第五章 索引

引 言

索引是指按照表中某些关键属性或者表达式建立元组的逻辑顺序，它是由一系列表元组的标识号组成的一个列表。使用索引可以快速访问表中的特定元组，被索引的表称为基表。索引不会改表表中元组的物理顺序，仅仅将对于元组的逻辑排序保存在索引文件中。当基表文件中的元组被修改或者删除的时候，其对应在索引文件中的逻辑顺序也会及时更新，以确保能够准确地找到所需要的数据。几乎所有的数据库都离不开索引技术的支持，该技术为快速查询提供可能。在 PostgreSQL 中实现了大量的索引技术，Greenplum 数据库是基于 PostgreSQL 的，因此在 Greenplum 中实现了 PostgreSQL 中大多数的索引技术，也存在一些部分的修改完善，同时 Greenplum 也定义了一些自己的索引。

传统的索引技术包括 B-Tree 索引[1]、Hash 索引、Gist 索引、GIN 索引等等。基于 SSD 的存储介质和内存数据库的出现对传统的索引技术提出了新的挑战。SSD 有着高速的读效率但是低效的写效率，以及“out-of-place update”的特性。传统的 B-树索引和 hash 索引主要是针对机械硬盘，需要重新设计来适应 SSD 的这些特性。最近 Google 发表的一篇文章 The Case For Learned Index Structures[2], 试图从如今比较流行的机器学习、深度学习入手，来解决传统索引技

术面临的挑战。这是一个开创性的工作，相信该技术在未来几年将会对数据库系统设计产生巨大影响。

本章将会从原理技术、在 Greenplum 上的实现、动手实现三个方面来详细介绍各个索引结构。在最后会介绍索引技术现阶段发展的趋势以及最新研究成果，并给出相关参考文献。

5.1 索引基本原理

Greenplum 中提供了 5 种索引方式：唯一索引、主键索引、多属性索引、部分索引以及表达式索引。

1）唯一索引：如果索引声明为唯一索引，那么就不允许出现多个索引值相同的元组。唯一索引可以用来强迫索引属性的唯一性，或者多个属性组合值得唯一性。唯一属性通过在创建索引命令中加上 UNIQUE 关键字来创建。一个多字段唯一索引认为只有两个元组的所有被索引属性值都相同的时候才是相同的，这种重复的元组定义才会被拒绝。目前只有 B-Tree 可以创建唯一索引。唯一索引的创建语句声明如下：CREATE UNIQUE INDEX idx_name ON tablename(column)。

2）主键索引：如果在一个表上定义了一个主键，那么数据库就会在主键所在属性上创建唯一索引。主键索引是唯一索引的一个特例。

3）多属性索引：如果一个索引定义在大于一个属性上，就称其为多属性索引。目前在 Greenplum 中的 B-Tree、Gist 和 GIN 支持多属性索引。一般来说，在单个属性上创建索引就足够了。如果说表的查询模式非常固定，那么创建多属性索引是可以提高查询速度的。否

则，超过 3 个属性的索引是没有意义的，几乎没有太大的用处。当对一个表创建多属性索引时，对于表中的一个元组，会依次去读取被索引的属性的值，用这些值一起作为该索引的索引键值。多属性索引中不仅可以使用表中的属性，也可以使用函数或表达式计算得到的值。

4）部分索引：部分索引是建立在一个表的子集上的索引，而该子集由一个条件表达式定义（表达式即部分索引的谓词）。该索引只包含那些满足谓词（表达式）的元组。部分索引的创建声明如下：CREATE INDEX stu_name_idx ON student(name) WHERE (id>1 AND id<255)。该语句在 student 表中，对 id 在 1 到 255 中的元组的 name 属性创建索引，这种索引就是部分索引。使用部分索引能够减小索引的规模，提高索引的查询效率。

5）表达式索引：在所属性索引中提到，索引并不一定要建立在一个表的属性上，还可以建立在一个函数或者从表中的一个或多个属性计算出来的标量表达式上。索引创建声明如下：CREATE INDEX stu_low_name_idx ON student(lower(name));上面的声明语句是在 student 表的 name 字段通过小写函数来创建表达式索引。表达式索引只有在查询时使用与创建时相同的表达式才会起作用。上述创建语句对应的查询语句可以为：SELECT * FROM student WHERE loewer(name)=’jack’;因为在创建索引过程中会根据表达式计算出实际索引的键值，在更新索引时会使用表达式进行重新计算，因此会导致插入或更新元组时效率变慢。

索引在逻辑上改变了元组的组织方式。如果不使用索引，数据库

需要逐行对表的进行扫描。使用了索引之后，数据库就可以快速定位了匹配的元组，大大提高数据查询的性能。而不同的索引方式在不同的查询场景下效率是不同的，如下面介绍的 B-tree，Hash 等方式由于组织方式的不同，适用的情况也不相同，而数据库会选择最合适的索引方式来响应用户提交的查询请求。下面的章节会对这些索引方式的原理进行介绍，以及在什么情况下数据库会使用这些索引方式。

5.2 索引相关技术

常见的索引技术包括 B-Tree 索引、Hash 索引、Bitmap 索引、Gist 索引以及 GIN 索引等。本小节将首先介绍一些已经在各种数据库中使用的比较成熟的索引技术，然后会介绍一些最新的科研成果。

5.2.1 B-Tree 索引

知识回顾

在 PostgreSQL 里提供了 B-tree 索引类型，其结构类似于 B+树。B-tree[1]适合支持比较查询以及范围查询。在一个建立了 B-tree 索引的属性涉及比较操作符进行比较的时候，会优先考虑 B-tree 索引。首先我们先回顾一下 B+树的知识。

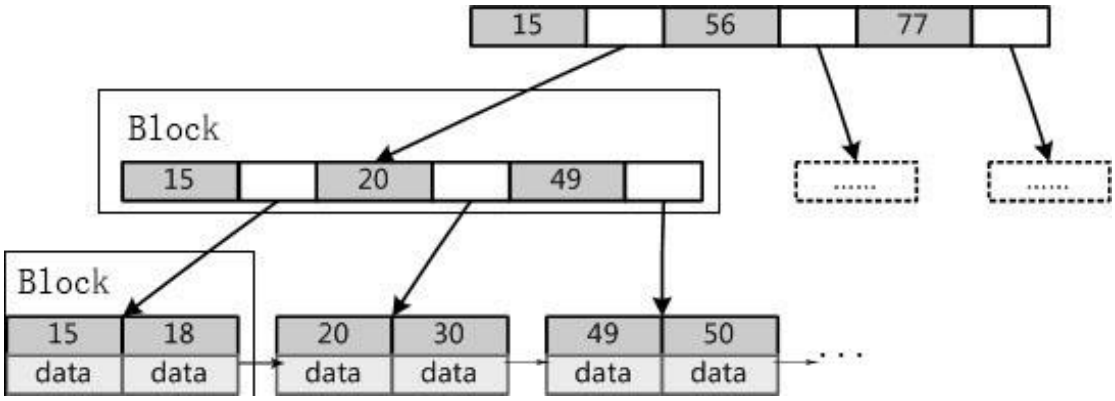

图 5-1 B+树示例

如图 5-1 就是一棵 B+树，B+树有一下特点：1. 非叶子结点的子树指针与关键字个数相同；2）所有的关键字都出现在叶节点中，查找必须要在叶节点命中；3）叶节点存在指向右兄弟节点的指针。

