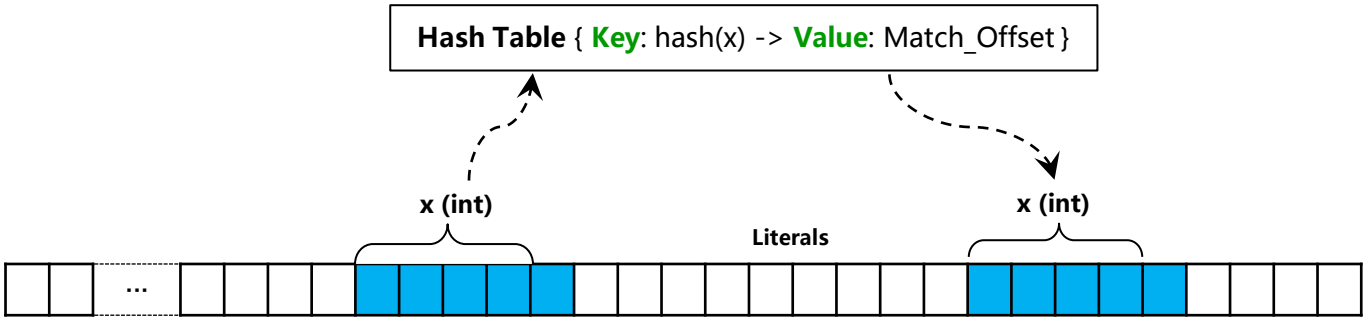
压缩算法简介：LZ77



影响压缩效果的几个关键因素：

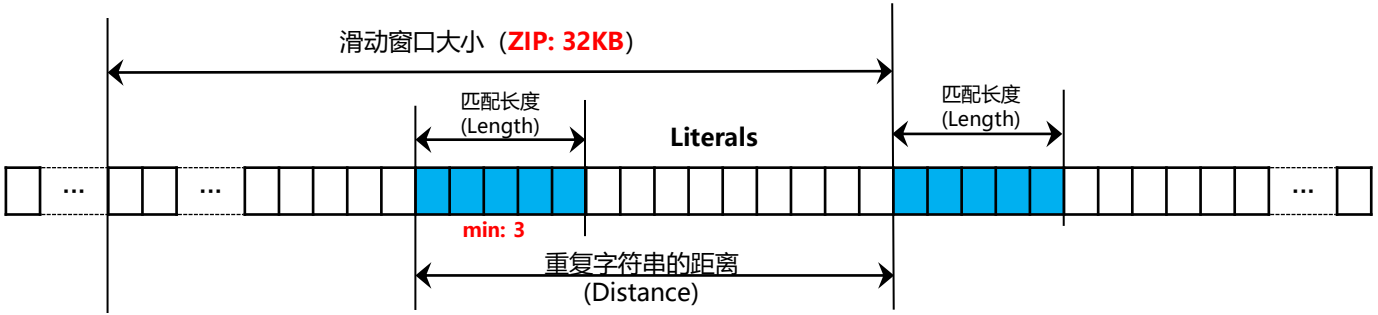
- 滑动窗口的大小（在多大的区间内找重）
- 重复字符串的最小匹配长度
- {Literal, Length, Distance}的编码方式

压缩算法简介：LZ4



- 在每一个position位置处，读取一个int值，计算Hash，然后查看该值是否在Hash Table中存在。如果存在且相等，则确保了至少4 Bytes的匹配长度。这是对LZ77算法的一点巧妙改进。
- {Literals，Length, Distance}直接存储，未做压缩编码转换。

