导读

本文将以 Bitmap 在 TD 内部的使用场景为切入点，较为深入地探讨不同类型 BitMap 算法的存储结构和基本操作，同时会对不同 Bitmap 的优缺点进行简单介绍。希望通过对 Bitmap 的学习，帮助大家在使用 bitmap 进行海量数据处理的时候，能够知其然更知其所以然。

1. Bitmap 简介

Bitmap 可以理解为通过一个 bit 数组来存储特定数据的一种数据结构，bit 存储的值只能为 0 或者 1，由于 bit 是数据的最小单位，所以这种数据结构往往是非常节省存储空间。此外得益于极其简单的数据结构，bitmap 的压缩性同样十分出色，通常 bitmap 作为一种特殊的索引结构被广泛的应用于数据库和搜索引擎的性能优化领域。近年来随着海量数据处理场景的增多，bitmap 凭借着其存储空间和压缩性能方面的优势，在大数据领域多场景下扮演起举足轻重的角色。

2. Bitmap 在 TD 内部的应用

目前，talkingdata 覆盖的移动终端总量已经超过了 61 亿，对于每一个终端都需要存储不同维度的数据，并在业务场景中进行多维交叉计算，这对系统的 I/O 性能开销和计算效率提出了异常高的要求，而 bitmap 的引入有效地解决了上述问题。

在 TD 内部，已经形成了一套基于 bitmap 结构的数据预处理、压缩存储和内存计算的完整解决方案。比如 TD 移动端 SDK 可以收集每个设备对某些 APP 的安装情况，现在需要记录安装了名称为 A 的 APP 的所有设备，传统的方案是记录下已安装 app A 的设备对应的 tdid 列表，比如 A:[1,2,3,4,5,6,7,8]。假如设备对应的 tdid 采用 long 数据类型，则保存安装了 App A 的设备记录需要 N 个 long，其中 N 是 TD 覆盖的设备终端总数。另一种方案则是构造一个 8-bit（01110011）的数组，将这 8 个设备根据 tdid 号分别映射到这 8 个位置，如果安装了 App A，则将对应的这个位置置为 1，否则置为 0；这样对于一款 APP 的安装情况只需要一个长度为 N 的 bitmap 即可记录。

通过上述存储方案，结合 bitmap 高效的压缩算法，可以节省大量的存储空间。运算的时候，在消耗较少 I/O 资源的前提下，一次性将 bitmap 全部加载到内存进行计算。此时 bitmap 在位运算（AND/OR/XOR/NOT）中的出色性能，天然支持了大数据多维交叉运算需求。比如上述 app 安装记录的例子中，如果想要获取同时安装了 A 和 B 两款 APP 的设备，只需要对 A、B 两款安装设备对应的 bitmap 进行一次位求交运算，然后检查哪些位置的 bit 值为 1，那么该位置对应的设备则同时安装了两款 APP。

随着数据量的急剧增长，为了进一步提高大数据计算系统的性能，TD 将分布式计算引入了 bitmap 计算引擎，把一个需要非常巨大的计算能力才能解决的问题分成许多小的部分，然后把这些部分分配给许多计算机进行处理，最后把这些计算结果综合起来得到最终的结果。

本文会聚焦于 bitmap 自身数据结构和算法的剖析，因此将不再进一步对 bitmap 计算引擎的构架进行深入探讨。

3. RLE bitmap

压缩算法直接决定了 bitmap 存储空间的大小和计算性能的好坏，其中 run-length encoding（RLE）编码技术是一种常用的 bitmap 压缩技术，RLE 的原则是对于重复出现的值，通过值加上重复出现的次数表示，从而达到 bitmap 数据的压缩。常见的基于 RLE 压缩技术的 bitmap 包括 Oracle's BBC、WAH、EWAH、Concise 等，TD 早期大范围采用了 EWAH 作为 bitmap 计算引擎的基础数据结构，因此本文选择 WAH 和 EWAH 两种结构对 RLE bitmap 的实现方式进行介绍。

3.1 WAH

WAH (Word-Aligned Hybrid)是 一种经典的 bitmap 压缩算法。它将未压缩的 bitmap 以 31 个 bits 为单位分组（groups），并且按照两种策略将这些组分为两类：fill groups 和 literal groups。如果一个组内所有 bit 的值都是一样的，那么这个组就叫作 fill groups，例如 0000000000000000000000000000000(或者写作 0^{31}) 就是一个 fill group。此外如果 fill group 中 31 个 bit 的值都是 1，那么可以称其为 1-fill group；如果 fill group 中 bit 值都是 0，那么就称其为 0-fill group。相反地，如果一个 group 中既有 0 又有 1，那么这个 group 就是 literal group。WAH 算法只压缩 fill groups，而不处理 literal groups。WAH 用 32 个 bits（一个 word）表示连续的 fill groups：第一个 bit 为 1 表示存储的是一个 fill groups，第二个 bit 表示这个 fill group 是的类别是 1-fill 或者 2-fill，然后剩余的 30 位表示连续 fill groups 的个数。WAH 同样用 32 个 bits 表示 literal bitmap：第一个 bit 为 0 表示存储的是一个 literal group，剩余 31 位存储了该 literal group 的值。举个例子，假如给定一个未压缩的 bitmap 如下（0^{20} 这种形式表示 20 个连续的 0-bit）

