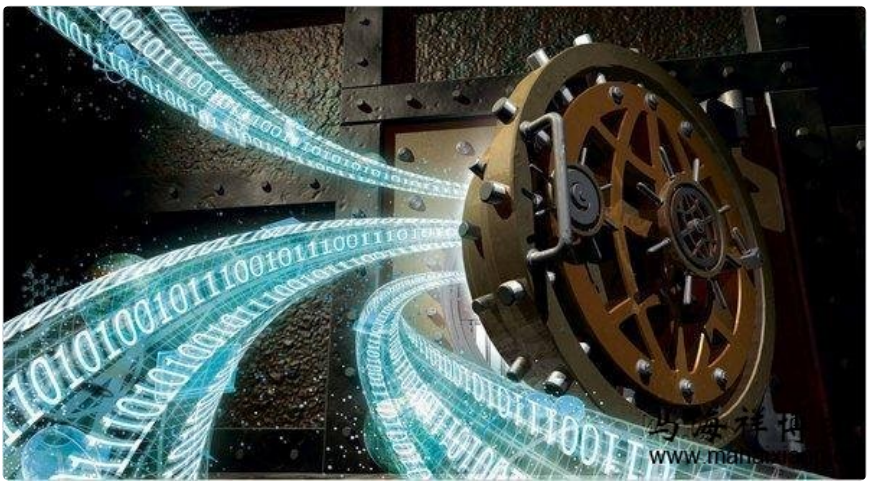
详解内存数据库中的索引技术

时间：2015-01-09 文章来源：马海祥博客 访问次数：3313 收藏到： 1

传统的数据库管理系统把所有数据都放在磁盘上进行管理，所以称作磁盘数据库（DRDB:Disk-Resident Database），磁盘数据库需要频繁地访问磁盘来进行数据的操作，磁盘的读写速度远远小于CPU处理数据的速度，所以磁盘数据库的瓶颈出现在磁盘读写上。

基于此，内存数据库的概念被提出来了，内存数据库(MMDB:Main Memory Database，也叫主存数据库)就是将数据全部或者大部分放在内存中进行操作的数据库管理系统，对查询处理、并发控制与恢复的算法和数据结构进行重新设计，以更有效地使用CPU周期和内存。

相对于磁盘，内存的数据读写速度要高出几个数量级，将数据保存在内存中相比从磁盘上访问能够极大地提高应用的性能。



近十几年来，内存的发展一直遵循摩尔定律，内存的价格一直下降，而内存的容量一直在增加，现在的主流服务器，几百GB或者几TB的内存都很常见，内存的发展使得内存数据库得以实现。

由于内存数据库与传统的磁盘数据库在设计和架构上都大不相同，所以传统的数据库索引不适用于内存数据库，研究者为改进内存数据库的索引结构做了相当多的研究跟工作，其中，影响较大的索引有早期的T树、基于缓存敏感(cacheconscious)的CSS/CSB+树，Trie-tree和Hash等等，本文中马海祥就这几种有代表性的索引算法进行研究和分析，为进一步改进内存数据库索引算法和提高索引性能打下坚实的基础。

一、T-tree

T-tree是针对主存访问优化的索引技术，T-tree是一种一个节点中包含多个索引条目的平衡二叉树，T-tree的索引项无论是从大小还是算法上都比B-tree精简得多。

1、T-tree的结构及特点

分类目录

SEO新闻	SEO思维
移动端SEO	SEO问答
医疗SEO	淘宝SEO
企业SEO	站外SEO
网站设计	交互设计
网站策划	网页制作
营销策划	营销案例
竞价技巧	数据分析
写作技巧	微信微博
自媒体	新媒体
内容营销	网站运营
O2O模式	App运营
产品运营	网赚教程
创新思维	电子商务
名人访谈	创业故事

热点推荐

基于眼球追踪技术对用户调研的探讨研究



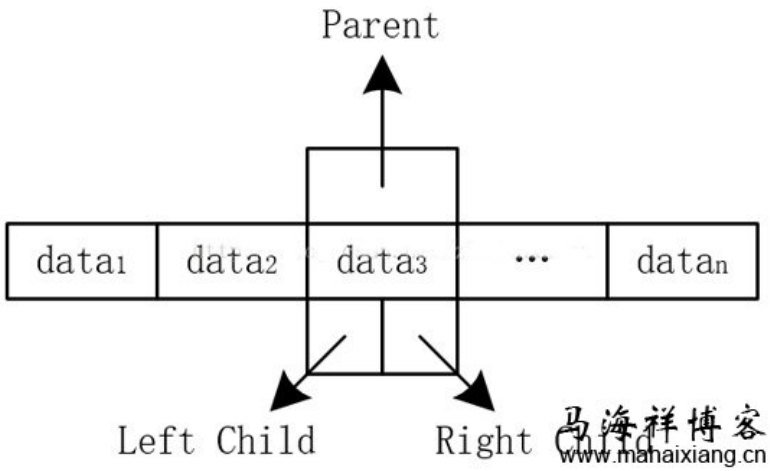

运营思维 更多>>

如何才能写出一篇优质文章?

立即访问

自媒体运营的规范准则

立即访问



T-Tree的结点

T-tree的搜索算法不分搜索的值在当前的节点还是在内存中的其他地方，每访问到一个新的索引节点，索引的范围减少一半。

T-tree索引用来实现关键字的范围查询，T-tree是一棵特殊平衡的二叉树（AVL），它的每个节点存储了按键值排序的一组关键字，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，现在T-tree已经成为内存数据库中最主要的一种索引方式。

T-tree具有以下特点：

（1）、左子树与右子树之差不超过1。

（2）、在一个存储节点可以保存多个键值，它的最左与最右键值分别为这个节点的最小与最大键值，它的左子树仅仅包含那些键值小于或等于最小键值的一记录，同理右子树只包括那些键值大于或等于最大键值的记录。