压缩算法简介：Deflate



DEFLATE算法在如何搜索匹配字符串的思路上基本上参考了LZ77，但该算法的核心在于对{Literals, Distance, Length}数据部分的压缩。

DEFLATE压缩算法的两大步骤：

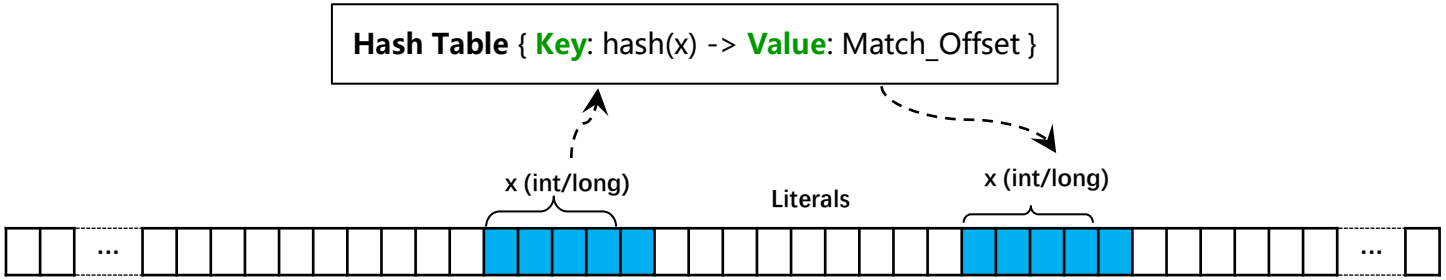
① 基于LZ77算法寻找重复字符串

② 对{literals, lengths, distances}进行Huffman+RLE组合编码

几个关键点：

- literals + lengths使用一个Huffman-Tree，distances单独使用一个Huffman-Tree。
- Huffman-Tree使用Codeword Length来表达，Codeword Length序列使用RLE编码。
- RLE编码得到的结果，进一步再使用Huffman编码。

压缩算法简介：Zstandard



- 每一个压缩块中包含Literals Section与Sequences Section。
- 一个Sequence由{LiteralLength, Offset, Match_Length} 构成。
- Literals采用Huffman编码。
- {LiteralLength, Offset, Match_Length}采用**FSE编码**。

> *FSE编码由Zstandard作者Yann Collet基于Jarek Duda的论文《Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding》中介绍的ANS算法实现*

Zstandard压缩算法核心: ANS算法

State/row	A	B	C
1	2	3	5
2	4	6	10
3	7	8	15
4	9	11	20
5	12	14	25
6	13	17	30
7	16	21	
8	18	22	
9	19	26	
10	23	28	
11	24		
12	27		
13	29		
14	31		

ANS算法的核心是Transform Table。

假设，某段数据中仅包含A, B, C三个字符，且出现的概率分别为：
P([A, B, C]) = [0.45, 0.35, 0.2]

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	A	B	A	C	B	A	B	A	C	B	A	A	B	C

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
A	B	A	A	C	B	B	A	A	C	B	A	B	A	C	A

字符A共出现了14次(14/31 ~= 0.45)，也代表着对整个区间划分成了14份。

在接收到输入字符A时，假设此时的状态为StateX：

StateX如果不在1~14的区间内，则需要右移，并且输出溢出的Bits。

StateX右移以后得到StateY(属于1~14内的一个值)。

也就是说： StateX被表达成了两部分： StateY与溢出的Bits。

StateY可以理解成**分区号**(StateY与A的交叉值得到一个新的StateZ)。

分区号本身包含在Transform Table中， 不需要在输出信息中表达。

这正是ANS数据压缩原理的奥秘所在。

Zstandard压缩算法核心: Transform Table

输入字符A -> 初始状态31

State/row	A	B	C
1	2	3	5
2	4	6	10
3	7	8	15
4	9	11	20
5	12	14	25
6	13	17	30
7	16	21	
8	18	22	
9	19	26	
10	23	28	
11	24		
12	27		
13	29		
14	31		

31右移位直到结果小于等于14，得到7

7与A的交集得到状态值16

State/row	A	B	C
1	2	3	5
2	4	6	10
3	7	8	15
4	9	11	20
5	12	14	25
6	13	17	30
7	16	21	
8	18	22	
9	19	26	
10	23	28	
11	24		
12	27		
13	29		
14	31		

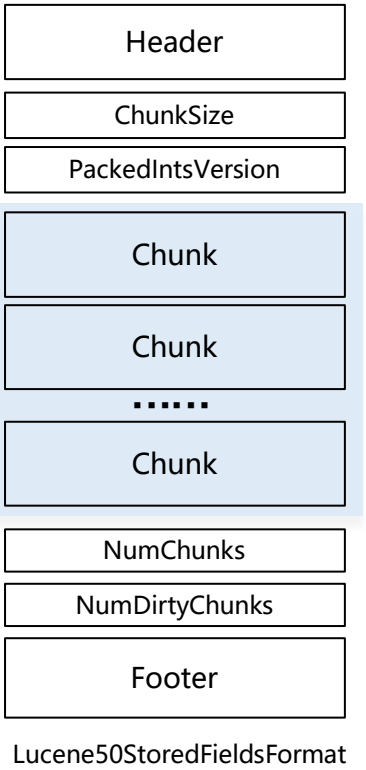
16右移位直到结果小于等于10，得到8

输入字符B -> 状态22

State/row	A	B	C
1	2	3	5
2	4	6	10
3	7	8	15
4	9	11	20
5	12	14	25
6	13	17	30
7	16	21	
8	18	22	
9	19	26	
10	23	28	
11	24		
12	27		
13	29		
14	31		

