**Hash Index**

现在我们为数据库实现一个基于磁盘的哈希表。这个哈希表负责快速搜索，而不必搜索数据库表中的每一条记录。利用哈希表可以实现对数据库表的快速搜索，而不必依次搜索数据库表中的每一条记录。

这里使用可扩展的哈希模式(extendible hashing scheme)来实现一个哈希表。

索引(表)(index)包含了一个directory page(目录页)，该目录页包含指向bucket page(桶页面)的指针。该表将通过之前实现的缓冲池访问页面。该表包含的目录页存储了table和bucket的所有元数据。这个哈希表需要支持对满/空bucket的拆分和合并，以及在全局深度必须更改时支持directory的扩展和收缩。

我们将实现如下功能：

* **页布局**
* **可扩展哈希的实现**
* **并发控制**

**一、页布局**

该哈希表应该通过DBMS的BufferPoolManager来访问。我们不应分配内存来存储(哈希表的)信息。(哈希表的)所有内容都必须存储在磁盘页中，以便能够通过DiskManager读取或者写入他们。如果创建一个哈希表，并将其页写入磁盘，然后重启DBMS，则应该能够在重启之后从磁盘加载回哈希表。

为了支持在页面顶部读写哈希表bucket，需要实现两个Page类来存储哈希表的数据，这需要理解如何从BufferPoolManager中以页面的形式分配内存。

**1. Hash Table Directory Page**

该类保存哈希表的所有元数据。它被划分为如下表所示的字段:

**变量名 大小 描述**

page\_id\_ 4 bytes 自身的pageid

lsn\_ 4 bytes 日志流水号（后面并发控制将用到）

global\_depth\_ 4 bytes 目录的全局深度

local\_depths\_ 512 bytes 每个bucket的局部深度的数组（uint8）

bucket\_page\_ids\_ 2048 bytes bucket的pageid的数组

bucket\_page\_ids\_数组将每个目录项对应的桶id映射为page\_id，bucket\_page\_ids\_中的第i个元素是第i个目录项的page\_id。

我们将在头文件src/include/storage/page/hash\_table\_directory\_page.h以及源文件src/storage/page/hash\_table\_directory\_page.cpp中实现Hash Table Directory Page

有一点需要注意的是:**这里的page\_id\_指的是就是该哈希目录页在缓冲池实现中的表示物理页的page\_id\_**！

**2. Hash Table Bucket Page**

Hash Table Bucket Page保存了三个数组：

* occupied\_ : 如果array\_的第i个索引曾经被占用过，则occupied\_的第i位为1。
* readable\_ : 如果array\_的第i个索引包含一个可读值，则readable\_的第i位为1。
* array\_ : 保存key-value对的数组。

Hash Table Bucket Page中可用的槽数取决于存储的Key和Value的类型，这里只需要支持定长的key和value即可。在单个hash table实例中，key和value的大小都是不变的，但是不同的实例中则有可能改变，例如，哈希表#1可以有32位键，而哈希表#2可以有64位键。

我们将在头文件src/include/storage/page/hash\_table\_bucket\_page.h以及源文件src/storage/page/hash\_table\_bucket\_page.cpp中实现Hash Table Bucket Page。

每个哈希表的Directory或Bucket页对应于从缓冲池中获取的内存页中的内容（即data\_数组）。每次尝试读写页面的时候，需要使用唯一的pageid从缓冲池中获取页(page)，然后将他们重新转换(reinterpret\_cast)为Directory或Bucket页，并且要在任何读写操作之后Unpin这个页面。

我们将完成以下函数：

Bucket Page: - Insert - Remove - IsOccupied - IsReadable - KeyAt - ValueAt

Directory Page: - GetGlobalDepth - IncrGlobalDepth - SetLocalDepth - SetBucketPageId - GetBucketPageId

注意：与该部分有关的测试文件在container文件夹中。

**补充说明：**

1) static\_cast

使用方法：static\_cast < type-id > ( expression )

功能：把expression转换为type-id类型，无运行时类型检查，转换存在不安全性。

用法：①用于类层次结构中基类和子类之间指针或引用的转换。进行上行转换（把子类的指针或引用转换成基类表示）是安全的；进行下行转换（把基类指针或引用转换成子类指针或引用）时，由于没有动态类型检查，所以是不安全的。