$10^{20}1^30^{111}1^{25}$ （共 160 个 bits）

那么 WAH 算法会将其分为 6 组：$G_1(10^{20}1^30^7)$, $G_2(0^{31})$, $G_3(0^{31})$, $G_4(0^{31})$, $G_5(0^{111}1^{20})$, $G_6(0^{26}1^5)$，并且对上述 6 个 groups 进行编码，用$(010^{20}1^30^7)$表示 G_1，G_2, G_3, G_4 合起来用 $100^{27}011$ 表示，G_5 使用 $00^{111}1^{20}$ 表示; G_6 用 $00^{26}1^5$ 表示。

字对齐技术为带来了一个巨大的优势，那就是 WAH bitmap 可以再不需要将 bitmap 解压的前提下，执行 AND/OR/XOR 等位计算，从而保证计算时候的低内存消耗；而采用 word（计算机的字长，64 位系统就是 64bit）对齐等技术又保证了对 CPU 资源的高效利用。

实际上，任何一种 bitmap 压缩算法都会采用类似的 word/bytes 对齐技术，来减少对内存和计算资源的消耗，真正的不同之处在于如果去定义每一个 word/bytes 的编码含义，以及如何执行每个 word/bytes 的位操作。

3.2 EWAH

EWAH (Enhanced Word-Aligned Hybrid)是 WAH 压缩算法的一个变种，它主要解决了 WAH 对 literal groups 压缩力度不足的问题。不同于 WAH, EWAH 将未压缩的 bitmap 以 32 个 bits 为单位分组(groups)。EWAH 定义了一个标识字（mark word），这个 mark word 的第一位 bit 表示 fill groups 的类型（0-fill 或者 1-fill），接下来的 16 位表示 fill groups 的个数 p，最后 15 位表示 literal groups 的个数 q，EWAH 压缩后的编码都是以一个 mark word 开始。举个例子，假如给定一个未压缩的 bitmap 如下：

$10^{20}1^30^{111}1^{25}$

那么 EWAH 会将其分为以下 5 组：$G_1(10^{20}1^30^8)$, $G_2(0^{32})$,$G_3(0^{32})$, $G_4(0^{32})$, $G_5(0^71^{25})$，然后将 G_1 表示为

$$\underbrace{00^{16}0^{14}1}_{marker}10^{20}1^30^8 \ (p=0,\ q=1)$$

将 G_2、G_3、G_4、G_5 合起来表示为

$$\underbrace{00^{13}1000^{14}1}_{marker}0^71^{25} \ (p=4,\ q=1).$$

4. Roaring Bitmap

即使采用了字对齐的处理，压缩后的 RLE bitmap 随机获取一个位置的值的性能仍然远逊色于未压缩的 bitmap，它检查（check）或者修改第 i 个 bit 对应值的时间复杂度是 O(n)。此外，对于 RLE 格式的 bitmap 而言，其跨过某段数据的能力是有限的，比如在做两个 RLE bitmap 的位 AND 操作时，如果其中一个 bitmap 大部分 bit 的值都为 0，我们可能更希望跳过这些 0-bit 对应位置的计算，但是如果没有特定的辅助索引，RLE bitmap 是很难满足这类需求的。

针对上述问题，Roaring bitmap 应运而生。Roaring bitmap 分别采用不同的结构存储密集和稀疏的数据集，从而获得了比 RLE 更为高效的运算效率。

4.1 Roaring 存储方式

对于长度为 n（n 为一个 32-bit 的整型）的 bit 空间，Roaring bitmap 将其划分为 2^{16} 个桶，每个桶对应一个 container，用 32 位整型的高 16 位作为 container 的索引（一级索引），每个 container 只存储公用该一级索引的整型的低 16 位值。

如果一个桶中存储的整型数少于 4096 个，那么这个桶中的容器就是一个 short 类型的有序数组；否则这个桶中的容器就是一个长度为 2^{16} 的 bitmap。换言之，Roaring bitmap 采用了两种不同类型的容器进行数据存储：一个 short 类型的有序数组存储稀疏数据，一个 bitmap 用来存储密集数据，这些容器由一个动态维护的 container array 维护。