示例与图截取自：Colt McAnlis & Aleks Haecky《Understanding Compression》

基于Zstandard压缩算法的行存文件压缩



针对行存数据，引入Zstandard压缩算法。实际效果：

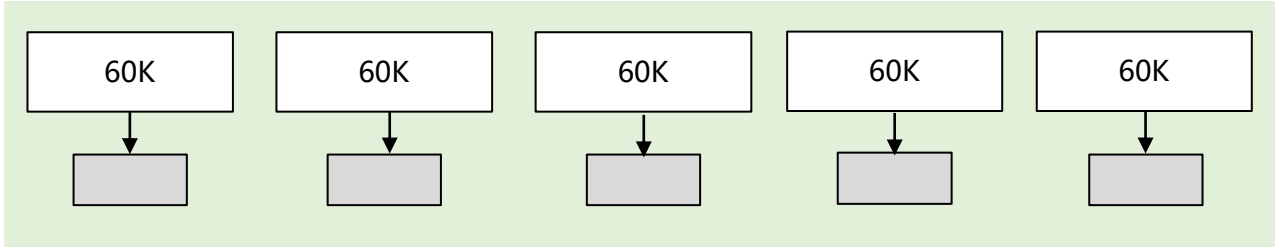
压缩算法	加载时间 (1 Shard)	加载时间 (5 Shards)	行存文件大小
LZ4	1143769ms	420447ms	4.15 GB
Deflate	1270408ms	448738ms	2.56 GB
Zstandard (16K Chunk)	1109414ms	415256ms	2.93 GB
Zstandard (32K Chunk)	1088959ms	406661ms	2.67 GB