②用于基本数据类型之间的转换，如把int转换成char，把int转换成enum。这种转换的安全性也要开发人员来保证。

③把void指针转换成目标类型的指针(不安全)。

④把任何类型的表达式转换成void类型。

void的字面意思是“无类型”，void \*则为“无类型指针”，void \*可以指向任何类型的数据。

2)dynamic\_cast

使用方法：dynamic\_cast < type-id > ( expression )

说明：该运算符把expression转换成type-id类型的对象。type-id必须是类的指针、类的引用或者void \*；如果type-id是类指针类型，那么expression也必须是一个指针，如果type-id是一个引用，那么expression也必须是一个引用。

使用场景：dynamic\_cast主要用于类层次间的上行转换和下行转换，还可以用于类之间的交叉转换。 在类层次间进行上行转换时，dynamic \_cast和static\_cast的效果是一样的；在进行下行转换时，dynamic\_cast具有类型检查的功能，比static\_cast更安全。

注意事项：

①dynamic\_cast是动态转换，只有在基类指针转换为子类指针时才有意义。

②static\_cast 和 dynamic\_cast 可以执行指针到指针的转换，或实例本身到实例本身的转换，但不能在实例和指针之间转换。static\_cast只能提供编译时的类型安全，而dynamic\_cast可以提供运行时类型安全。

3)reinpreter\_cast

使用方法：reinpreter\_cast <type-id> (expression)

说明：type-id必须是一个指针、引用、算术类型、函数指针或者成员指针。它可以把一个指针转换成一个整数，也可以把一个整数转换成一个指针（先把一个指针转换成一个整数，再把该整数转换成原类型的指针，还可以得到原先的指针值）。

reinterpret\_cast 可以转换任意一个32bit整数，包括所有的指针和整数。可以把任何整数转成指针，也可以把任何指针转成整数，以及把指针转化为任意类型的指针，威力最为强大！但不能将非32bit的实例转成指针。总之，只要是32bit的东东，怎么转都行！

因为任何指针可以被转换到void\*，而void\*可以被向后转换到任何指针（对于static\_cast<> 和reinterpret\_cast<>转换都可以这样做），如果没有小心处理的话错误可能发生。

**二、实现哈希表**

我们将实现一个可扩展的哈希表，它支持插入(Insert)，点搜索(GetValue)和删除(Remove)。我们应严格按照API要求实现Insert、GetValue和Remove，此外，不能修改VerifyIntegrity函数。

我们实现的哈希表同时支持唯一键和非唯一键(允许key重复出现)。同一键不允许有重复的值。这意味着(key\_0, value\_0)和(key\_0, value\_1)可以存在于同一个哈希表中，但(key\_0, value\_0)和(key\_0, value\_0)不能存在于同一个哈希表中。如果Insert方法试图插入已有的键值对，则只返回false。

哈希表的实现隐藏了键值对的细节以及相关的比较器的信息，如下所示：

template <typename KeyType, typename ValueType, typename KeyComparator>

class ExtendibleHashTable {

// ---

};

和键值对以及比较器有关的类已经被实现：

* ‘KeyType’:哈希表中每个键的类型，键的实际大小由模板参数指定，并且取决于索引属性的数据类型。
* ‘ValueType’:哈希表中每个值的类型。
* ‘KeyComparator’:该类用于比较两个KeyType实例的大小，KeyComparator在和KeyType有关的文件中实现。`KeyComparator`类型的变量本质上是函数，例如，给定两个键`KeyType key1` 和`KeyType key2`，以及一个键比较器`KeyComparator cmp`，则可以通过 `cmp(key1, key2)` 比较键。

注意，ValueType实例的值可以使用简单的==来比较。

例如，测试文件中的语句HashTableBucketPage<int, int, IntComparator>中就指定了这三者的类型或相应函数。

**可扩展哈希表的细节**

此处我们实现Bucket的拆分/合并以及目录的增长/收缩。一些可扩展哈希的实现跳过了Bucket的合并，因为它可能在某些场景中导致抖动。不过我们这里实现了它。

1. Directory Indexing

当插入哈希索引时，我们希望使用最低有效位来索引目录。当然，正确使用最高有效位索引目录是可能的，但是使用最低有效位会使目录展开操作简单得多。

2.Splitting Buckets

如果没有插入的空间，就必须将Bucket一分为二。我们可以在桶刚满的时候就执行split。但是，参考的解决方案是只有当插入会造成页面溢出的时候才进行拆分，我们会使用这种方法。