（3）、同时拥有左右子树的节点被称为内部节点，只拥有一个子树的节点被称为半叶节点，没有子树的节点被称为叶子。

（4）、为了保持空间的利用率，每一个内部节点都需要包含一个最小数目的键值。

由此可知，T-tree是一个每个结点含有多个关键字的平衡二叉树，每个节点内的关键字有序排列，左子树都要比根节点关键字小，右子树都要比根节点关键字大。

在上述T-tree结点结构中，马海祥觉得还包含如下信息：

（1）、balance(平衡因子)，其绝对值不大于1，balance=右子树高度-左子树高度；

（2）、Left_child_ptr和Right_child_ptr分别表示当前结点的左子树和右子树指针；

（3）、Max_Item表示结点中所能容纳的键值的最大数；

（4）、Key[0]至K[Max_Item-1]为结点内存放的关键字；

（5）、nItem是当前节点实际存储的关键字个数。

对于T-tree有如下特征：

（1）、与AVL树相似，T-tree中任何结点的左右子树的高度之差最大为1；

（2）、与AVL树不同，T-tree的结点中可存储多个键值，并且这些键值排列有序；

（3）、T-tree结点的左子树中容纳的键值不大于该结点中的最左键值；右子树中容纳的键值不小于该结点中的最右键值；

（4）、为了保证每个结点具有较高的空间占用率，每个内部结点所包含的键值数目必须不小于某个指定的值，通常为(Max_Item-2)(Max_Item为结点中最大键值目)。

2、T-tree索引的操作

用T-tree作为索引方式主要完成三个工作：查找，插入，删除，其中插入和删除都是以查找为基础，下面分别介绍三种操作的流程。

（1）、查找

T-tree的查找类似于二叉树，不同之处主要在于每一结点上的比较不是针对结点中的各个元素值，而是首先检查所要查找的目标键值是否包含在当前结点的最左键值和最右键值所确定的范围内，如果是的话，则在当前结点的键值列表中使用二分法进行查找。

如果目标键值小于当前结点的最左键值，则类似地搜索当前结点的左孩子结点；如果目标键值大于当前结点的最右键值，则类似地搜索当前结点的右孩子结点。

（2）、插入

T-tree的插入是以查找为基础，应用查找操作定位目标键值插入位置，并记下查找过程所遇到的最后结点。

如果查找成功，判断此结点中是否有足够的存储空间，如果有，则将目标键值插入结点中；否则将目标键值插入此结点，然后将结点中的最左键值插入到它的左子树中(此时是递归插入操作)，之后结束。

否则分配新结点，并插入目标键值，然后根据目标键值与结点的最大最小键值之间的关系，将新分配的结点链接为结点的左孩子或右孩子，对树进行检查，判断T-tree的平衡因子是否满足条件，如果平衡因子不满足则执行旋转操作。

（3）、删除

T-tree的删除操作也是以查找为基础，应用查找操作定位目标键值。

如果查找失败，则结束；否则令N为目标键值所在的结点，并从结点N中删除目标键值；删除节点后，如果结点N为空，则删除结点N，并对树的平衡因子进行检查，判断是否需要执行旋转操作；如果结点N中的键值数量少于最小值，则根据N的平衡因子决定从结点N的左子树中移出最大的键值或者右子树中移出最小值来填充。

3、T-tree索引实现关键技术

实现T-tree索引即要实现T-tree的查找，插入和删除，其中又以查找为基础，对T-tree的维护也就是T-tree的旋转为关键，当由于插入或删除键值导致树的失衡，则要进行T-tree的旋转，使之重新达到平衡。

在插入情况下，需要依次对所有沿着从新创建结点到根结点路径中的结点进行检查，直到出现如下两种情况之一时中止：某个被检查结点的两个子树高度相等，此时不需要执行旋转操作；某个被检查结点的两个子树的高度之差大于1，此时对该结点仅需执行一次旋转操作即可。

在删除情况下，类似地需要依次对所有沿着从待删除结点的父结点到根结点路径中的结点进行检查，在检查过程中当发现某个结点的左右子树高度之差越界时，需要执行一次

旋转操作，与插入操作不同的是，执行完旋转操作之后，检查过程不能中止，而是必须一直执行到检查完根结点。

由此可以看出，对于插入操作，最多只需要一次旋转操作即可使T-tree恢复到平衡状态；而对于删除操作则可能会引起向上的连锁反应，使高层结点发生旋转，因而可能需要进行多次旋转操作。

为了对T-tree进行平衡，需要进行旋转操作，旋转是T-tree中最关键也是最难的的操作，下面介绍T-tree旋转的技术。

旋转可分为四种情况：由左孩子的左子树的插入（或者删除）引起的旋转记为LL旋转，类似有LR，RR及RL旋转。插入时的情况与删除类似。

二、CSS/CSB+树

CSS-trees(Cache-SensitiveSearch Trees)可以提供比二分查找更为迅速的查询操作而又不需大量额外的空间，该技术在在一个以排好序的数组顶端存储一个目录结构，且该目录结构的节点大小与机器cache-line大小相匹配，将该目录结构存储在数组中而无需存储内部节点的指针，子节点可通过数组偏移量定位，这与B+-trees不同。

1、FULL CSS-Tree

构造一棵结点包含m个键值的查询树，树的深度是d，那么一直到d-1的深度这棵树是一棵完全(m+1)-查询树，而在d层叶子结点从左往右分布，一棵m=4的实例树图3-1所示，其中方块数就是结点数，且每个结点有四个键值。