某些场景下，Zstandard性能比LZ4更优，CPU使用率与LZ4接近。压缩效果逼近Deflate。

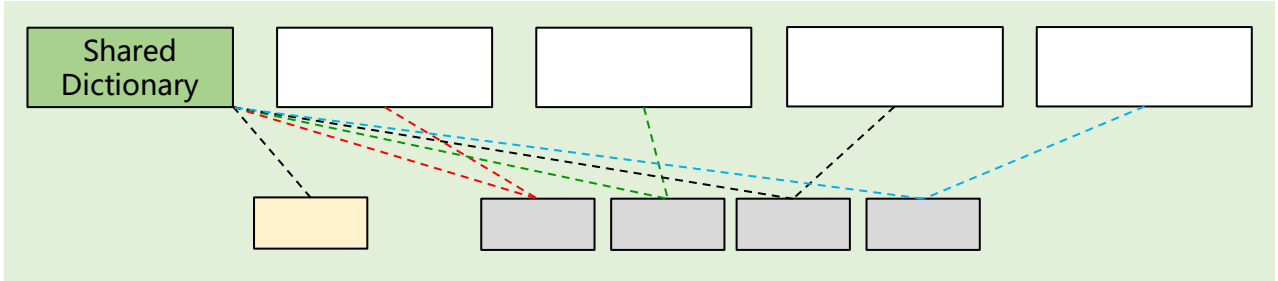
Zstandard压缩另外具备的一点优势： **可以支持预训练字典。**

可完美替代LZ4作为Elasticsearch/Lucene作为默认的压缩算法。

行存文件社区动态： Preset Dictionary for Stored Fields



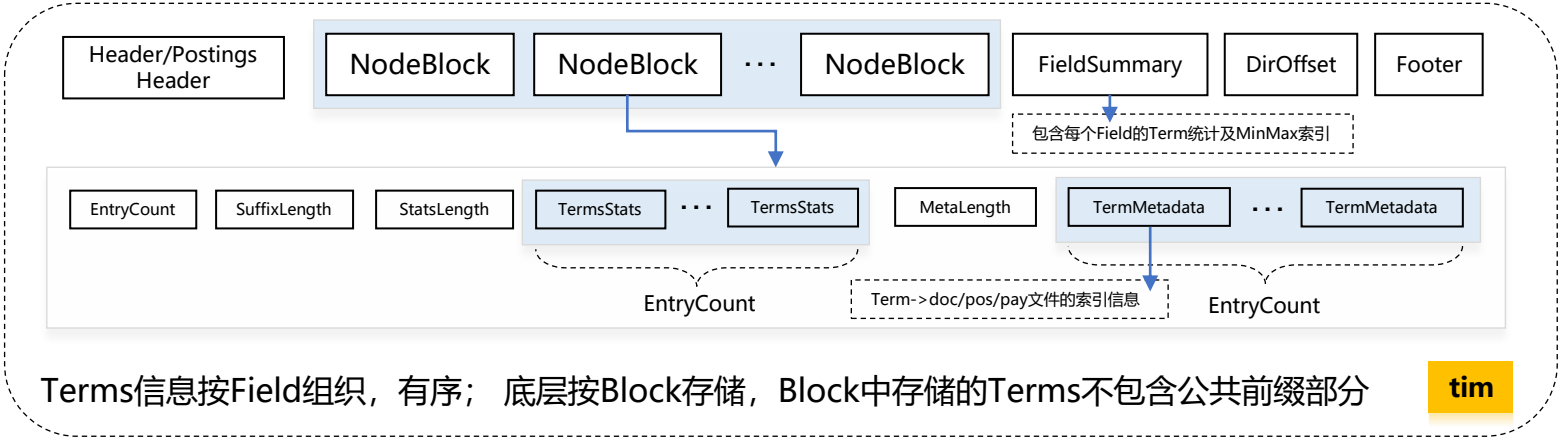
Compressing Per Chunk



Compressing per Chunk-Group

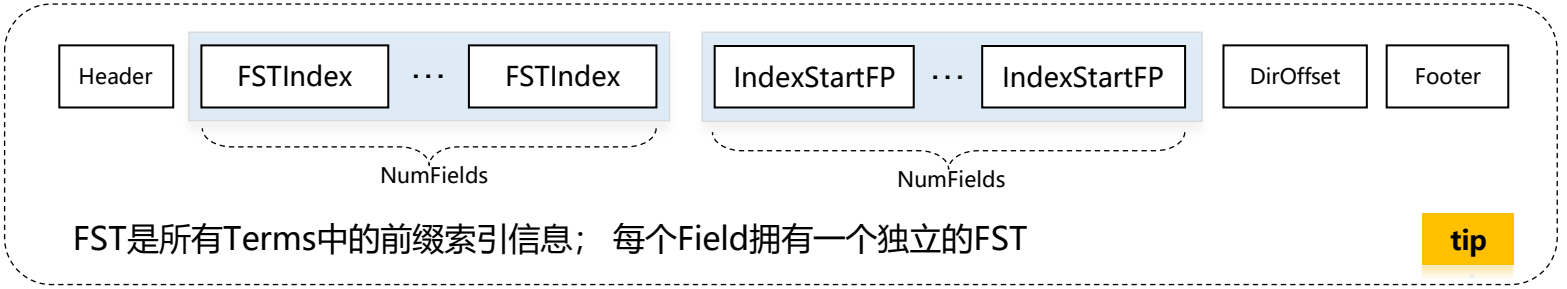
Apache Lucene 8.7版本引入Preset Dictionary特性