3.Merging Buckets

当Bucket变空之后，必须要尝试合并他们。通过检查Bucket的占用及其split image，可以实现更积极地合并。但这些昂贵的检查和额外的合并会增加抖动。

为了保持相对简单，使用以下规则进行合并：

* 只合并空bucket
* 只有在buckets的split image具有相同的local depth时，才将bucket和他们的split image合并
* 只有local depth大于0的bucket才可以被合并

4.Directory Growing

这部分没什么规则，我们只能扩展深度或者不扩展。

5.Directory Shrinking

只有当每个bucket的local depth均严格小于global depth时，才执行目录收缩。我们可能看到其他关于目录收缩的测试，但这个测试并不重要，因为我们将local depth保存在目录页中。

**性能**

一个重要的有关性能的细节是只在需要时使用write lock和latches。经常使用write locks将会导致性能下降。

此外，一个潜在的优化是在bucket页面上考虑合适的扫描方式，这可以在某些情况下避免重复扫描。我们发现检查bucket页面的许多内容通常需要进行全面扫描，因此我们可以一次性收集所有这些信息。

**注意：**

Split image类似一个bucket的兄弟bucket，即他在装满之后分裂出来的那个bucket（也可以说两个bucket互为split image）。故在分裂合并的时候要寻找对应的split bucket，这就是前面GetSplitImageIndex函数的作用(该函数在hash\_table\_directory\_page.cpp中实现)。

实现细节：

1.注意是如何使用缓冲池得到哈希目录的，注意src\container\hash\extendible\_hash\_table.cpp文件中的**ret = reinterpret\_cast<HashTableDirectoryPage \*>(page->GetData());**一句

**三、并发控制**

到目前为止，我们可以假设哈希表只支持单线程执行。现在我们将修改实现，使其支持多个线程同时读/写表。

我们需要在每个bucket中使用latches，以便当一个线程写入bucket时，其他线程不会同时读取或修改该索引。另外，我们允许多个读取线程同时读取同一个bucket。

当需要分割或合并桶时，以及全局深度发生变化时，我们需要latch整个哈希表。

**注意**

在这个项目中有两个锁需要关注。第一个是extendible\_hash\_table.h中的table\_latch\_，它为可扩展哈希表上锁。它来自src/include/common/ RWLatch .h中的RWLatch类，它是由std::mutex实现的。

第二个是src/include/storage/page.h中的内置页面锁定功能。这是我们必须使用的用于保护桶页的锁。注意，要获取table\_latch\_上的读锁，需要从RWLatch.h中调用RLock，但是要获取bucket页上的读锁，必须通过reinterpret\_cast<Page \*>将bucket页重新解释为一个页指针，再从page.h中调用RLatch方法。

**附录：可扩展哈希详细介绍**

来自：<https://www.geeksforgeeks.org/extendible-hashing-dynamic-approach-to-dbms/>

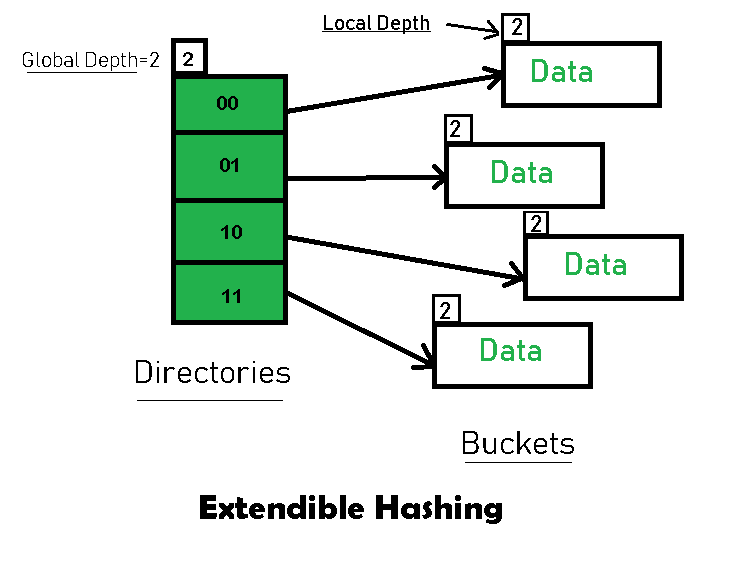
**可扩展哈希(Extendible Hashing)**是一种动态哈希方法，这是一种非常灵活的方法，其中哈希函数也经历动态变化。