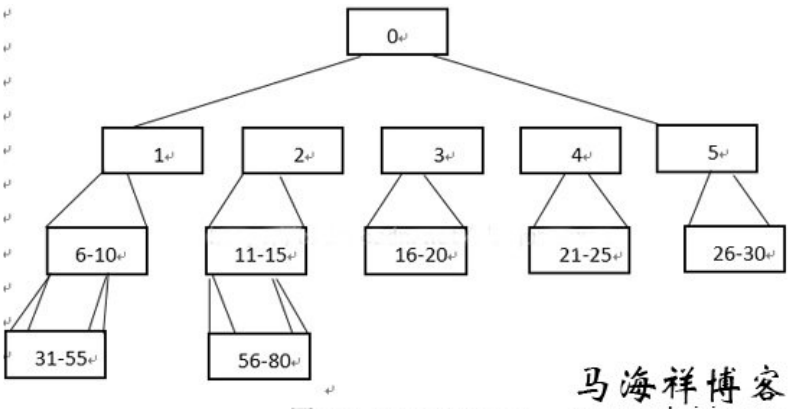


图 3-1 Full CSS-Tree

CSS-Tree的结点可以存储在数组中，如图3-2所示：

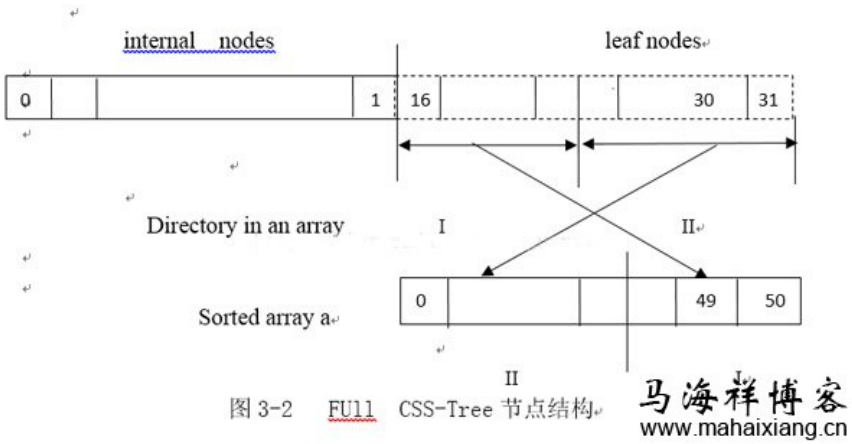


图 3-2 Full CSS-Tree 节点结构

（1）、构造FULL CSS-Tree

本月热点文章
1 如何收集和存储服务器运营的数据
2 关于大型网站架构的负载均衡技术详解
3 HTTPS建设使用的方案教程解析
4 自然语言处理的单词嵌入及表征方法
5 基于贝叶斯推断应用原理的过滤垃圾...
6 深入解析互联网协议的原理
7 HTTP、SSL/TLS和HTTPS协议的区...
8 详解内存数据库中的索引技术
9 基于眼球追踪技术对用户调研的探讨...
10 基于高斯模糊原理的模糊图片的研究

标签云
网页 友情链接 公众号 ded 淘宝 阿里巴巴 思维 模式 门户网 百度推广 qq pr 百度: 优化方法 互联网 团队管理 Java 转化率 文章写 SEM 技巧 seo技 wordpress 搜索 设计 SEO问题 新手 手机 SMO Java 社交媒体

将一个已排好序的数组构造一棵相应的Full CSS-Tree，首先将数组分为两部分，并且在叶子节点和数组元素间建立匹配，然后从最后一个内部节点开始，将节点直接左子树的最大键值作为节点入口。

对于某一些内部节点，也就是最深层最后一个叶子节点的祖先，可能完全键值，可以用数组前半部分最后的一个元素来填充这些键值，所以在某些内部节点会有一些复制的键值，尽管要增量更新一棵Full CSS-Tree树是很困难的，但构造这样一棵树花费并不大。

实验表明对于有着两千五百万键值的数组，构造其相应的Full CSS-Tree花费的时间不足一秒。

（2）、查询Full CSS-Tree

从根节点开始，每次都查询一个内部节点，利用二分查找来决定查找哪一个分支，重复上述行为直到叶子节点，最后将叶子节点与排好序的数组进行匹配。

在节点内所有的查询都由if-else构成，在内部节点进行二分查找时，一直比较左边的键值是否不小于要查询的键值，当找到第一个比要查询的键值小时，停止比较并进入右边的分支（如果找不到这样的值，就进入最左边的分支）。

这样可以保证当在节点中有复制的值时，我们可以在所有复制的键值中找到最左边的键值。

2、LevelCSS-Tree

对于每个节点有m个记录的Full CSS-Tree，有严格的m个键值，所有的记录都会被利用到，对于m=2t，我们定义每个节点只有只有m-1条记录，并且有一个分支因子m。

一棵Level CSS-Tree树比一棵相应的Full CSS-Tree树的深度大，因为分支因子是m而不是m+1，然后对于每一个节点，需要的同伴数更少。

若N为一个已排好序的数组元素所对应的节点数，Level CSS-Tree有logmN层，而Full CSS-Tree有logm+1N层，每个节点的同伴数是t，而Full CSS-tree是t*(1+2/(m+1))，所以Level CSS-tree的总的同伴数是logmN*t=log2N，而Full CSS-tree是logm+1N*t*(1+2/(m+1))=log2N*logm+1m*(1+2(m+1))。

因此，Level CSS-Tree所需的companion数比Full CSS-tree少，另一方面，Level CSS-Tree需要logmN个cache accesses，遍历logmN个节点，而Full CSS-Tree需logm+1N。

构建一棵Level CSS-Tree与Full CSS-Tree类似，我们也可以利用每个节点的空槽，来存储最后一个分支的最大值，来避免遍历整棵子树来获取最大元素值，查询一棵Level CSS-Tree也与查询Full CSS-Tree类似，唯一的不同就是子节点偏移量的计算。

3、CSB+-Tree

尽管CSS-Tree相比二分查找和T-Trees查询性能更好，但是它是用于决策支持的有着相对静态数据的工作负载设计的。

CSB+-Tree(CacheSensitive B+-Trees)是B+-Trees的变体，连续存储给定节点的子节点，并且只存储节点的第一个子节点的地址，其他子节点的地址可以通过相对这个子节点的偏移量计算获得，由于只存储一个子节点的指针，cache的利用率是很高的，与B+-Tree类似，CSB+-Tree支持增量更新。

CSB+-Tree有两种变体，分段CSB+-Tree(SegmentedCSB+-tree)和完全CSB+-tree(FullCSB+-Tree)分段CSB+-Tree将子节点分段，在同一段的子节点连续存储，在每个

节点中，只有每一段的起始地址才会被存储。

当有分裂时，分段CSB+-Tree可以减少复制开销，因为只有一个分段需要移动，完全CSB+-Tree为整个节点重新分配空间，因此减少了分裂开销。

（1）、CSB+-Tree上的操作

①、Bulkload.

对于CSB+-Tree树，一个有效的bulkload方法就是一层一层的建立索引结构，为每一个叶节点分配空间，计算在高层需要的节点数，并给该层分配连续的存储空间，通过将低层每一个节点的最大值填入高层的节点，并设置每一个高层节点的第一个子节点指针，重复上述操作直到高层只有一个节点，且这个节点为根节点，因为同一层的所有节点是连续的，所以构造节点组无需额外的复制操作。

②、Search

查询CSB+-Tree与查询B+-Tree类似，当最右边节点的键值K比要查询的键值小，给第一个子节点增加K的偏移量来获得子节点的地址。

例如，K是节点的第三个键值，可以用一个c语句找到子节点：child=first_child+3，其中child和first_child是节点的指针。

③、Insertion

对CSB+-Tree的插入操作也与B+-Tree类似，首先要查找键值的插入口，一旦定位至相应叶节点，判断该叶节点是否有足够的空间，如果有，就简单的将键值放置在该叶节点中，否则，需要分裂该叶节点。

当需要分裂叶节点时，基于父节点是否有足够的空间存放键值会产生两种情况，假设父节点p有足够的空间，令f为p的第一个子节点的指针，g为f指向的节点组，构建一个新的比g多了一个节点的节点组g’，将g中所有的节点复制到g’，g中要分裂的节点在g’中变为两个节点，更新p中第一个子节点的指针f，使它指向g’，并且重新分配g。