在 B+树上进行查找，插入，删除的过程基本与 B 树相同，只是查找必须要在叶节点命中。不管在内部节点命中与否，查找过程都是从根节点到叶节点的一条完整路径。

由于 B+树的关键字在叶节点都会出现，适合文件索引系统。1981 年 Lehman 和 Yao 发表在《Acm Transactions on Database Systems》上的论文“Efficient Locking For Concurrent Operations On B-Trees”提出了一种 B+树的变种，增加了内部节点指向右兄弟节点的指针，即图 5-2 的 P_{2k+1}：

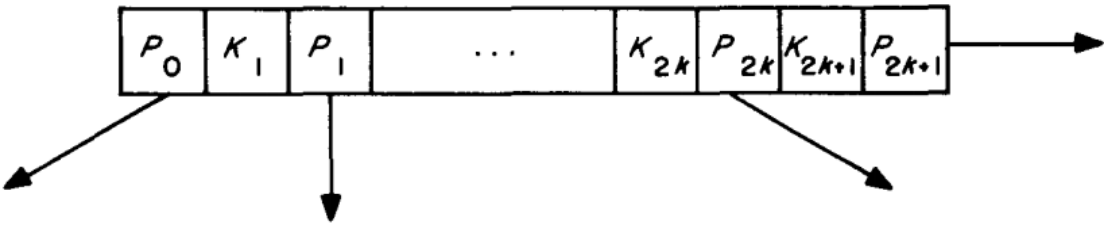

图 5-2 B^{link}-树示例

B^{link}-tree 还对除每层最右节点外的其他节点设置了一个“high-

key”（K_{2k+1}），high-key 只是保存最大的关键字允许的范围，不指向任何实际的元组。当要插入的索引关键字大于 high-key 时，向右边的节点移动寻找合适的插入节点。而对于指针 P_i，指向的子节点包含的键值的范围为（K_{i-1}，K_i）。

B-Tree 在 Greenplum 中的实现

在 Greenplum 中，索引同样是按照第三章提到了文件页的结构进行存储。每一个文件页都会有一个 level 的标志表示该文件页在 B-树中的层数。结构如图 5-3 所示：

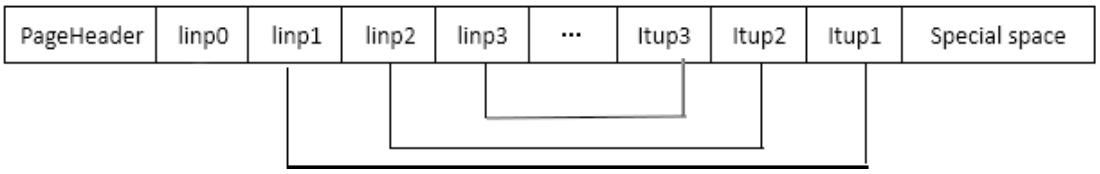

图 5-3 索引页结构

实际上，linp0 是属于 PageHeader 的，保存的是 high-key。linp1、linp2、linp3 是指向 itup1，itup2，itup3 的指针。itup1-3 存储的是实际的索引记录。在页面的填充过程中，linp0 没有被赋值。当页面填充完之后，将会判断当前页是否是该层的最右节点做以下的操作：

1） 若该页不为最右的节点 首先将 itup3 赋值到右兄弟节点中，然后将 linp0 指向 itup3，删除 linp3。索引是一种有序的结构，itup3 是最大的索引项，所以 high-key 保存了 itup3 的值。但是 high-key 并不指向实际的元组，所以要先将 itup3 赋值到右兄弟节点中，删除 linp3.

2） 该页为最右的节点最右节点不保存 high-key 值，这时 linp0 指向的是 itup 同理所有的 linp 递减一个位置，itup3 同样不再使用。

下面对索引的创建，查询，删除的过程做一下介绍：

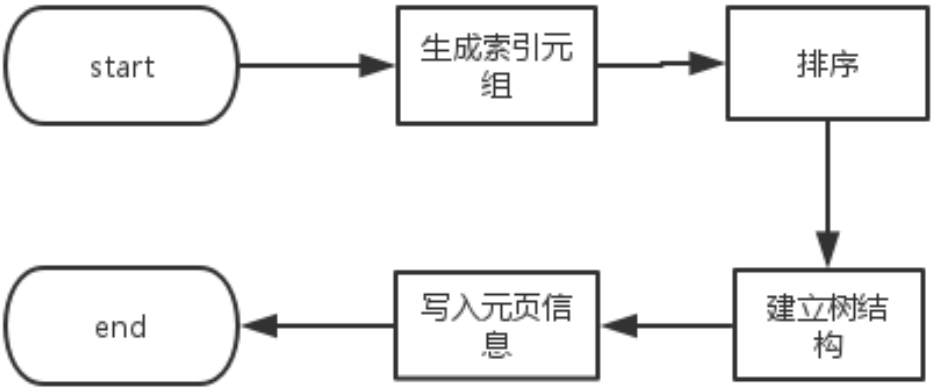

图 5-4 B-tree 索引的创建

在用户创建一张索引表时，首先对基表进行扫描，将表元组封装成索引元组，然后对索引元组进行排序，如果为唯一索引的话会同时检查索引元组是否重复。之后会在排好序的索引元组上建立一棵 B-树结构，更新元页信息。元页主要说明该 B-tree 的版本，根节点位置等信息。建立 B-树的流程如图 5-5 所示：

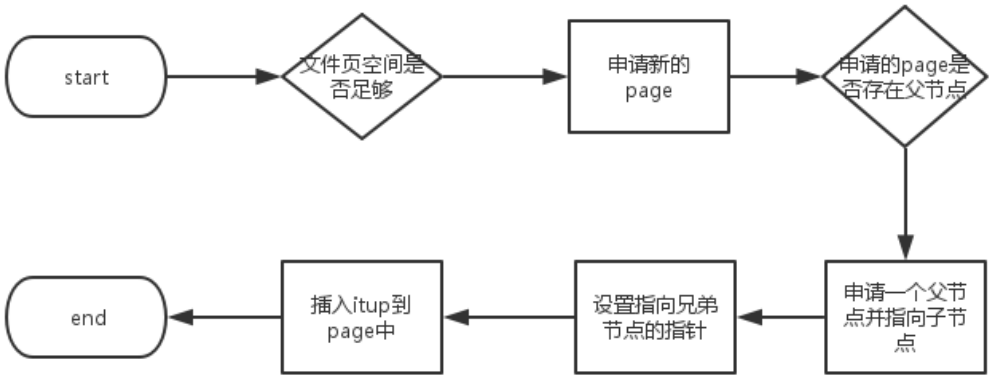

图 5-5 创建一棵 B-tree

创建了索引之后，当表有新的元组插入时，索引也需要插入相应的记录。首先将表元组封装成索引元组，然后沿着 B-tree 往下找到合适的插入节点。找到正确的节点后，若该索引是唯一索引，则在节点中进行验证是否重复。如果不存在重复就直接进行插入。