**1.可扩展哈希的主要特征**:

**目录(Directories)**:目录在指针中存储桶的地址。每个目录都会被分配一个ID，ID可能会在每次目录扩展**(Directory Expansion)**时发生更改。

**桶(Buckets)**:桶用于对实际数据进行哈希处理。

**2.可扩展哈希的基本结构:**



**3.可扩展哈希中的常用术语:**

·**目录(Directories)**:目录存储指向桶的指针。每个目录都有一个唯一的ID，该ID可能在目录扩展时更改。哈希函数返回此目录ID，该目录ID用于导航到相应的桶。目录数= 2^全局深度。

·**桶(Buckets)**:存储键值对。目录指向桶。如果桶的局部深度小于全局深度，则有多个目录项指向该桶。(参考下面例子插入22的过程)

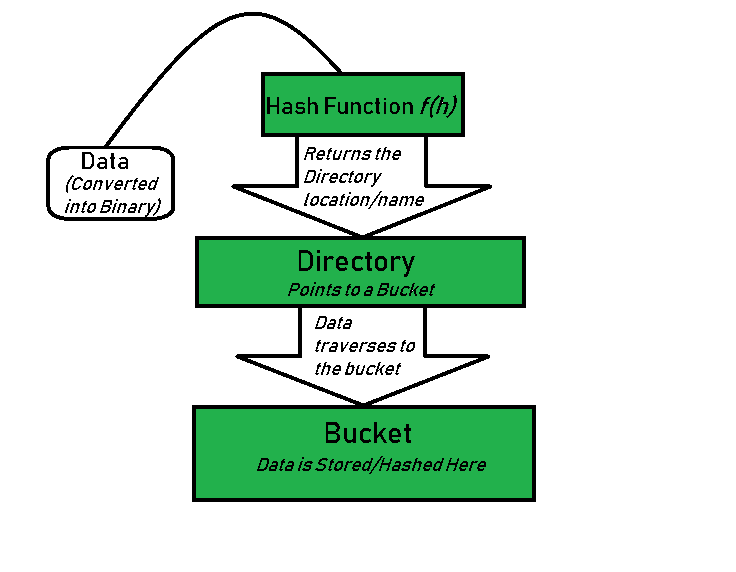
·**全局深度(Global Depth)**:它与目录相关联。它是哈希函数用来对键进行分类的数字的位数。全局深度=目录ID的位数。

·**局部深度(Local Depth)**: 与Global Depth类似，只是Local Depth属于桶，而不是目录。根据全局深度与局部深度，可确定何时溢出并采取相应行动。局部深度始终小于或等于全局深度。

·**桶拆分(Bucket Splitting)**:当桶中的元素数量超过特定大小时，该桶将分为两部分。

·**目录扩展(Directory Expansion)**:当桶溢出时，将进行目录扩展。当溢出桶的本地深度等于全局深度时，将执行目录扩展。

**4.可扩展哈希的工作流程**



**·步骤1-分析数据元素**:数据元素可能以各种形式存在，例如整数，字符串，浮点数等。当前，让我们考虑整数类型的数据元素。例如:49。

**·步骤2–转换为二进制格式**:将数据元素转换为二进制格式。对于字符串元素，可考虑带有起始字符(以区分各字符)的ASCII等效整数，然后将其转换为二进制形式。考虑整数类型的数据元素49，其二进制格式为110001。

**·步骤3-检查目录的全局深度**。假设哈希目录的全局深度为3。

**·步骤4―识别目录**:考虑二进制数中“全局深度”位的LSB(least-significant bits,最低有效位)，并将其与目录ID匹配。例如。得到的二进制文件:110001，全局深度为3。因此，哈希函数会返回110001的3位LSB:001。