当父节点没有额外的空间并且自身需要分裂时，问题显得更为复杂，令f为p中第一个节点的指针，需要构建新的节点组g’，将g中的节点均分至g’和g中，p中一半的键值转移至g’中，为了将p分裂为p和p’，包含p的节点组需要像第一种情况一样进行复制，或者，如果节点组也是满的，我们需要递归的分裂p的父节点，父节点再重复上述操作。

④、Deletion

删除操作类似于插入操作，一般的，简单的定位数据入口并且加以删除。无需调整树保证50%的occupancy。

（2）、Segmented CSB+-Tree

考虑128字节的cache-line，CSB+-Tree中每个节点最多有30个键值，意味着每个节点可以有31个子节点，那么一个节点组最大可达31*128近4KB，因此每一个分裂，需要复制4KB的数据来创建一个节点组，若cache-line更大，分裂一个节点的开销将会更大。

修改节点结构可以减少分裂时的复制操作。可以将子节点分段，将每一段的地址存储在节点中，每一段形成了一个节点组，只有在同一段的子节点被连续存储。

第一种考虑是固定每一个分段的大小，填充第一个分段的节点，一旦第一个分段满了，就将节点放在第二个分段，若一个节点落在第二个分段，我们只需将第二个分段的节点复制到新的段中，而无需管第一个分段，若新的节点落在第一个分段(已经满了)，我们

需要将数据从第一个分段移至第二个分段，在上述例子中，针对随机插入，分裂产生的数据复制将会减少至1/2(1/2+3/4)*4KB=2.5KB。

另一种就是允许每个分段的大小不同，最终将节点分为两段，当有节点插入时，为这个节点所属的分段创建一个新的分段，并更新相应分段的大小，在这种方法中，严格来说每次插入只涉及到一个分段(但当父节点也需要分裂，此时两个分段都要复制)，若一个新的节点等可能的落入其中一个分段，一个分裂产生的数据复制量为1/2*4KB=2KB，这种方法可以进一步的减少数据复制量。

有两个分段的SegmentedCSB+Tree如图3-3所示(每个叶节点只有两个键值)：

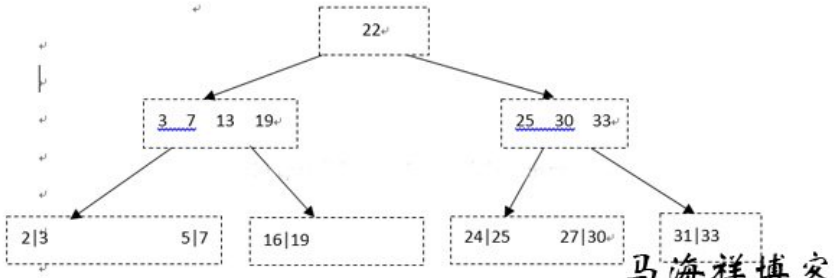


图 3-3 CSB+-tree of Order 2 with 2 Segments

分段CSB+-Tree可支持所有对树的操作，方法与非分段CSB+-Tree类似，然而，查找每个节点的右孩子比起非分段的CSB+-Tree的开销大，因为需要找到孩子所在的分段。

（3）、FULLCSB+-Tree

在FULLCSB+-Tree中，节点分裂的开销比CSB+-Tree小，在CSB+-Tree中，当节点分裂时，需要将节点组整个复制到新的组中，而在FullCSB+-Tree中，只需访问节点组的一半。

对于这种转移操作的源地址和目的地址有大的交叉，访问的cache-line的数目限制在s内，FULLCSB+-Tree在分裂上的平均时间开销是0.5s，而CSB+-Tree需时2s。

4、时间空间分析

假定键值、子节点指针、元组ID有着相同的空间大小K，n为叶节点数，c为cache-line的字节数，t为分段CSB+-Tree的分段数，每个节点的槽值为m，其中m=c/K，假定节点大小与cache-line相同，各个参数及其相应的值如图3-4所示：

Parameter	Typical Value
K	4 bytes
n	10^7
c	64 bytes
t	2
m=c/K	16

图 3-4 参数与相应值

Method	Branching Factor	Total Key Comparisons	Cache Misses	Extra Comparisons per Node
Full CSS-Trees	$m+1$	$\log_2 n$	$\frac{\log_2 n}{\log_2 (m+1)}$	0
Level CSS-Trees	m	$\log_2 n$	$\frac{\log_2 n}{\log_2 m}$	0
B^+-Trees	$\frac{m}{2}$	$\log_2 n$	$\frac{\log_2 n}{\log_2 m-1}$	0
CSB^+-Trees	$m-1$	$\log_2 n$	$\frac{\log_2 n}{\log_2 (m-1)}$	0
CSB^+-Trees (t segments)	$m-2t+1$	$\log_2 n$	$\frac{\log_2 n}{\log_2 (m-2t+1)}$	$t-1$
Full CSB^+-Trees	$m-1$	$\log_2 n$	$\frac{\log_2 n}{\log_2 (m-1)}$	$t-1$

图 3-5 CSB+-Tree查询时间分析

图3-5显示了各种方法间分支因子、键值差异数、cache未命中数、每个节点其他差异的比较，B+-Tree的分支因子比CSS-Tree小，而CSB+-Tree存储的子节点指针少，所需的分支因子与CSS-Tree相近，这导致每个方法的cache未命中次数不一样，节点的分支因子越大，cache未命中次数相应的越小。

在CSB+-Tree每增加一个分段，分支因子就会减少2，这是由于需要一个槽来存储子节点指针，另一个槽来存储新增分段的大小。

一般而言，B+-Tree中节点的70%空间是满的，需要相应的调整分支因子大小。

Method	Accessed Cache Lines in a Split	Typical Values (cache lines)
B^+-Trees	2	2
CSB^+-Trees	$(m-1)*2$	30
CSB^+-Trees (t segments)	$\frac{(m-2t+1)*2}{t}$	13
Full CSB^+-Trees	$\frac{m-1}{2}$	7.5

图 3-6 分裂时间分析

图3-6显示了在分裂时预期要访问的cache-line数，由于复制时源地址和目的地址有交叉，所以FullCSB+-Tree所需的数目小，分裂开销是插入操作总开销的一部分，另一部分是定位优叶子节点产生的查询开销。分裂开销相对独立于树的深度，这是由于大多数的分裂都发生在叶节点。