在对索引进行插入操作时，如果是随机查找，则从树的顶端开始进行查找，一直查找到叶节点，即使在内部节点命中也要继续向下查找。如果是范围查找，则根据上一次查找的位置直接命中下次查询的位置。

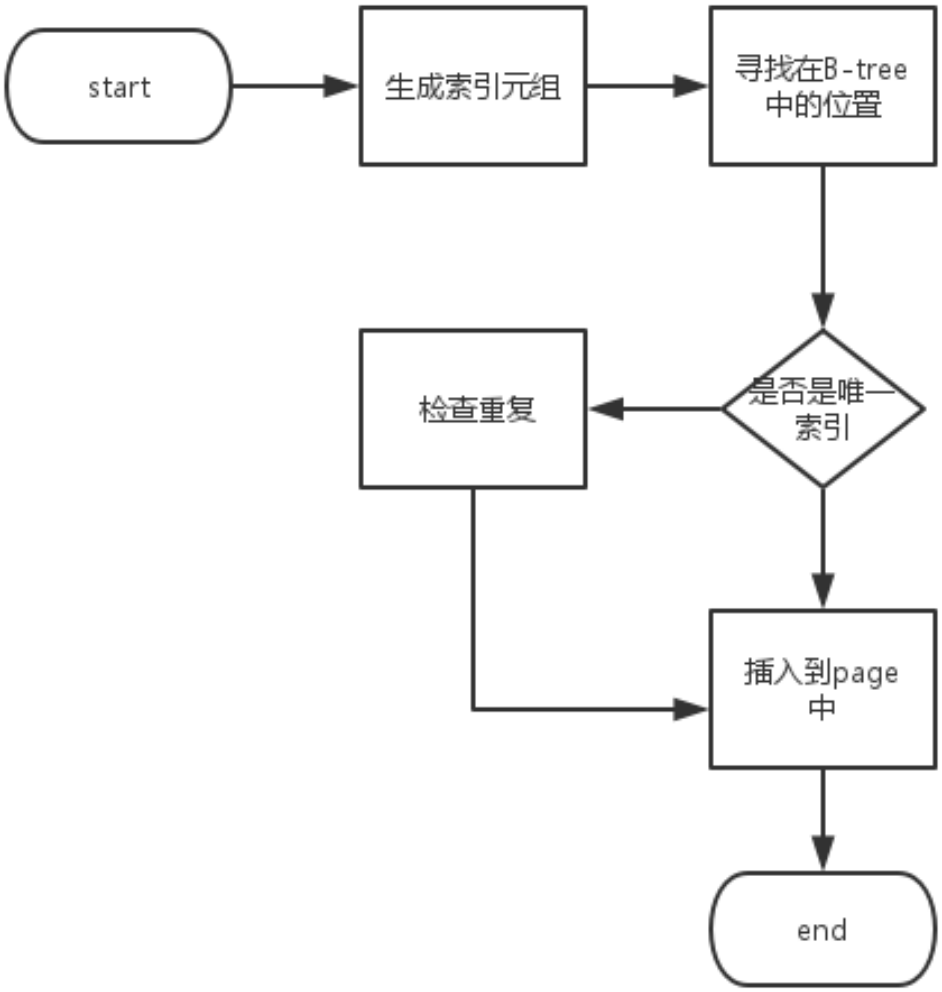

图 5-6 B-tree 索引的插入

删除操作从叶节点开始查找，查找到节点后进行删除操作然后对节点进行调整。在删除了基表中的元组之后，系统并不会立即删除该表元组对应的索引元素，而是在 VACUUM 的时候删除。

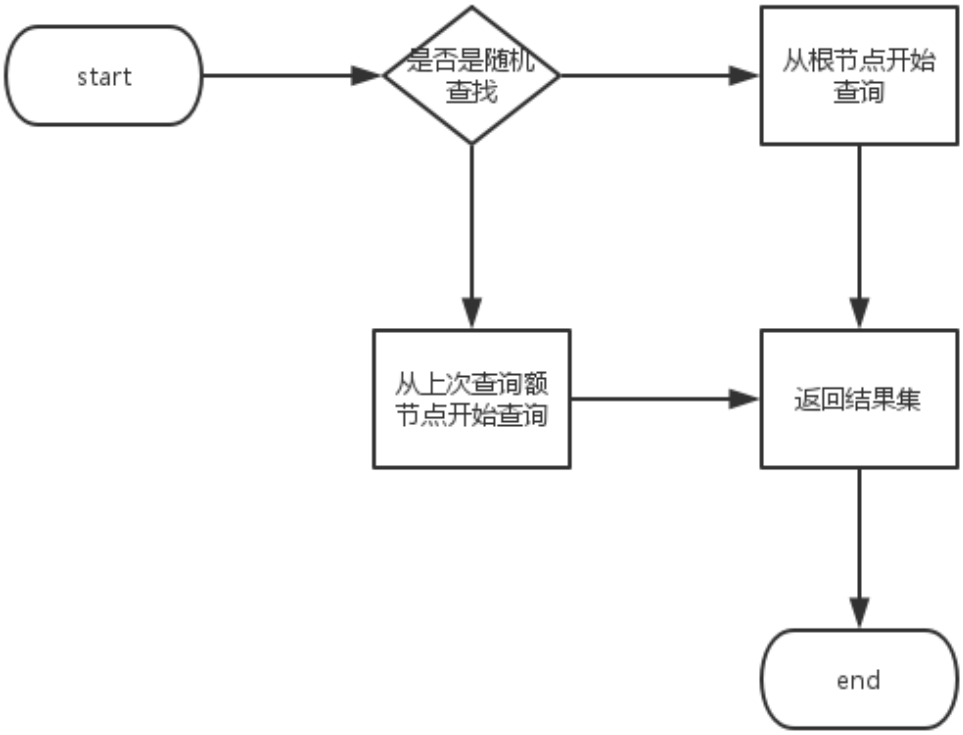

图 5-6 B-tree 索引的查询

图 5-7 B-tree 索引的删除

5.2.2 Hash 索引

知识回顾

在 Hash 索引中，首先对几个关键的要素进行定义：

h：hash 函数，将查找键通过 h 转换成 hash 值，存储在一个 size 为 B 的桶中。记查找键为 K，元组存储的地址为 addr，转换的表达式

为 addr=h(K)。

碰撞：碰撞主要分为两种，一种是函数计算出的 addr 相同，这类碰撞是 hash 函数要解决的问题。另一种是计算的 addr 存储在相同的桶中，解决这种碰撞方法有一次探测，二次探测，开链等。Greenplum 是以开链解决这个问题的。

静态 hash：桶的数目在指定之后就不能变化。

动态 hash:　桶的数目随着元组的插入可以增长。动态 hash 主要用两种：

可扩展 hash:

可扩展 hash 的结构如图 5-8 所示。可扩展 hash 为桶增加了一个中间层，也叫做目录层。如果索引项在桶中溢出，目录项将会翻倍，而溢出的桶会进行分裂。这里 D 指的是全局的深度，L 指定是局部的深度。每一层可表示的数目为 2^k，如目录层是 2^2 可以表示 4 个目录项，桶最多可容纳的数量是 2^2。

在图 5-8 的基础上我们插入索引项。索引项的关键字分别为 data1，data2，经变换后为 h(data1)=13，h(data2)=20.用二进制表示（13）$_{10}$=（1101）$_2$，（20）$_{10}$=（10100）$_2$。因为全局深度为 2，只看后两位。对于 13，其二进制的后两位是 01，插入第二个桶。

对于 20，其二进制的后两位是 00，应该插入第一个桶，但此时一号桶已经满了，于是桶的数目进行翻倍，局部深度变成 3。此时局部深度大于全局深度，目录页需要翻倍，全局深度也变为 3。随

后一号桶进行分裂，数据需要进行局部的重组。重组后的结果如图 5-9 所示：

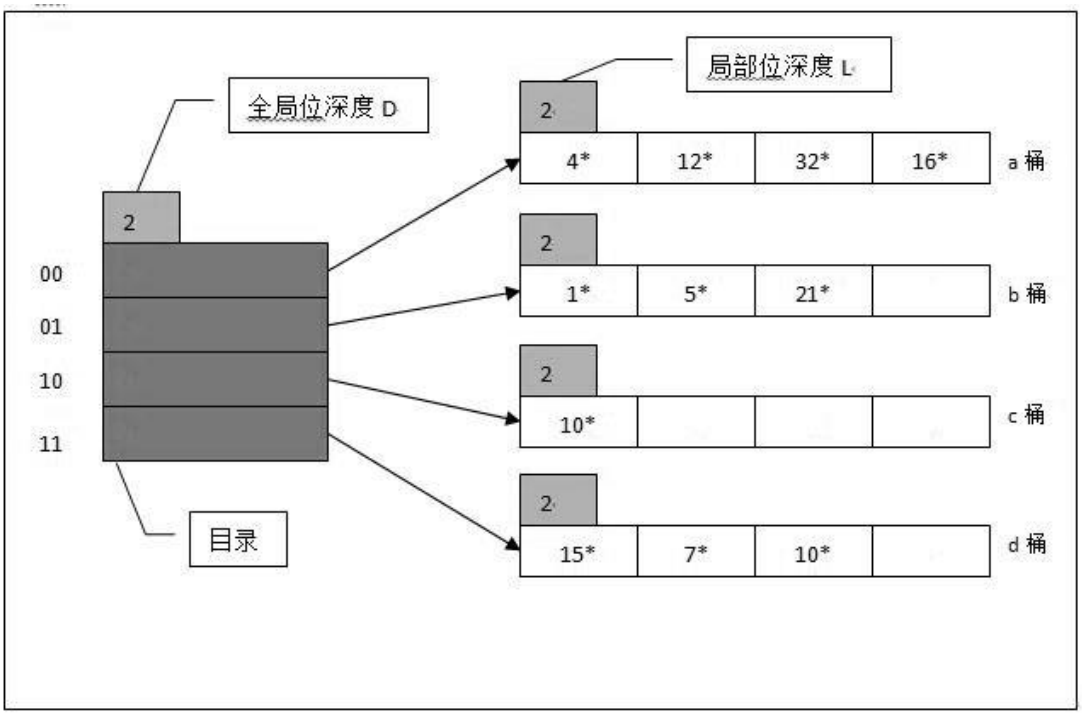

图 5-8 可扩展 hash

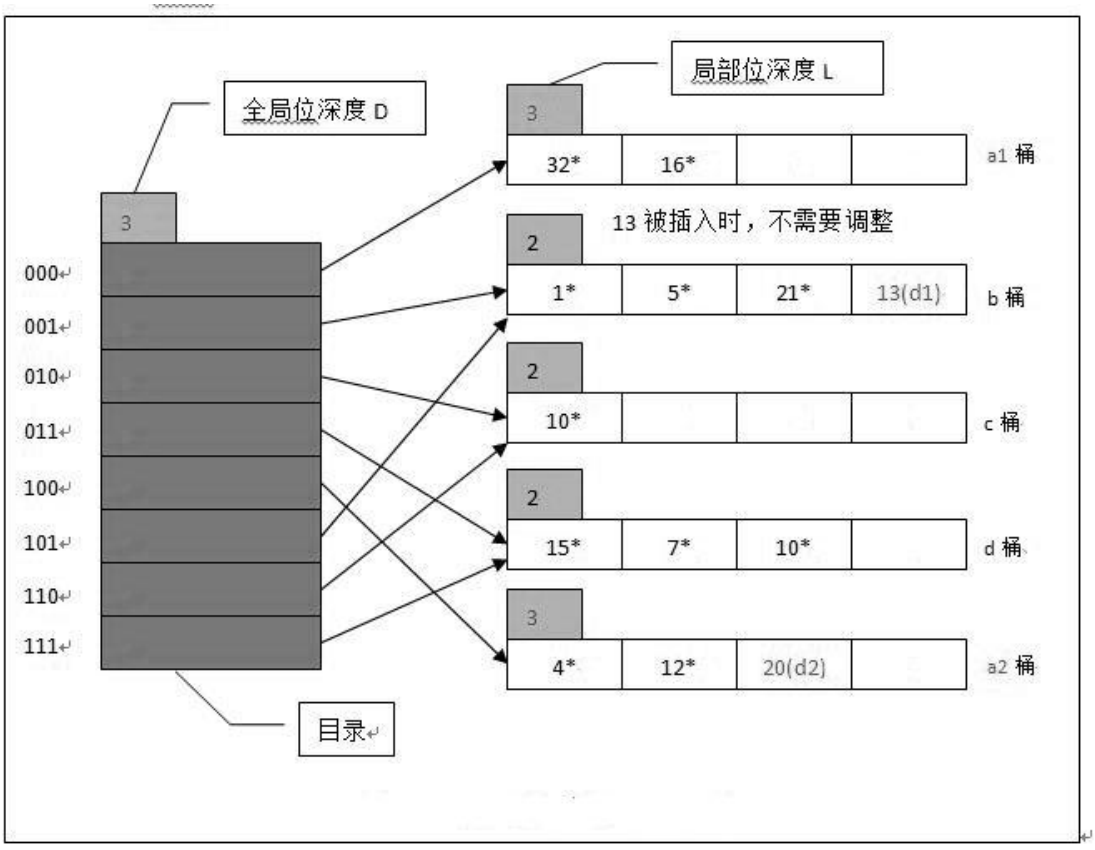

图 5-9 插入数据 data1，data2

线性 hash：线性 hash 也能随着数据的插入动态地改变桶的数据。但不同于可扩展 hash，线性 hash 不需要目录项，而且桶的分裂有一定的顺序。

线性 hash 的分裂条件可以是桶的空间达到了实现时定义的阈值，也可以当桶满了再进行分裂。各个桶按照编号的顺序轮流进行分裂，当一次分裂完之后分裂从头重新开始。由于桶的分裂是有一定的次序的，与当前哪一个桶满了无关，所以允许“溢出页”的存在，等到轮到当前桶进行分裂之后再重新分配数据。