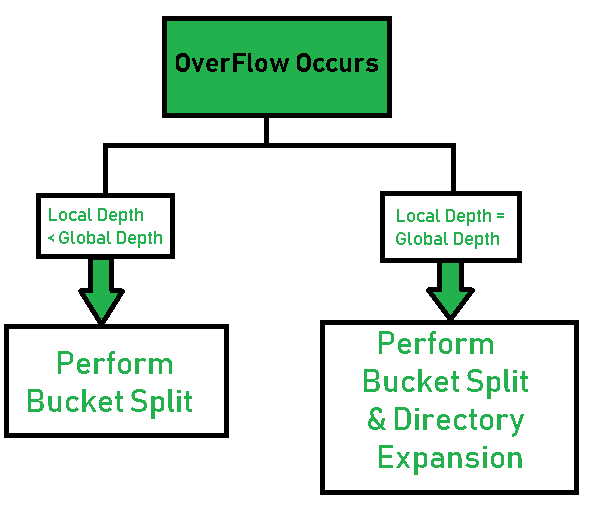
·**步骤5-导航**:现在，导航到目录ID为001的目录所指向的存储桶。

·**步骤6-插入和溢出检查**:插入元素并检查桶是否溢出。如果遇到溢出，执行步骤7，然后执行步骤8，否则，执行步骤9。

·**步骤7–在数据插入过程中解决溢出状况**:将数据插入桶内时，可能会发生桶溢出的情况。在这种情况下，我们需要遵循适当的程序以避免处理不当。首先，检查局部深度是小于还是等于全局深度。然后选择以下情况之一。

**·情况1**:如果溢出的桶的局部深度等于全局深度，则需要执行目录扩展(Directory Expansion)以及桶拆分(Bucket Split)。然后将全局深度和局部深度值加1。然后，分配适当的指针。目录扩展将使哈希结构中存在的目录数量增加一倍。

**·情况2**:如果局部深度小于整体深度，则仅发生桶拆分(Bucket Split)。然后仅将局部深度值加1。然后，分配适当的指针。



**·步骤8-重新哈希拆分后桶中的元素**

·**步骤9**-**该元素被成功散列(哈希)**。

**5.基于可扩展散列的示例**

现在，让我们考虑对以下元素进行哈希：

16,4,6,22,24,10,31,7,9,20,26。

**桶容量(Bucket Size)**:3

**哈希函数(Hash Function):**假设全局深度为X。则哈希函数返回X位的LSB(least-significant bits,最低有效位)。

**解答**:

·首先，计算每个给定数字的二进制形式：

16-10000

4-00100

6-00110

22-10110

24-11000

10-01010

31-11111

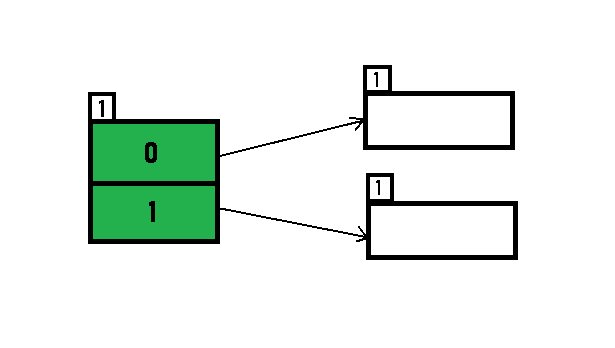
7-00111

9-01001

20-10100

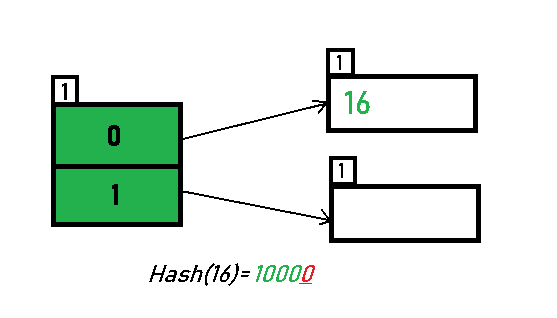
26-11010

·最初，global-depth和local-depth均为1。由于global-depth为1，二进制1位数能表示两个数字，因此目录有两项，指向两个空桶，则哈希帧如下所示:



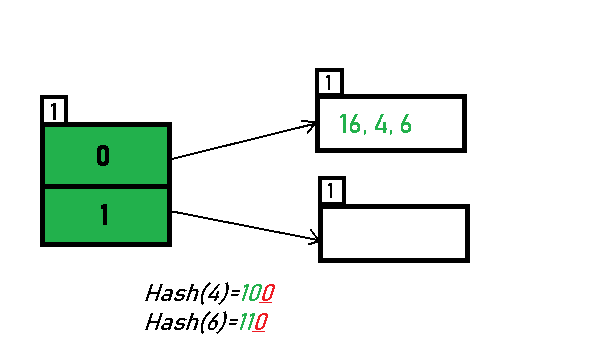
·插入16:

由于16的二进制格式是10000，全局深度是1。因此哈希函数返回10000的1位LSB，即O。因此，16映射到id=0的目录。



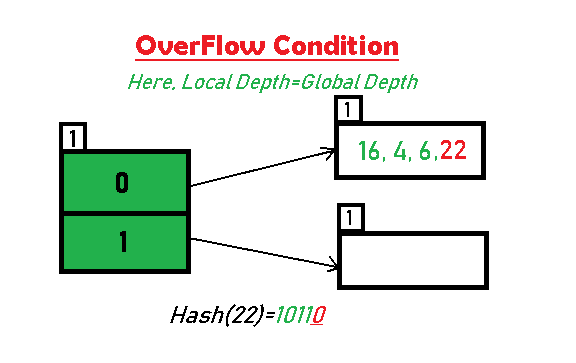
·插入4和6:

4和6的1位LSB都为0。因此，它们散列如下:

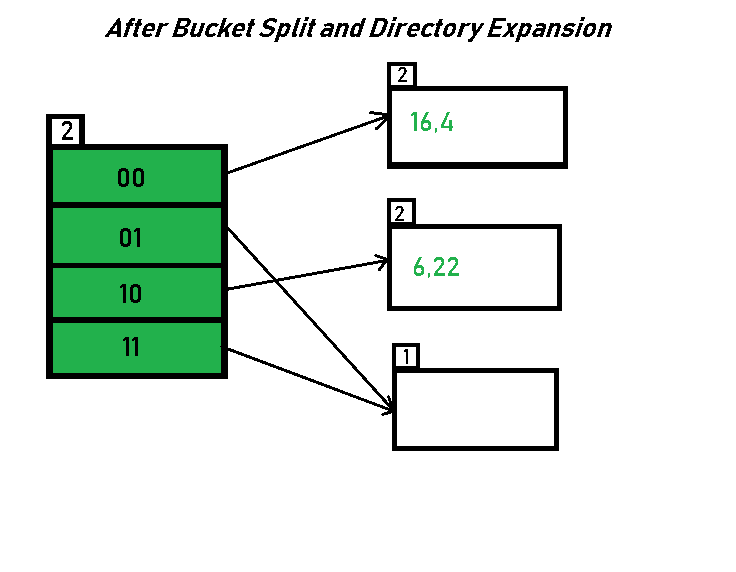


·插入22:

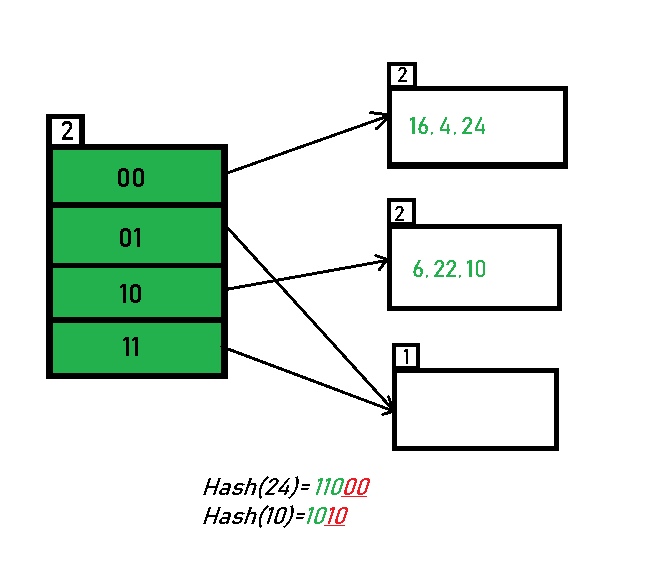
22的二进制形式是10110。它的1位LSB为0。目录0指向的存储桶已满。因此，发生溢出。