然而，当树的规模越来越大时，相应的查询产生的开销也会增大，CSB+-Tree的分裂开销比B+-Tree大，但是插入产生的总开销还与树的规模有关。

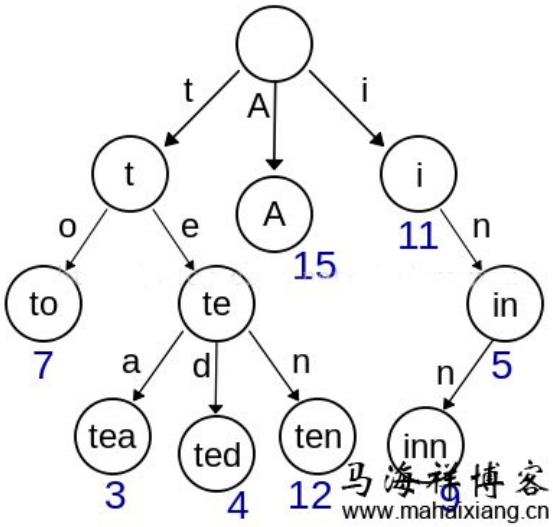
Method	Internal Node Space	Typical Value	Leaf Node Space	Typical Value
B^+-Trees	$\frac{4nc}{0.7(m-2)(0.7m-2)}$	28.4 MB	$\frac{2nc}{0.7(m-2)}$	130.6 MB
CSB^+-Trees	$\frac{2nc}{0.7(m-2)(0.7m-1)}$	12.8 MB	$\frac{2nc}{0.7(m-2)}$	130.6 MB
CSB^+-Trees (t segments)	$\frac{2nc}{0.7(m-2)(0.7(m-2t)-0.3)}$	16.1 MB	$\frac{2nc}{0.7(m-2)}$	130.6 MB
Full CSB^+-Trees	$\frac{2nc}{(0.7)^2(m-2)(0.7m-1)}$	18.3 MB	$\frac{2nc}{(0.7m-2)}$	186.6 MB

图 3-7 CSB+-Tree 空间分析

图3-7显示了不同算法的空间需求，假定所有节点70%的空间是满的，且分别计算内部节点和叶节点的空间大小，假定每个叶节点有2个兄弟节点指针，内部节点空间大小等于叶节点空间乘以1/(q-1)(q为分支因子)，这里不比较CSS-Tree，因为CSS-Tree不可能部分满。

三、Trie-tree索引

Trie-Tree又称单词查找树或键树，是一种树形结构，是一种哈希树的变种，典型应用是用于统计和排序大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计，它的优点是：最大限度地减少无谓的字符串比较，查询效率比哈希表高。



图展示了一个基本的tire-tree结构

1、Trie-tree性质

一般来说，Trie-tree有三个基本的性质：

（1）、根节点不包含字符，除根节点以外每一个节点都只包含一个字符。

（2）、从根节点到某一节点，路径上经过的的字符连接起来，为改节点对应的字符串。

（3）、每个节点的所有子节点包含的字符都不相同。

2、Trie树的基本实现

字母树的插入、删除和查找都非常简单，用一个一重循环即可，即第i次循环找到前i个字母所对应的子树，然后进行相应的操作，实现这棵字母树，我们用最常见的数组保存（静态开辟内存）即可，当然也可以开动态的指针类型（动态开辟内存）。

至于结点对儿子的指向，一般有三种方法：

（1）、对每个结点开一个字母集大小的数组，对应的下标是儿子所表示的字母，内容则是这个儿子对应在大数组上的位置，即标号。

（2）、对每个结点挂一个链表，按一定顺序记录每个儿子是谁。

（3）、使用左儿子右兄弟表示法记录这棵树。

在马海祥看来，这三种方法，各有特点，第一种易实现，但实际的空间要求较大；第二种，较易实现，空间要求相对较小，但比较费时；第三种，空间要求最小，但相对费时且不易写。

3、实现方法

搜索字典项目的方法为：

（1）、从根结点开始一次搜索。

（2）、取得要查找关键词的第一个字母，并根据该字母选择对应的子树并转到该子树继续进行检索。

（3）、在相应的子树上，取得要查找关键词的第二个字母,并进一步选择对应的子树进行检索。

（4）、迭代过程……。

（5）、在某个结点处，关键词的所有字母已被取出，则读取附在该结点上的信息，即完成查找。

其他操作类似处理

（1）、Trie原理

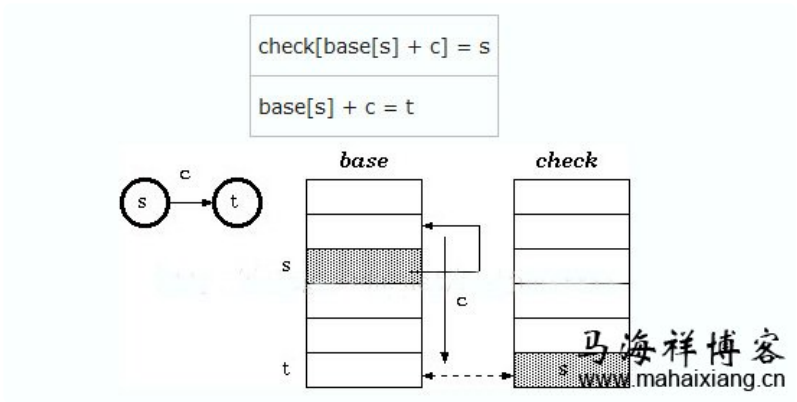
Trie的核心思想是空间换时间。利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。

（2）、Trie树的高级实现Double-Array实现

可以采用双数组（Double-Array）实现，如图1.3，利用双数组可以大大减小内存使用量，具体实现方法是：

两个数组，一个是base[]，另一个是check[]，设数组下标为i，如果base[i], check[i]均为0，表示该位置为空，如果base[i]为负值，表示该状态为终止态（即词语），check[i]表示该状态的前一状态。

定义1、对于输入字符c，从状态s转移到状态t，双数组字典树满足如下条件：



从定义1中，我们能得到查找算法，对于给定的状态s和输入字符c：

```
t := base[s] + c;
    if check[t] = s then
       next state := t
    else
       fail
    endif
```

我们知道双数组的实现方法是当状态有新转移时才分配空间给新状态，或可以表述为只分配需要转移的状态的空间，当遇到无法满足上述条件时再进行调整，使得其base值满足上述条件，这种调整只影响当前节点下一层节点的重分配，因为所有节点的地址分配是靠base数组指定的起始下标所决定的。

插入的操作，假设以某字符开头的base值为i，第二个字符的字符序列码依次为c1, c2, c3…cn，则肯定要满足base[i+c1], check[i+c1], base[i+c2], check[i+c2], base[i+c3], check[i+c3]…base[i+cn],check[i+cn]均为0。