字典文件： Terms Dictionary的块状组织结构



Terms信息按Field组织，有序； 底层按Block存储，Block中存储的Terms不包含公共前缀部分

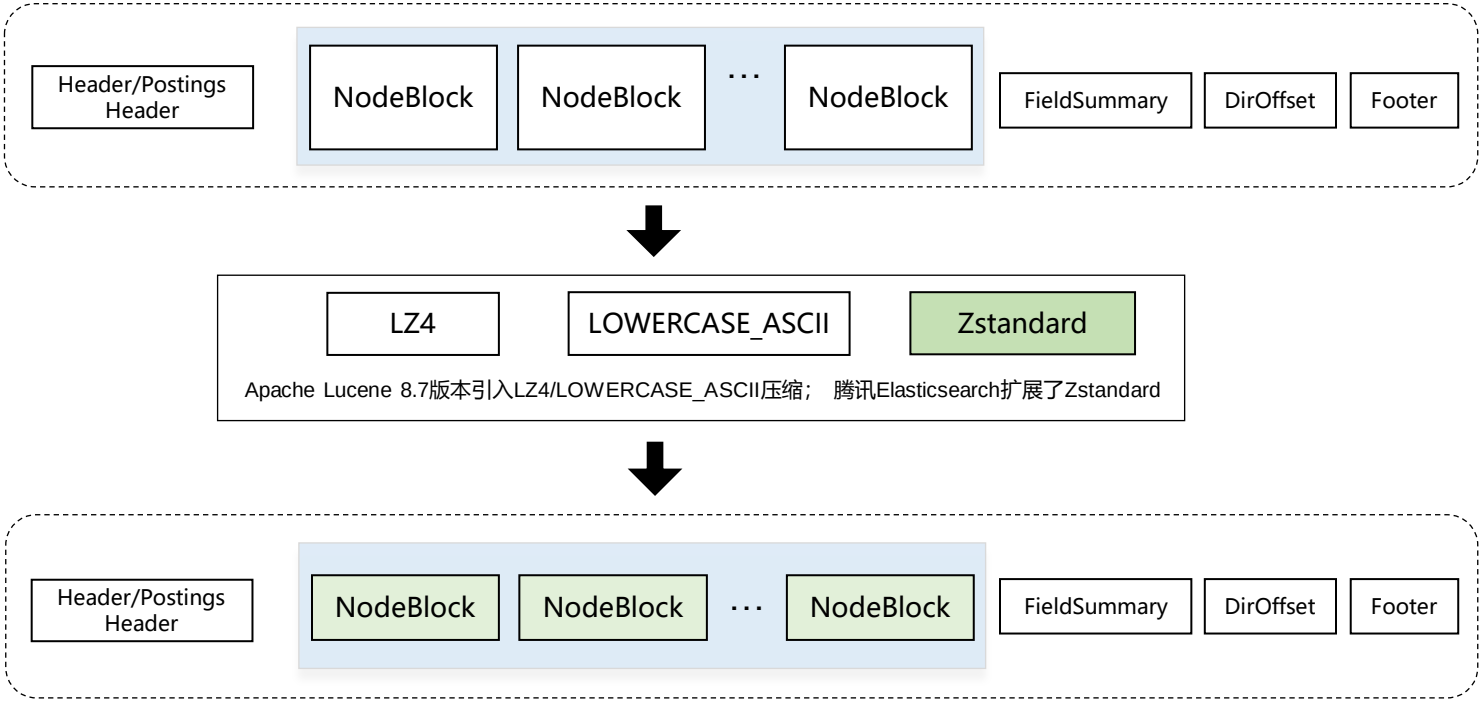
tim



FST是所有Terms中的前缀索引信息； 每个Field拥有一个独立的FST