举个例子来具体说明 Roaring bitmap 的存储结构。假设设定以下整型位置的 bit 为 1:

1) 62 的前 1000 个整数倍对应的整型，即 0、62、124、186、248...61938;
2) $[2^{16}, 2^{16}+100)$区间内的所有整型;
3) $[2\times 2^{16}; 3\times 2^{16})$区间内所有的偶数；

在 Roaring Bitmap 中，1)和 2)将会分别存储在一个 short 类型的数组中，3)将存储在一个 2^{16} 位的 bitmap 中，如下图所示。

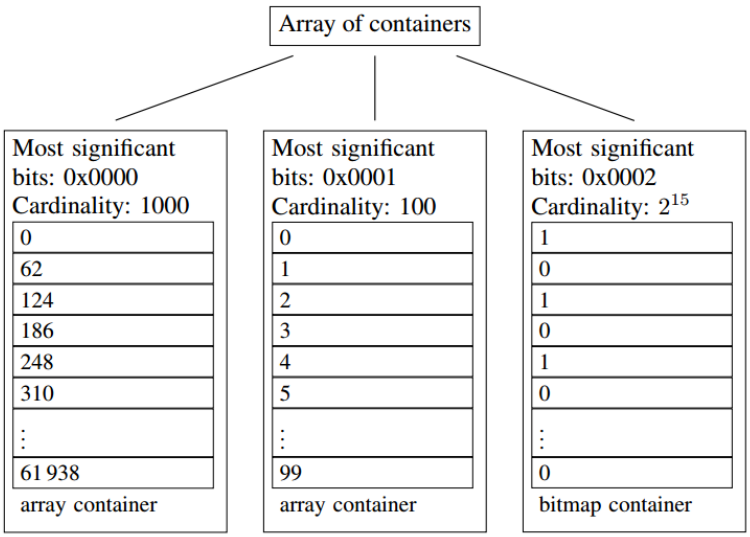

图 Roaring Bitmap 的存储结构

每个 Roaring 容器都会维护一个计数器用以存储该容器的基数，从而能够通过计数器求和快速获得一个 Roaring Bitmap 的基数大小。容器计数器的存在，还能够快速确定一个整型所属的容器，以及支持 bitmap 某个位置值的高效查询。

由于需要维护容器和及容器数组，在极端情况下，Roaring bitmap 可能会消耗超过 16bit 的空间存储一个整型。实际上，只要容器的个数小于存储的整型的个数，Roaring 的空间利用率远不会如此低效，甚至可以认为当数据集的密度小于 0.1%的时候，相较 short 型数组而言 bitmap 就已经不是一种合适的数据存储结构了。

在上述结构的基础上，Roaring 进一步采用了一些优化措施，比如对于密度较大的数据集，当容器中整型的个数大于$(2^{16}-4096)$的时候，可以通过一个 short 类型的数组存储 bit 值为 0 的整型序列代替原本的 2^{16}位的 bitmap；对于 bitmap 容器，可以采用算法对连续的整型序列进行压缩。

4.2 Roaring bitmap 相关操作

对于 Roaring Bitmap 的检索操作，比如判断某个整数 X 是否存在，首先通过 $X/2^{16}$ 获得一级索引，定位到所在的桶。如果桶容器是一个 bitmap，检查 $X \bmod 2^{16}$ 这个 bit 位，如果桶容器是一个数组，使用二分查找。

Roaring bitmap 的逻辑运算也是以桶为单位进行，由于存在 Bitmap 和 Array 两种桶容器，因此所有的逻辑运算(AND/OR/XOR)都应该是以下三种操作类型之一：

1) Array vs Array，直接 merge 两个 Array；
2) Array vs Bitmap，遍历 Array，把 Array 中的值依次映射到 bitmap 上并操作；
3) Bitmap vs Bitmap，直接按位操作即可；

RLE bitmap 中并没有对稀疏数据集的存储和运算采取过多的优化，Roaring Bitmap 采用了与 RLE 完全不同的思路设计了 bitmap 的存储结构。在绝大多数场景下，Roaring 相较 RLE 而言都具有更出色的计算效率。关于二者的性能比较，将会另开一篇文章介绍，在此不做深入探讨。

5. 总结

Bitmap 的引入，避免了系统将大量的 CPU 和内存消耗在数据装载，数据过滤等环节，有效降低了磁盘 I/O 和网络 I/O 的开销，同时支持高速的、灵活的多维交叉分析。对不同类型 bitmap 的了解和掌握，将有助于我们在实际生产场景下，能够快速准确地选择合适的 bitmap 模型；而对不同类型 bitmap 的数据结构和基础算法的学习和实践，有助于我们在开发工作中开拓思路、持续进步。