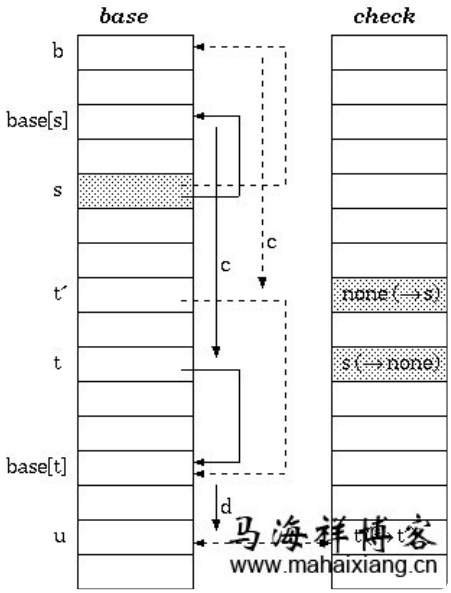


图4-3 Double Array 实现

假设，Tire里有n个节点，字符集大小为m，则DATrie的空间大小是n+cm，c是依赖于Trie稀疏程度的一个系数，而多路查找树的空间大小是nm。

注意，这里的复杂度都是按离线算法（offline algorithm）计算的，即处理时已经得到整个词库，在线算法（online algorithm）的空间复杂度还和单词出现的顺序有关，越有序的单词顺序空间占用越小。

查找算法的复杂度和被查找的字符串长度相关的，这个复杂度和多路查找树是一样的。

插入算法中，如果出现重分配的情况，我们要附加上扫描子节点的时间复杂度，还有新base值确定的算法复杂度，假如这儿我们都是用暴力算法（for循环扫描），那插入算法时间复杂度是O(nm + cm2)。

实际编码过程中，DATrie代码难度大过多路查找树，主要是状态的表示不如树结构那样的清晰，下标很容易搞混掉。

有个地方需要注意的是，base值正数表示起始偏移量，负数表示该状态为终止态，所以在查找新base值时，要保证查到的值是正数。

如：空Trie状态下，插入d时，因为第一个空地址是1，所以得到base=1-4=-3，这样base正负的含义就被破坏了。

4、Trie树的应用

Trie是一种非常简单高效的数据结构，但有大量的应用实例。

（1）、字符串检索

事先将已知的一些字符串（字典）的有关信息保存到trie树里，查找另外一些未知字符串是否出现过或者出现频率，例如：

①、给出N个单词组成的熟词表，以及一篇全用小写英文书写的文章，请你按最早出现的顺序写出所有不在熟词表中的生词。

②、给出一个词典，其中的单词为不良单词，单词均为小写字母。再给出一段文本，文本的每一行也由小写字母构成，判断文本中是否含有任何不良单词，例如，若rob是不良单词，那么文本problem含有不良单词。

（2）、字符串最长公共前缀

Trie树利用多个字符串的公共前缀来节省存储空间，反之，当我们把大量字符串存储到一棵trie树上时，我们可以快速得到某些字符串的公共前缀。

例如：给出N个小写英文字母串，以及Q个询问，即询问某两个串的最长公共前缀的长度是多少?

解决方案：首先对所有的串建立其对应的字母树，此时发现，对于两个串的最长公共前缀的长度即它们所在结点的公共祖先个数，于是，问题就转化为了离线（Offline）的最近公共祖先（LeastCommon Ancestor，简称LCA）问题。

而最近公共祖先问题同样是一个经典问题，可以用下面几种方法：

①、利用并查集（Disjoint Set），可以采用采用经典的Tarjan算法；

②、求出字母树的欧拉序列（Euler Sequence）后，就可以转为经典的最小值查询（Range Minimum Query，简称RMQ）问题

（3）、排序

Trie树是一棵多叉树，只要先序遍历整棵树，输出相应的字符串便是按字典序排序的结果。

例如：给你N个互不相同的仅由一个单词构成的英文名，让你将它们按字典序从小到大排序输出。

（4）、作为其他数据结构和算法的辅助结构

如后缀树，AC自动机等。

5、Trie树复杂度分析

（1）、插入、查找的时间复杂度均为O(N)，其中N为字符串长度。

（2）、空间复杂度是26^n级别的，非常庞大（可采用双数组实现改善）。

Trie树是一种非常重要的数据结构，它在信息检索，字符串匹配等领域有广泛的应用，同时，它也是很多算法和复杂数据结构的基础，如后缀树，AC自动机等。

四、TrieMemory

Trie Memory是一种在内存中存储和检索信息的方式，这种方式的优点是访问速度快，具有冗余存储信息的优点，主要的缺点是存储空间利用率很低。

1、基本的Trie Memory模型

假设我们需要跟踪一系列的单词集合，这些集合是字母组成的序列，这些单词序列有各种各样的长度，我们必须记住的是这些字母组成的有限序列在这个集合中，总得来说，我们需要判断一个序列是不是这个集合的成员。

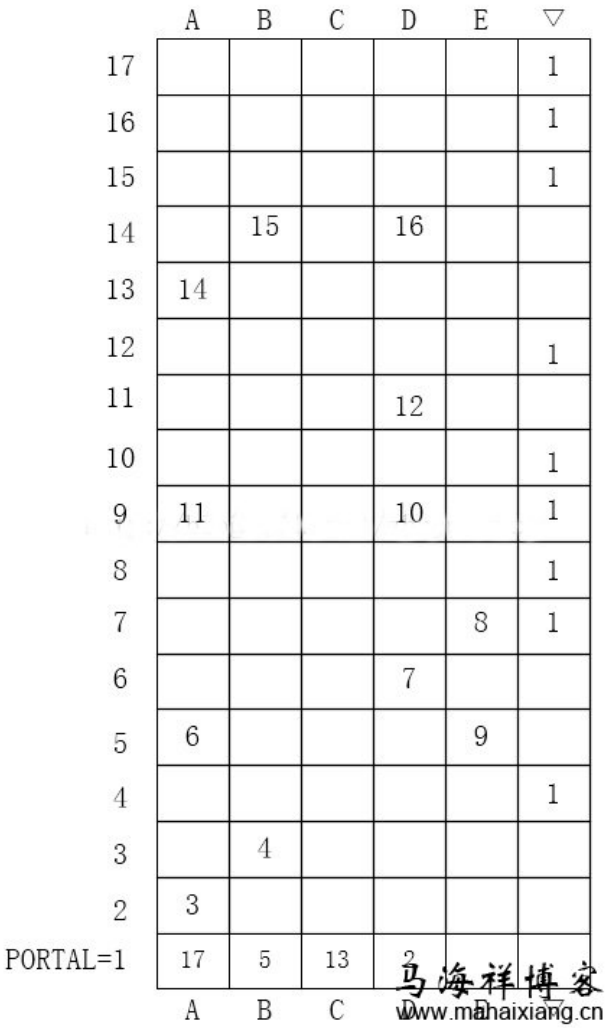
刚开始trie仅仅是register组成的一个集合，除此之外还有两个register，一个是α另一个是δ，每一个register都有cell来存储整个字母表，如果我们要存储“space”的话，每个register必须拥有27个cell。