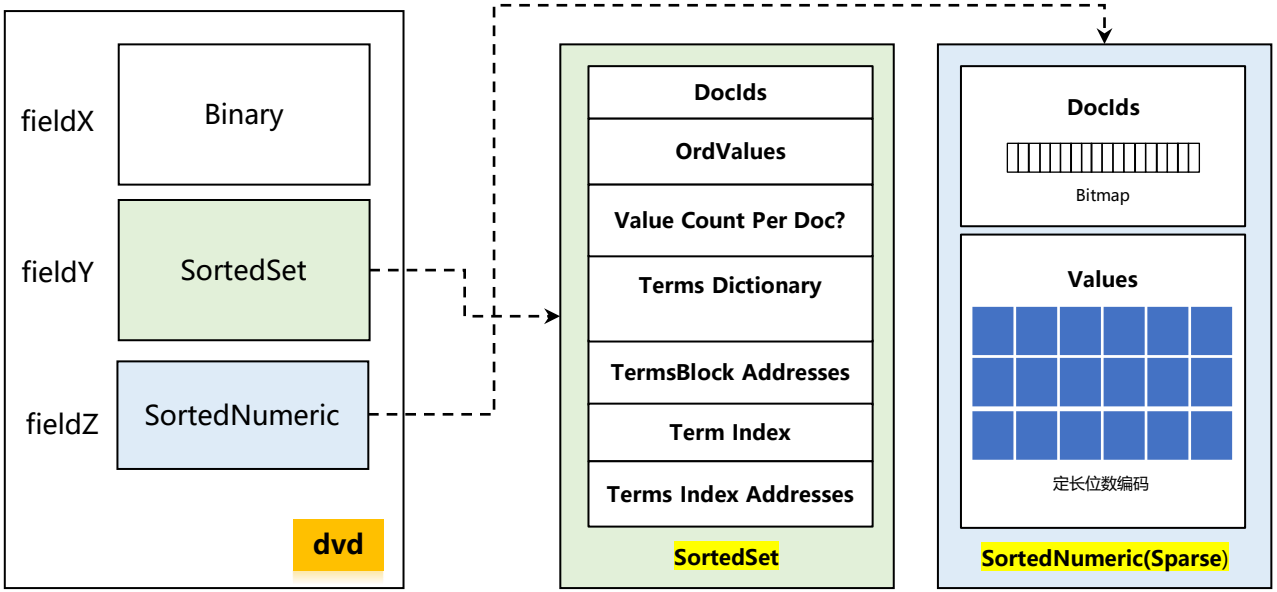
tip

字典文件： Terms Dictionary压缩



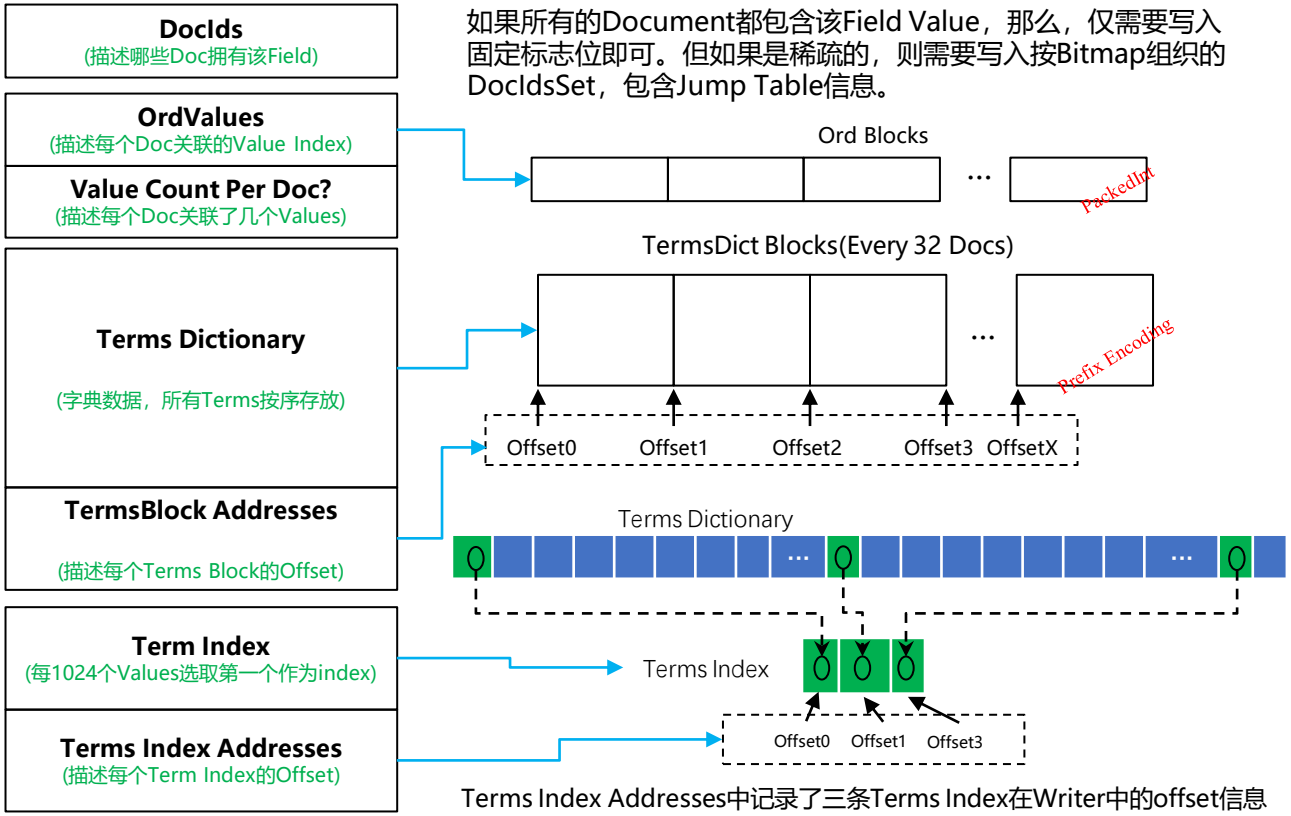
实际效果： Terms Dictionary(tim)文件占用空间下降**40%+**