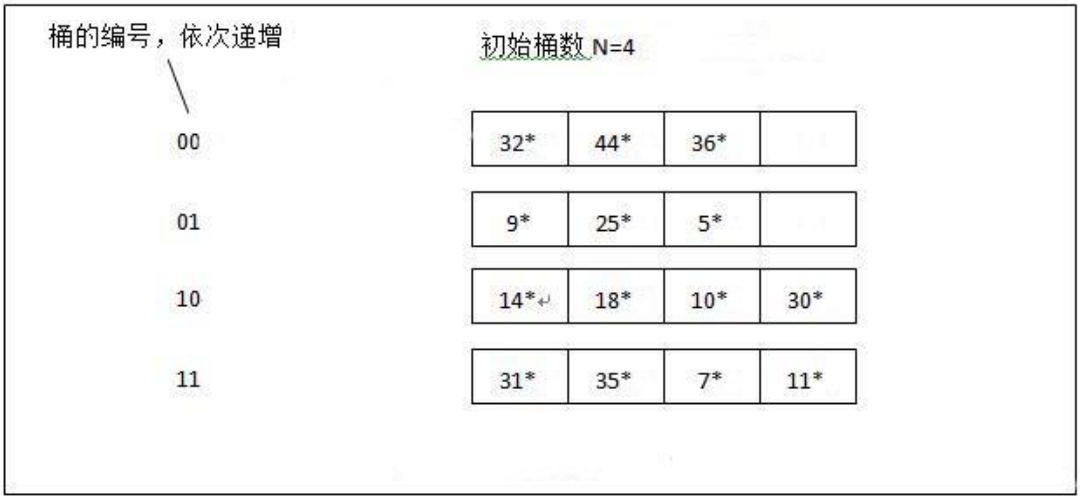

图 5-10 线性 hash 的结构

图 5-10 是一个线性 hash 表的示例。假设我们当前将要分裂的是 00 号桶，插入 h(data1)=37，h(data2)=43 两组数据。经转换为二进制之后 37 进入 01 号桶，此时桶可以继续插入，我们假定只有当桶满才进行分裂。当插入 43 时，由于 43 进入 11 号桶，触发了分裂条件，00 号桶进行分裂，11 号桶增加溢出页。重组后的数据分布如图 5-11：

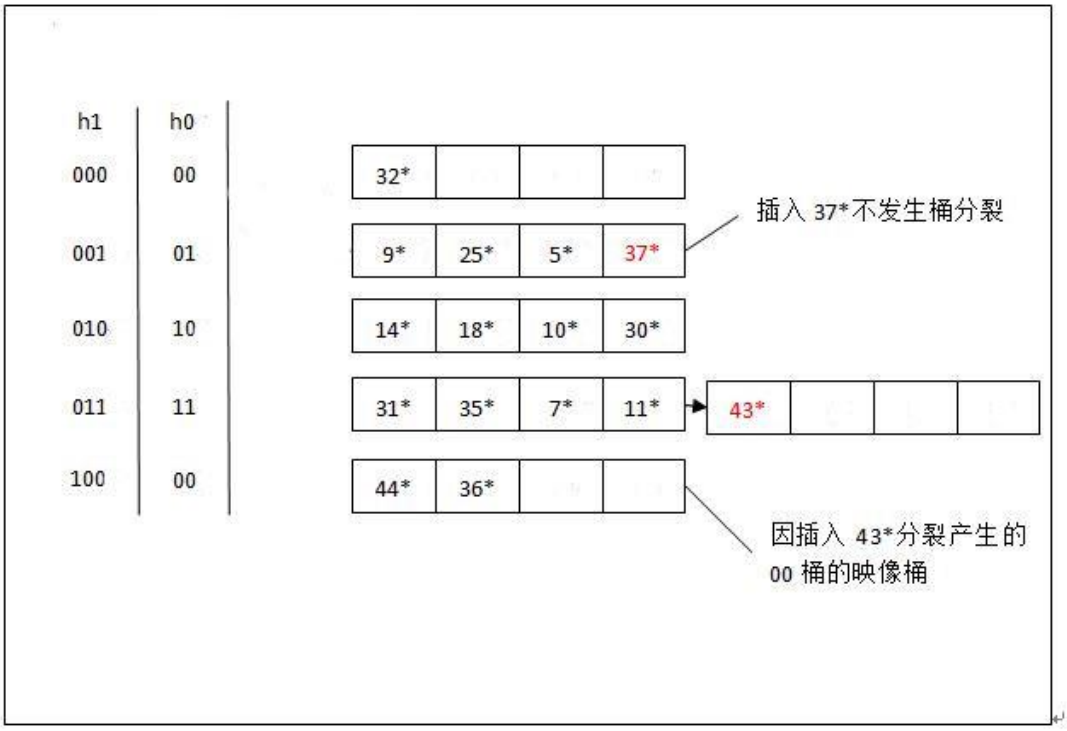

图 5-11 插入数据 data1，data2

需要注意这里分裂的是 00 号桶，因为桶的分裂是有顺序的。在 00 号桶分裂之后下一次的分裂是 01 号桶。11 号桶会暂时产生一个溢出页，一直等到 11 号桶分裂。

Hash 索引在 Greenplum 中的实现

在 Greenplum 中，考虑到 OLAP 的特性，并不支持 hash 索引，但是在后台的代码中定义了 hash 索引的结构，根据用户的需要可以自己进行扩展，这一部分的代码跟 PostgreSQL 是一样的。

下面介绍一下 PostgreSQL 中对于 hash 索引的实现方式，我们可以参考对 Greenplum 进行扩展。hash 索引在随机查找中效率很高，会优先使用：

hash 索引的页面结构跟 B-tree 的组织方式类似。但是在 hash

索引中有四种不同的页面：元页，桶页，溢出页和位图页。

元页：元页是当前 hash 索引的 0 号页，记录了 hash 索引的版本号，被索引的基表的元组号等相关信息。

桶页：桶页存放的是索引元组。每个 hash 索引由一个或多个桶组成。每个桶的第一页称为桶页。在桶页之后可能会有一个或多个溢出页。桶页和溢出页以及溢出页和溢出页之间通过双向链表链接。桶页是成组分配的。如果第一次分配 2 个桶，那么第二次将分配 4 个桶，依次类推。但分配的桶不会立即被使用，只是使用将要分裂的桶，其他桶暂时不会分裂。而且一次分配的桶在物理存储上是连续的。

溢出页：如果当前的桶已经满了但是索引没有触发分裂条件时，当前的索引元组会以溢出页的形式存储起来。等到分裂之后溢出页可能会进行回收，但内存不会返还给操作系统，而是等待 hash 索引的下一次的使用。