每一个cell都有空间来存储其它register在内存中的地址，trie中的cell还没有用来存储信息，通常包含的是register α的地址信息。一个cell如果包含了非register α的register地

址，则它表示存储了信息，这些信息代表了这个cell的名称，“A”表示A cell,“B”表示B cell。下一个register的地址在序列中。

下面用一个例子来说明，为了让例子简单些，我们使用字母表的前5个字符来表示整体。然后用?表示“space”，假设我们想存储DAB,BAD,BADE,BE,BED,BEAD,CAB,CAD和A，接下来用图来说明整个流程。

在图中每一行代表一个register，每个register有6个cell，最后一行代表第三个特殊的register叫做portal register，是我们进入系统内存的通道它除了是入口外，也和其它register是一样的，其它register是编号的，register α将会选择它们，刚开始的时候register α是register 2。



基本Tire Memory模型

为了存储DAB，我们引入地址“2”进入portal register的D 单元格，然后我们移动到register 2然后引入地址“3”到A单元格，然后我们进入到register 3后把地址“4”放入单元格B，最后我们移动到register 4 并且把地址“1”放入?单元，它是终止参数，至此DAB存储结束。

然后我们转到第二个单词BAD，引入地址“5”进入portal register的B单元格来表示字母B，然后到register 5的A单元格写入地址“6”，再到register 6的D单元格写入地址“7”，最后到register 7的?单元格写入地址“1”。

当我们开始存储BADE时，我们发现B,A,D已经在trie中了，因此我们沿着已经存在的BAD的路径到register 7然后引入地址“8”到单元格E中去，然后把地址“1”放入register 8的?单元。

2、Register的类型

在刚才提到的结构中我们可以把register分为4种类型：

（1）、α(address) register来指向下一个存储信息的地址。

（2）、δ(deletion) register

（3）、v(next) register，下一步将要存储的信息(在空内存中，它是portal register)。

（4）、χ(exterior)类型χ是所有register中还没有接受存储信息并且没有被指向为下一个存储位置的register。

（5）、o(occupied)类型o是存有信息的register。

3、Trie的读和写

在上述的所有的register中除了χ都在trie中，存储和读取操作现在能够被简单的公平的定义如下。

（1）、写操作

①、把第i个参数字符传入下一个register，如果是第一个字符，则是portal register。

②、选择对应字符串的的cell,如果第i个参数字符是字母表的第j个字符，选择第j个cell。

③、检测来自第i个单元的联结。

④、如果这种联结使得register α：

（a）、通过αregister把联结投射到链接的头部，这样就可以存储信息。

（b)、投射从αregister到链接头部的联结来创建一个“next”register(v)。

（c)、最后，把所有的从v发出来的联结指向αregister。

⑤、如果源于第j个cell的联结指向非 αregister的话，移动到那个register去：

（a）、如果是第一个register，这参数是一个存储集合的成员(结束流程)。

（b）、如果不是register 1的话，i加1并且转到第二步去。

（2）、读操作

使用相同的流程，但是不要使用投射，不要投射任何关系，如果联结指向register 1，则这个参数是存储集合的一个成员，如果任何点的联结指向αregister，换句话说这个参数不是存储集合的成员。

五、HASH索引

HASH就是把关键词直接映射为存储地址，达到快速寻址的目的，即Addr=H(key)，其中key为关键词；H为哈希函数，主要有以下几种常用的哈希函数：

①、除留余数法(DivisionMethod)，H(key)=keyMOD p，p一般为质数；

②、随机数法(RandomMethod)，H(key)=random(key)，random为随机函数；

③、平方取中法(MidsquareMethod)。

HASH索引结构不需要额外的存储空间，并且能够在O(1)的时间复杂度下准确定位到所查找的数据，将磁盘数据库中的数据查找时间代价优化至最小，Hash索引结构由于以上优点在磁盘数据库中广泛的运用。

经历长久的研究，先后发展出了链接桶哈希(chainedbucket hash)，可扩展哈希(extendible hash)、线性哈希(linearhash)和修正的线性哈希(modified linear hash)。

但是这些哈希算法虽然针对内存数据库进行了少许优化，但是与传统数据库中所用的哈希算法没有明显不同，到了2007年，KennethA. Ross提出了基于现代处理器的Hash预取算法将SIMD指令集融入到Hash算法中，才真正从内存索引的角度改进了哈希算法，提升数据组织的效率。