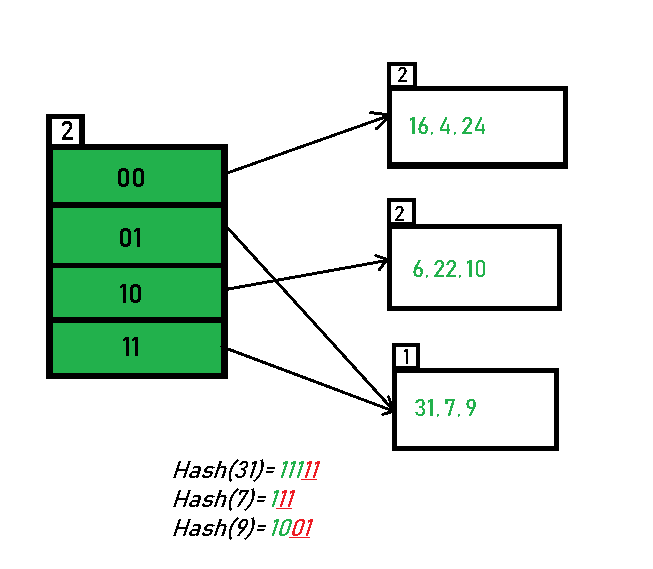
·按照步骤7-情况1的指示，由于“本地深度=全局深度”，桶将拆分并进行目录扩展。同样,在拆分之后，将对溢出的桶中存在的数字进行重新哈希处理。并且，由于全局深度增加1，因此现在全局深度为2。因此，会将数字16、4、6、22用2位LSB重哈希。[16(100**00**),4(1**00**),6 (1**10**),22(101**10**)]。另外注意，是原来目录项为0的桶分裂了，分裂成的两个桶分别对应目录项0**0**和1**0**，且局部深度均为2，而原来目录项为1对应的桶未分裂，因此新的目录项0**1**，1**1**都对应一个桶，且该桶局部深度不变仍为1。(桶的局部深度是它左上角的数字)



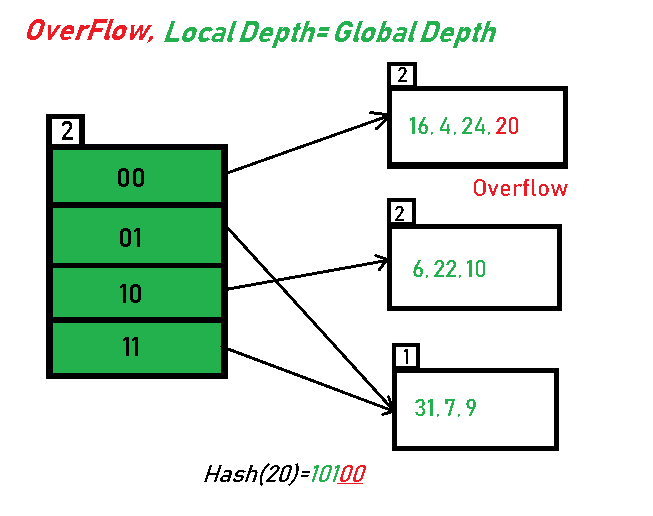
·插入24和10:对24(110**00**）和10 (10**10**）进行哈希处理。在这里，我们不会遇到溢出情况。



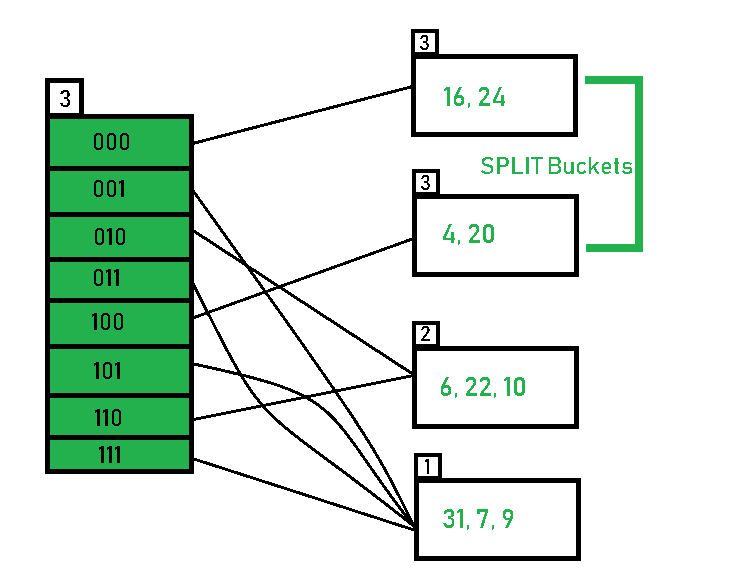
·插入31,7,9:所有这些元素[31(111**11**)，7 (1**11**），9 (10**01**）]的LSB均为01或11。因此，它们映射到01和11所指向的存储区上。在这里，我们不会遇到任何溢出情况。