位图页：位图页记录了溢出页以及位图页本身的使用情况。上面说过，桶的分裂可能会引起溢出页的回收，数据库会标记该溢出页再次可用。在位图页中，0 表示可用，1 表示不可用。

图 5-12 是一次具体的分裂过程之后产生情况。

桶一	溢出页一	溢出页二
桶二		
桶三	溢出页三	
桶四		

页面	元页	桶一	桶二	位图页	溢出页一	溢出页二	桶三	桶四	溢出页三
块号	0	1	2	3	4	5	6	7	8

图 5-12 hash 索引的逻辑表示及在 block 的分布

在 Greenplum 中，创建一张 hash 表的过程如下：首先初始化该表的元页、初始化桶以及位图页。然后对基表进行扫描生成索引元组，然后将索引元组插入到 hash 表中并更新元页信息。过程图示如图 5-13：

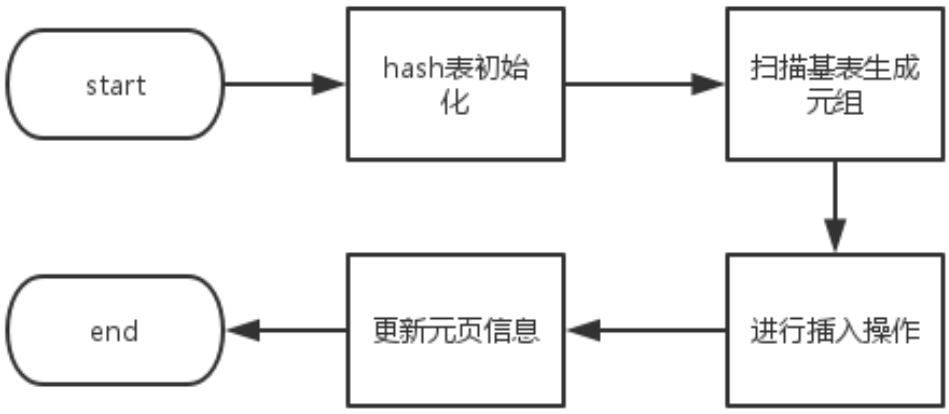

图 5-13 hash 索引的创建过程

对元组进行插入时，首先读取 hash 表的元页信息，然后判断元组应当被插入到哪一个桶中。如果桶中没有足够的空间进行插入，则申请溢出页并插入到溢出页之中。然后更新元页信息并根据元页信息再选择是否进行桶的分裂操作。

5.2.3 Bitmap 索引

知识回顾

位图索引是 Greenplum 添加的索引方式，PostgreSQL 没有这种索引方式。位图的基本概念是用一个位（bit）来标记某个数据的存放状态，由于采用了位为单位来存放数据，所以节省了大量的空间，也被叫做 bitplane。　　位图索引是一种使用位图的特殊数据库索引。

位图索引主要针对大量相同值的列而创建(例如：类别，操作员，部门 ID,库房 ID 等),索引块的一个索引行中存储键值和起止 Rowid,以及这些键值的位置编码,位置编码中的每一位表示键值对应的数据行的有无。如图 5-14，假设我们有一张表的属性是性别，有“男”和“女”两种属性值，有 5 个元组，属性值依次为“女女男女男”。

男	1	5	00101
女	1	5	11010

图 5-14 用位图表示属性值

在一个块可能指向的是几十甚至成百上千行数据的位置。这种方式存储数据,相对于 B-Tree 索引,占用的空间非常小,创建和使用非常快。当根据键值查询时,可以根据起始 Rowid 和位图状态,快速定位数据。当根据键值做 and,or 或 in(x,y,..)查询时,直接用索引的位图进行位运算,快速得出结果行数据。

假设一个 100W 行的表有一个字段会频繁地被当做查询条件，我们会想到在这一列上面建立一个索引。但是这一列只可能取 3 个值。那么

如果建立一个 B-树索引（普通索引）是不行的，因为无论查找哪一个值，都可能会查出很多数据。如果建立一个位图索引，可以快速地根据要查询的值取出对应的元组，效率很高。

Bitmap 在 Greenplum 中的实现

在 Greenplum 中，位图索引由元页，LOV(List Of Values)页和位图页组成。LOV 页存储一组被区分开的属性值，关联的位图页的元信息以及指向位图页的指针。在数据库的实现中，属性值会被包装成 itemData 类型的数据结构，LOV 页通过维护这些 itemData 来管理位图。为了快速定位到这些 itemData，数据库同时为这些属性值创建一个堆表，为堆表建立一个 B-tree 结构，从而快速找到该 itemData。

位图页存储着对应的位图信息。位图页由两部分组成：header words and content words。每一位 header words 和 content words 相对应，如果 header words 为 1，表明它对应的 content words 是被压缩的。在位图页的 special space 中，存储着指向下一页位图的信息，从而维护对每个属性值的位图链。

下面介绍一下 bitmap 索引的构建。首先创建位图的元页以及第一块 LOV 页，然后扫描整个元组页，进行插入。这一部分跟正常索引的创建类似。（在这里对 Greenplum 的索引创建过程做一下特别的说明。在索引的扫描之后，数据库会根据用户指定的索引创建方法--am 方法，存储在 pg_am 中--调用相应的方法，这里的代码强调了

代码的可复用性。

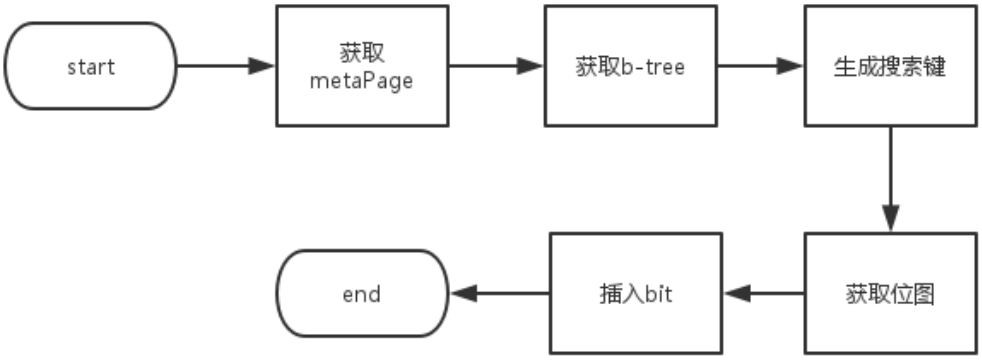

图 5-14 bitmap 的插入过程

在位图索引的插入过程中，首先获取元页，然后根据元页信息获取 LOV 页的堆表和 b-tree。根据用户所指定的索引键生成搜索键，按照搜索键在 b-tree 中快速找到对应的位图，然后在位图中插入一个 bit。过程如图 5-14 所示。

5.2.4 Gist 索引

Gist 索引(Generalized Search Tree)并非一种具体的索引策略，它定义了一个基础的模板，允许用户在这个模板之上开发自己的索引策略。Gist 提供了一个高度的抽象，在这个抽象的基础上，用户不用理解数据库内部的工作模式，只需要对 Gist 提供的一系列结构进行实现，其他的任务如并发，日志都由 Gist 完成。