列存文件： DocValue结构

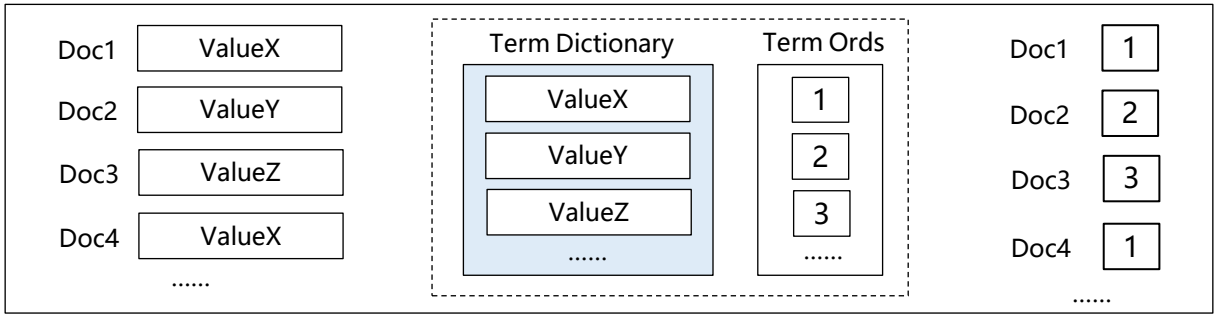


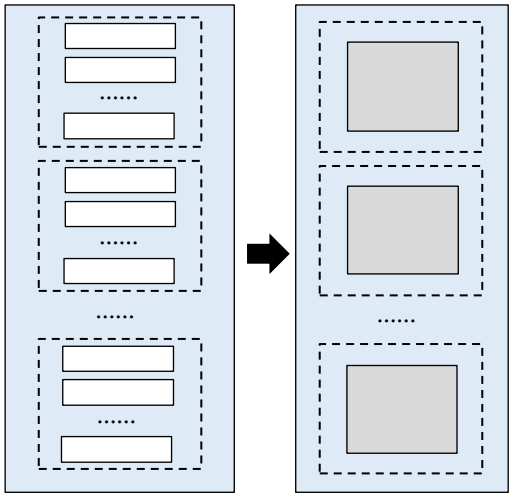
列存文件中的数据按Field组织；不同的Field Type所对应的DocValue的内部结构不同

列存文件： SortedSet DocValue结构



列存文件： SortedSet DocValue压缩优化





Term Dictionary Compression

针对High Cardinality类型的Field，启用压缩。

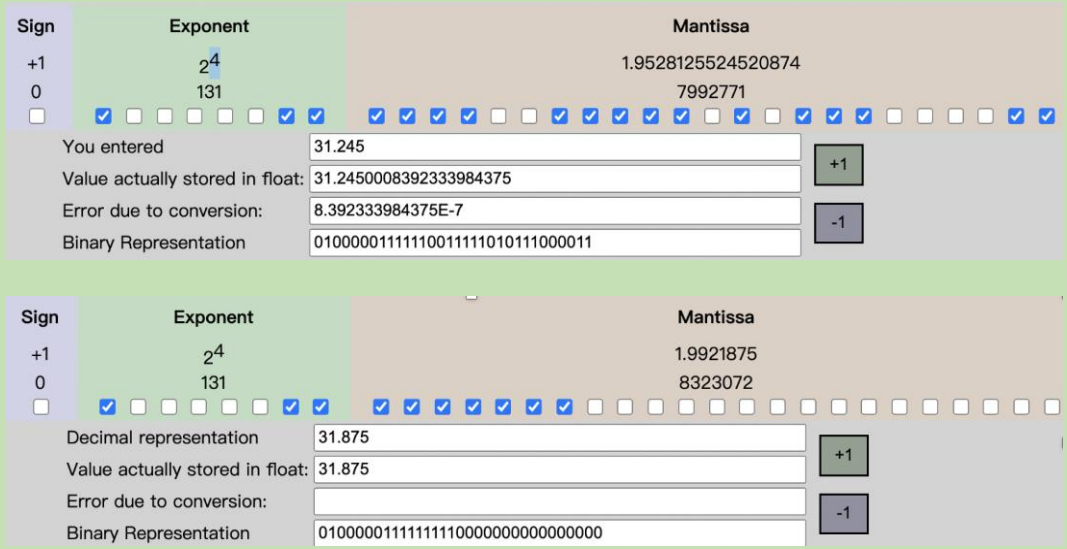
实际优化效果：

压缩算法	优化前	优化后
写入耗时	591972 ms	618200 ms
Merge耗时	270661 ms	294663 ms
dvd文件大小	1.95 GB	**1.15** GB **(↓40%+)**

列存文件：Numeric DocValue优化

Elasticsearch在存储Double/Float类型时，按照IEEE-754标准将其转换成Long值

IEEE-754标准中定义的浮点数表达示例(32-bits)：



数值上相邻的点31.245与31.875，小数部分的二进制表达差异非常大

列存文件：Numeric DocValue优化

Facebook Gorilla论文中针对浮点数存储优化的XOR算法：

Previous Value	12.0	0x4028000000000000
Value	24.0	0x4038000000000000
XOR	-	0x0010000000000000