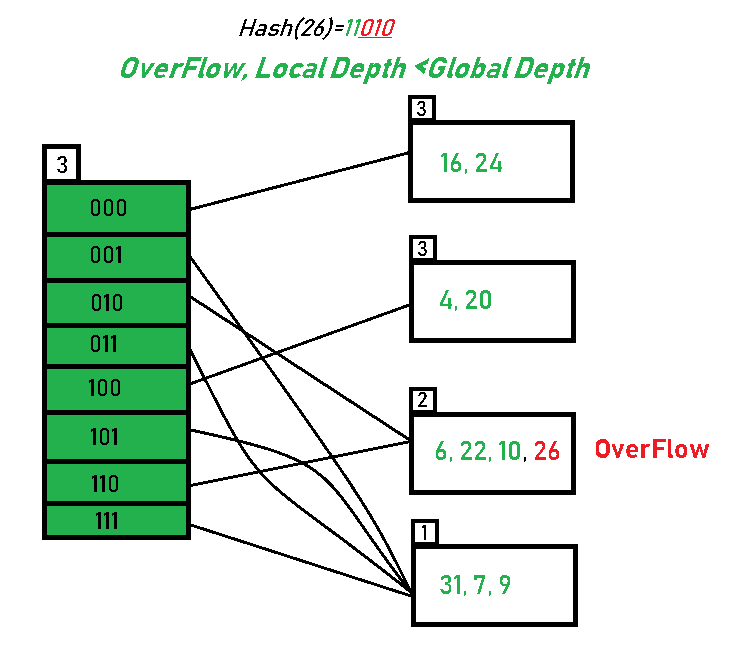
·插入20:插入数据元素20 (101**00**）将再次导致溢出问题。



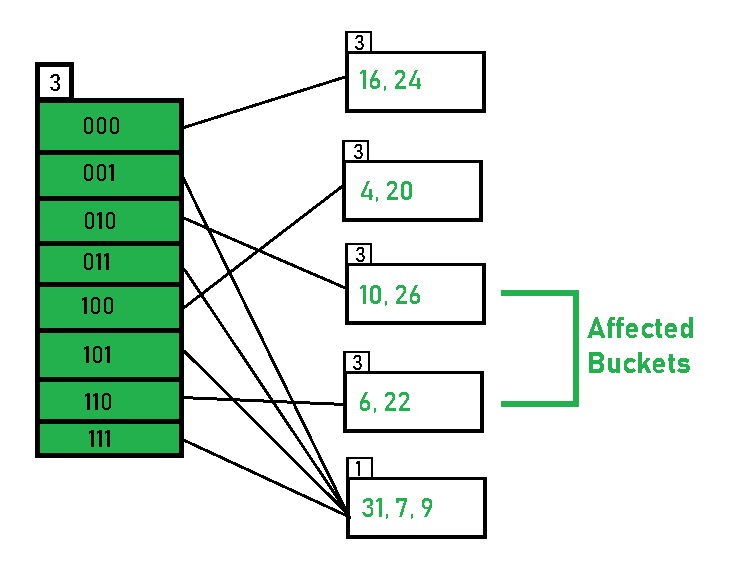
·如步骤7-情况1所示，由于桶的本地深度=全局深度，因此目录扩展(加倍)与存储桶拆分一起发生。溢出的桶中的元素将使用新的全局深度进行重新哈希。现在，新的哈希表如下所示:



·插入26:全局深度为3。因此，考虑了26 (11**010**）的3个LSB。因此，26适合目录010指出的桶。发生溢出。



·桶溢出，并且按照步骤7-情况2的指示，由于桶的本地深度<全局深度(2<3)，目录没有加倍，仅是桶被拆分，溢出桶中的元素被重新哈希。



**观察:**

1.如果桶的局部深度小于全局深度，则将有多个目录项指针指向该桶。

2.当桶中发生溢出时，该桶将分裂，其中的所有条目都将被重新哈希。

**可扩展哈希的好处:**

1.数据检索更便捷(就计算而言)。

2.由于存储容量是动态增加的，因此不会出现数据丢失的问题。

3.随着哈希函数的动态变化，与此关联的旧值(分裂桶中的值)将被新的哈希函数重新哈希。

**可扩展散列的局限性:**

1.如果多个记录哈希在同一目录项上，使得记录分布不均匀，则目录大小可能会大大增加。

2.每个桶的尺寸是固定的。

3.当全局深度和局部深度之差变得过大时，指针将浪费内存。

4.此方法的编码很复杂。