Gist 是一种平衡的，树状结构的访问方法，由加州大学 Berkeley 分校开发，支持研究人员对新的数据类型开发实验索引。除根结点的扇出数在 2 和 M 之间外，每个节点的扇出数在 kM 和 M 之间，这里

2/M<=k<=1/2。常量 k 称作该树的最小填充因子，M 为一个结点可以容纳索引项的最大数目。索引项形式为（p，ptr），其中 p 是用作搜索码的谓词。在叶结点中，ptr 为指向数据库中某一元组的指针；而在非叶结点中，ptr 为指向其子树根结点的指针。谓词中可以包含自由变量，只要相应子树中叶结点标识的所有元组能实例化这些变量即可。

Gist 可以建立一种可扩展的索引结构，包括数据类型和谓词的扩展。如数据库本身并不支持颜色的比较，但用户可以通过建立颜色这种数据类型并定义颜色的比较函数达到这种效果。Gist 允许用户快速地为自己的数据类型建立索引方法，提高查询速度。

为了使用 Gist 创建新的索引策略，用户需要对自定义的策略实现 7 个必要的接口和一个可选的借口。这 7 个必要的接口是 same()，union()，consistent(),penalty(),picksplit(),compress(),decompress()。一个可选的接口是 distance()。

same(E，E)：判断两个索引项是否相同，如果相同，返回真，否则，返回假。

union(P)：对于给定的索引项，返回谓词 r，使得索引项组中各个索引项子树中所有的元组均满足 r。

consistent (E,q):对于给定的谓词 r，判断索引项 E（p1，ptr1）是否与谓词匹配，如果匹配，返回真，否则返回假。

penalty(E，E)：对于两个索引项 E1（p1，ptr1），E2（p2，ptr2），

当把 E2 插入到 E1 的子树时，返回一个与索引数据域相关的测度值。

PickSplit(P)：对于包含 M+1 个索引项（p，ptr）的集合 P 而言，将 P 划分为两个索引项的集合 P1 和 P2，每个集合至少包含 kM 个索引项。

Compress(E)：对于给定的索引项（p，ptr），返回（a，ptr），a 为 p 的压缩形式。

Decompress(E)：对于索引项（a，ptr），其中 a=Compress（p），返回（r，ptr），使得 p→r。

Distance(E1,E2):返回两个元素在树上的距离。

Gist 在内部的实现中已经定义好索引项的创建,删除和查找操作，这些操作都依据用户实现的接口。用户如果对这些方法已经定义好，数据库就可以调用这些方法来实现对数据库的操作。

鉴于 OLAP 系统的特性，应该谨慎并保守地使用索引。避免在频繁更新的列上使用索引；在高选择性的列上使用 B-tree 索引；在低选择性的列上使用 Bitmap 索引。

通常来说在传统数据库中使用索引可以有效的提高数据访问效率，特别是在 OLTP 系统中，往往只是需要从大表中获取几行或者部分的数据记录，这个情况下索引确实是特别有效的提高数据获取速度的方法。但是在 gp 中未必如此，因为首先数据都是均匀最大可能的均匀分布到 segs 的，这意味着每个 instances 只是扫描整体数据的一部分，而且如果使用了分区技术，需要扫描的数据可能会更少，最后不同于

OLTP 的是在大数据环境中往往获取的都是大部分的记录集，在这个情况下使用索引未必是有效的提高数据获取速度的方法，应该保守使用索引。正确的方式应该是在不创建任何索引的情况下测试下性能，而后再做出正确的决定。

5.2.5 其他相关技术及最新研究成果

基于 SSD 的存储介质和内存数据库的出现对传统的索引技术提出了新的挑战。SSD 有着高速的读效率但是低效的写效率，以及“out-of-place update”的特性。传统的 B-树索引和 hash 索引主要是针对机械硬盘，需要重新设计来适应 SSD 的这些特性，研究[3]针对传统的 Tree 做了优化来提高索引效率。而对于内存数据库来说，移除了辅助存储层，不存在 buff manger。研究证明了尽管 hash 索引在内存数据库中可以达到较好的性能，但是传统的 B-树在内存数据库中效率很低，因此在内存数据库中引进了一些较新的索引技术。

在内存数据库的领域中，有影响的索引有早期的 T 树索引、基于缓存敏感的 CSS/CSB＋树，Trie-tree 和 hash 等。

T-tree[4]主要解决索引在内存中开销较大的问题，与传统的 B-tree 相比，前者无论是大小还是算法上都精简得多。T-tree 是一棵特殊的平衡二叉树（AVL），它的每个节点的关键字是有序的。T-tree 除了较高的节点空间占有率，遍历一棵树的查找算法在复杂程度和执行时间上也占有优势。

CSS-tree[5]（Cache-SensitiveSearch Trees），可以提供比二

分查找更为迅速的查询速度同时占用空间也较小。CSS 在一个排序过的数组顶端维护一个目录结构，且该结构的节点大小与机器 cache-line 大小相同。构造一棵有两千五百万键值的数组的 CSS-tree，花费的时间不足一秒。

CSB+-Tree[6](CacheSensitive B+-Trees),是 B+Tree 的变体，连续存储给定节点的子节点，并且只存储第一个节点的地址，其他节点的地址通过偏移量进行计算。由于只存储一个节点，cache 的利用率很高。

另外，其他的一些索引结构如 radix tree 也有了新的改进。一种叫做 ARTful（adaptive radix tree for short）在研究[7]被提出。Radix tree 也被称为基数树或前缀树，与 trie tree 的区别在于 radix tree 压缩了单词，节点变得更少。而在 ART 中吸收了 Judy 的思想，将节点分为四种，可以提供节点上的水平压缩；同时也将不必要的节点合并达到在垂直方向上进行压缩的目的。

在 2017 年 Google 公布了一项最新工作，The Case For Learned Indexed Structure。该文章就思考，既然索引会根据数据的分布来给出对于查询数据的预测，那么是否可以使用当今最流行的机器学习、深度学习方法来模拟这个过程？这是一个非常大胆的想法，而这个想法也在他们的实验中得到验证，用深度神经网络对已有数据分布进行训练，学习到数据分布。在对于新的查询请求，该方法会给出请求数据的相应位置。该文章突破了传统意义上对于索引的优化问题，也是一个非常最要的突破口。相信在不久的未来，数据库的设计会着重考

虑到该方法，也会彻底改变数据库索引技术的设计。

5.3 索引实验部分

本小节仅仅介绍一个索引实例分析，读者可以根据自己需求参照实例进行自己的实验。

5.3.1 索引分析实例

Greenplum 中创建索引的命令如下：

```
testdb=# \h create index
Command:        CREATE INDEX
Description: define a new index
Syntax:
CREATE [UNIQUE] INDEX name ON table
        [USING btree|bitmap|gist]
        ( {column | (expression)} [opclass] [ ASC | DESC ] [ NULLS
{ FIRST | LAST } ] [, ...] )
        [ WITH ( FILLFACTOR = value ) ]
        [TABLESPACE tablespace]
        [WHERE predicate]
```