11 leading zeros, # of meaningful bits is 1

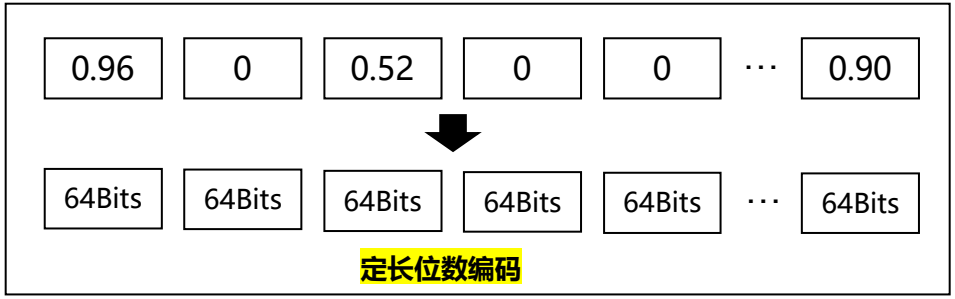
实际测试后的效果并不佳，主要原因有两点：

1. 数据通常是无序的
2. 按块编码，对于随机访问并不友好

图片截取自：Gorilla: A Fast, Scalable, In-Memory Time Series Database

列存文件：Numeric DocValue优化

doc	cpu_usage
1	0.96
2	
3	0
4	
5	0.52
6	0
7	0
.....	.....
9999	0
10000	0.90



基于IEEE-754标准将浮点数转换后的Long值几乎涵盖了整个可能的Long值区间，所以，基于定长位数编码的话，每个浮点数仍需要64位来表达。

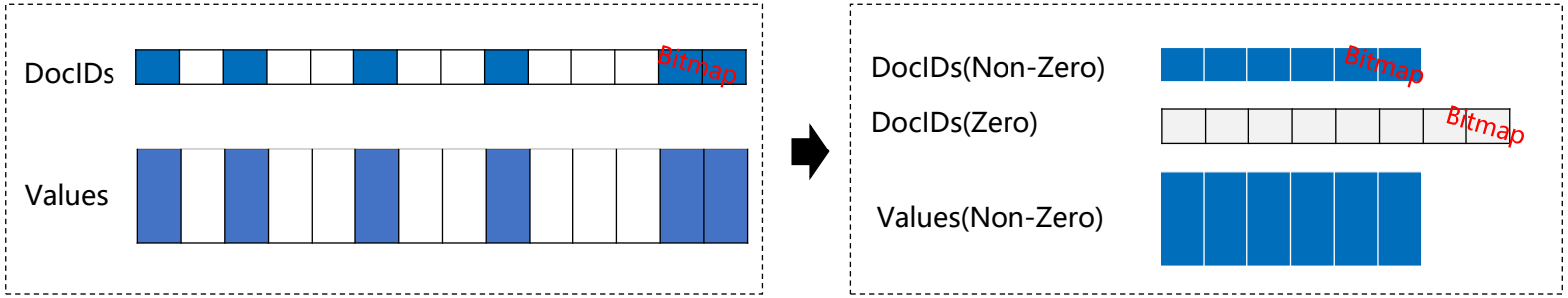
从数据特点来看：某些业务场景下，**0值的占比非常高。**

基于定长位数编码，大部分场景中，**每个0值仍需64位来表达。**

列存文件压缩：Numeric DocValue优化

0值与Null值拥有不同的业务含义，因此，不能直接在写入时忽略存储0值。

考虑到0值是一个非常重要的默认值，在0值占比较高的场景下，我们通过优化Numeric DocValue的底层数据组织结构来降低0值占用存储空间。该优化对于用户读取而言是透明的。以Sparse模式为例：



实际优化效果：

在内部某业务中，有大量的列的0值占比较高。选取了该业务索引的其中一个Segment中某个0值占比较高的Field（0值占比超过了96%），基于该方案，优化后的效果如下表所示：

模式	优化前	优化后
Sparse模式	122.4 MB	11.8 MB (↓90%)
Dense模式	133.3 MB	10.4 MB(↓92%)

细粒度压缩特性控制

按照不同的业务负载/数据访问等特点以及成本控制需求，可灵活选用不同的压缩策略控制。

```
createRequest.settings(Settings.builder()
        .put("index.number_of_shards", shards)
        .put("index.number_of_replicas", replicas)
        .put("index.codec", "custom")
        .put("index.compression.stored_fields", "zstandard")
        .put("index.compression.terms", "none")
        .put("index.compression.sortedset", false)
        .put("index.compression.skip_zeros", true)
);
```

- 行存压缩算法
- SortedSet DocValue压缩
- Numeric DocValue优化
- Terms Dictionary压缩

目录

下一步重点工作

01 智能压缩编码策略

02 针对日志数据的定制压缩算法

03 计算与存储分离

Thanks