利用第三章建立的 orders 表作为测试数据，首先不建立索引进行顺序扫描查看消耗的时间：

```
testdb=# explain analyse select o_custkey from orders where o_orderkey=1;
                                          QUERY PLAN

------------------------------------------------------------------------------
----------------
 Gather Motion 1:1  (slice1; segments: 1)  (cost=0.00..25438.75 rows=1 width=4)
   Rows out:  1 rows at destination with 329 ms to end, start offset by 0.491 ms
.
   ->  Seq Scan on orders  (cost=0.00..25438.75 rows=1 width=4)
         Filter: o_orderkey = 1
         Rows out:  1 rows with 0.415 ms to first row, 322 ms to end, start offs
et by 7.481 ms.
 Slice statistics:
   (slice0)    Executor memory: 318K bytes.
   (slice1)    Executor memory: 163K bytes (seg0).
 Statement statistics:
   Memory used: 128000K bytes
 Total runtime: 330.538 ms
(11 rows)
```

建立 B-tree 索引之后：

```
testdb=# explain analyse select o_custkey from orders where o_orderkey=1;
                                       QUERY PLAN

------------------------------------------------------------------------------
---------
 Gather Motion 1:1  (slice1; segments: 1)  (cost=0.00..200.37 rows=1 width=4)
   Rows out:  1 rows at destination with 41 ms to end, start offset by 0.347 ms.
   ->  Index Scan using idx on orders  (cost=0.00..200.37 rows=1 width=4)
         Index Cond: o_orderkey = 1
         Rows out:  1 rows with 16 ms to first row, 17 ms to end, start offset b
y 25 ms.
 Slice statistics:
   (slice0)    Executor memory: 318K bytes.
   (slice1)    Executor memory: 145K bytes (seg0).
 Statement statistics:
   Memory used: 128000K bytes
 Total runtime: 42.489 ms
(11 rows)
```

可以看到建立索引之后性能提高了十倍左右。

在建立 bitmap 索引，这里不能在 o_custkey 建立，因为 o_custkey 是没有重复的，建立的代价很大，而且查询也几乎不会有提高。

```
testdb=# explain analyse select * from orders where o_orderstatus ='F';
                                                                  QUERY PLAN

------------------------------------------------------------------------------
------------------------------------------------------------
 Gather Motion 2:1  (slice1; segments: 2)  (cost=5458.46..21357.57 rows=739529 w
idth=107)
   Rows out:  729409 rows at destination with 4.594 ms to first row, 1447 ms to
end, start offset by 52 ms.
   ->  Bitmap Heap Scan on orders  (cost=5458.46..21357.57 rows=369765 width=107
)
         Recheck Cond: o_orderstatus = 'F'::bpchar
         Rows out:  Avg 364704.5 rows x 2 workers.  Max 364763 rows (seg1) with
2.512 ms to first row, 223 ms to end, start offset by 54 ms.
         ->  Bitmap Index Scan on idx_bm  (cost=0.00..5273.58 rows=369765 width=
0)
               Index Cond: o_orderstatus = 'F'::bpchar
               Bitmaps out:  Avg 1.0 x 2 workers.  Max 1 (seg0) with 3.218 ms to
 end, start offset by 53 ms.
 Slice statistics:
   (slice0)    Executor memory: 338K bytes.
   (slice1)    Executor memory: 658K bytes avg x 2 workers, 658K bytes max (seg0
).
 Statement statistics:
   Memory used: 128000K bytes
 Total runtime: 1551.075 ms
```

跟没有建立索引的查询进行时间比较：

```
testdb=# explain analyse select * from orders where o_orderstatus ='F';
                                                                    QUERY PLAN

------------------------------------------------------------------------------
------------------------------------------------------------
 Gather Motion 2:1  (slice1; segments: 2)  (cost=0.00..25438.75 rows=739529 widt
h=107)
   Rows out:  729409 rows at destination with 1.402 ms to first row, 1610 ms to
end, start offset by 0.305 ms.
   ->  Seq Scan on orders  (cost=0.00..25438.75 rows=369765 width=107)
         Filter: o_orderstatus = 'F'::bpchar
         Rows out:  Avg 364704.5 rows x 2 workers.  Max 364763 rows (seg1) with
0.162 ms to first row, 305 ms to end, start offset by 1.435 ms.
 Slice statistics:
   (slice0)    Executor memory: 330K bytes.
   (slice1)    Executor memory: 199K bytes avg x 2 workers, 199K bytes max (seg0
).
 Statement statistics:
   Memory used: 128000K bytes
 Total runtime: 1664.070 ms
(11 rows)
```

可以发现时间几乎是一样的，这样的索引建立几乎是没有意义的。

查询建立的 idx 和 idx_bm 在磁盘中占用的空间（单位是 Byte）：

```
testdb=# select pg_relation_size('idx');
 pg_relation_size
------------------
         33587200
(1 row)

testdb=# select pg_relation_size('idx_bm');
 pg_relation_size
------------------
          1114112
(1 row)
```

跟原始的基表占用的空间进行比较：

```
testdb=# select pg_relation_size('orders');
 pg_relation_size
------------------
        218071040
(1 row)
```

可以看到索引页占到了很大一部分空间。idx_bm 索引几乎没有提高性能而且占用了大量空间。所以在使用索引的时间我们一定要慎重。

5.3.2 实验要求

1、在 Greenplum 采用不同的索引方式验证两者的特性；判断何种情况那种索引的查询速度最大。

2、在 Greenplum 中比较不同索引在磁盘中所占的空间大小。

5.4 本章小结

本章主要介绍索引的一些基本知识，更重要的是读者需要明白索引技术的基本原理，明白常见的索引在数据库中的应用，最好可以自己动手在 Greenplum 中实践一下。在索引相关技术最后一小节中介绍了目前索引工作的最新动态，有心的读者可以思考一下下一步索引可以有哪些进一步的发展，有兴趣的读者也可以自己思考去解决索引目前还存在的瓶颈。

5.5 参考文献及进一步阅读材料

[1] Comer D. The Ubiquitous B-Tree.[J]. 1979, 11(11):121-137.

[2] Kraska T, Beutel A, Ed H. Chi, et al. The Case for Learned Index Structures. arXiv 2017.

[3] Jin P, Yang C, Jensen C S, et al. Read/write-optimized tree indexing for solid-state drives[J]. Vldb Journal, 2016:1-23.

[4]Lu H, Ng Y Y, Tian Z. T-tree or B-tree: main memory database index structure revisited[C]// Database Conference, 2000. Adc 2000. Proceedings. Australasian. IEEE, 2000:65-73.

[5] Rao J, Ross K A. Making B+- trees cache conscious in main memory[J]. Acm Sigmod Record, 2000, 29(2):475-486.

[6]Rao J, Ross K A. Cache Conscious Indexing for Decision-Support in Main Memory[C]// International Conference on Very Large Data Bases. Morgan Kaufmann Publishers Inc. 1999:78-89.

[7] Alvarez V, Richter S, Chen X, et al. A comparison of adaptive radix trees and hash tables[C]// ICDE. 2015:1227-1238.