1、链接桶哈希

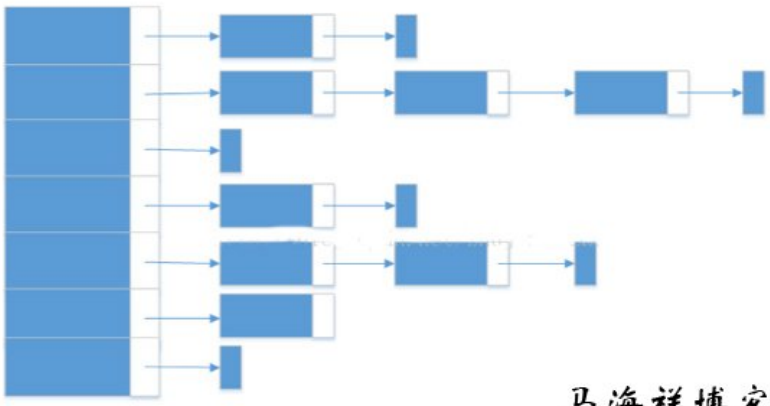


图 5-1 链接桶哈希结构

链接桶哈希是一个静态的结构，可用于内存中与磁盘中，因为它是静态结构，不用对数据进行重组织，所以它速度很快。

但这也是它的缺陷，面对动态数据，就显得不合适了，因为链接桶哈希必须在使用之前知道哈希表的大小，而这恰恰很难预测。

如果预测的表大小过小，其性能会大受影响；如果过大，空间浪费较为严重，最好情况下，它只有一些空间的浪费，用来存放指向下一个桶的指针。

2、可扩展哈希

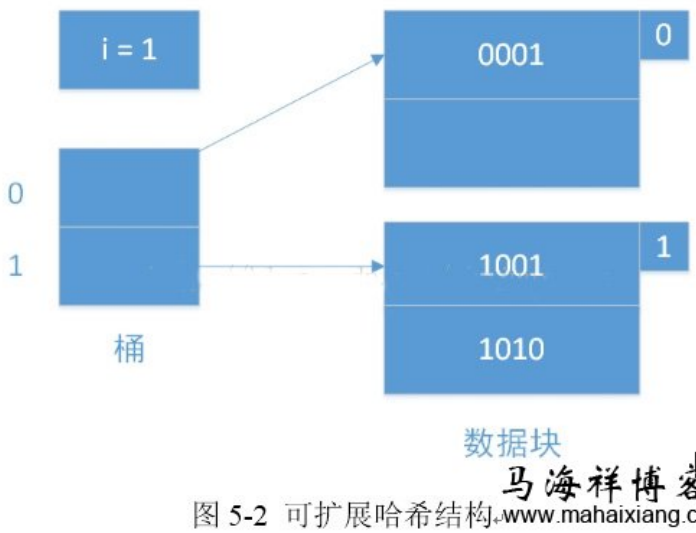


图 5-2 可扩展哈希结构

可扩展哈希引入了目录文件的概念，采用可随数据增长的动态哈希表，因此克服了链接桶哈希的缺陷，其哈希表大小不需要预先知道，一个哈希节点包含多个项，当节点数量溢出时将其分裂为两个节点，目录按2的指数倍增长，当一个节点装满而且到达了一个特定的目录大小目录就会倍增。

哈希函数为每个键计算一个K位的二进制序列，桶的数量总是使用从序列第一位或者最后一位算起的若干位[]，但是可扩展哈希的一个问题是任意一个节点都会引起目录的分裂，当哈希函数不够随机时，目录很可能增长的很巨大。

3、线性哈希

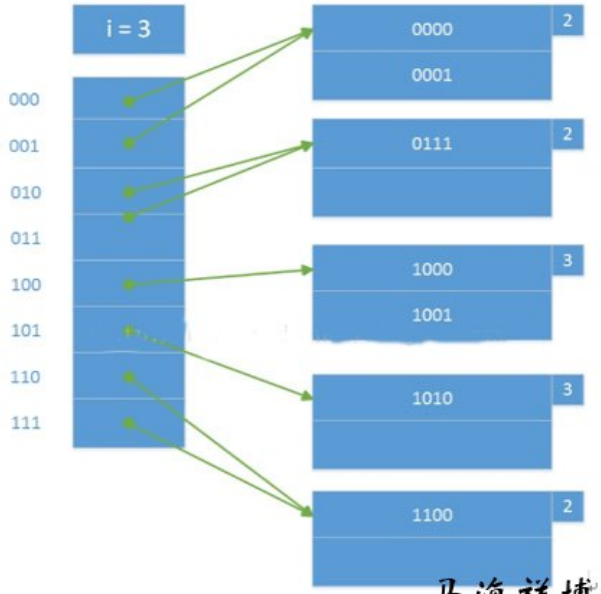


图 5-3 线性哈希结构

线性哈希也使用动态的哈希表，但是同可扩展哈希有较大差别，线性哈希选择桶数总是使存储块的平均记录保持与容量成一个固定的比例，而且哈希桶不总是可以分裂，允许有溢出块，当插入的记录没有对应的桶，将其哈希值首位改为0，再次插入，否则直接插入对应桶或其溢出块中，当记录数量比容量达到一个阈值，增加一个桶，再分配。

相对于可扩展哈希，线性哈希的增长较为缓慢，重组织的次数和代价都较小，同时，线性散列不需要存放数据桶指针的专门目录项，且能更自然的处理数据桶已满的情况，允许更灵活的选择桶分裂的时机。

4、修正的线性哈希

修正的线性哈希相对于线性哈希主要面向内存环境，通过使用更大的连续节点替代目录，普通的线性哈希由于有空节点而浪费空间，而且，除非有一个巧妙的方案解决潜在的虚拟内存映射机制问题，不然每次目录增长时那个连续的节点都要被拷贝到一个更大的内存块。

修正的线性哈希采用跟可扩展哈希一样的目录，除了目录为线性增长的，链接的是单个项目的节点和分配内存是从一个常规的内存池。

这个算法节点分裂的准则是基于性能，举例来说，监控哈希链的平均长度比监控存储利用率能够更直接的控制平均搜索和更新时间。

5、Hash预取算法

Hash预取算法面向的是键和哈希值都是32位的场景，特地对内存环境进行了优化，此算法使用乘法散列，这种方法十分普遍、计算高效，更重要的是适用于矢量，达到了一

次计算多个哈希函数的目的。

针对现代处理器的SIMD架构，将键值与哈希值共同放在一个指令当中，达到大大减少指令数的目的，令每次所需的数据长度恰好等于L2的cacheline，大大降低了性能代价，在内存环境中，大大提高了cache的性能。

马海祥博客点评：

索引是为检索而存在的，如一些书籍的末尾就专门附有索引，指明了某个关键字在正文中的出现的页码位置，方便我们查找，但大多数的书籍只有目录，目录不是索引，只是书中内容的排序，并不提供真正的检索功能，可见建立索引要单独占用空间；索引也并不是必须要建立的，它们只是为更好、更快的检索和定位关键字而存在。

打赏

相关标签搜索： 数据库 内存数据库 索引技术

上一篇：自然语言处理的单词嵌入及表征方法
下一篇：如何收集和存储服务器运营的数据

相关文章推荐：

1 HTTP与HTTPS的区别
2 关于大型网站架构的负载均衡技术详解
3 自然语言处理的单词嵌入及表征方法
4 深入解析互联网协议的原理
5 HTTPS建设使用的方案教程解析
6 HTTP、SSL/TLS和HTTPS协议的区别与联系
7 今日头条的个性化推荐算法
8 如何开启苹果系统的两步验证机制，避免
9 HTTP服务的七层架构技术解析及运用
10 基于眼球追踪技术对用户调研的探讨研究

您可能还会对以下这些文章感兴趣！

详解大型网站系统的特点和架构演化发展历程

大型网站的挑战主要来自庞大的用户，高并发的访问和海量数据，任何简单的业务一旦需要处理数以P计的数据和面对数以亿计的用户，问题就会变得棘手，大型网站架构主要就是解决这类问题。大型网站不是从无到有一步就搭建好一个大型网站，而是能够伴随小型网站业务的渐进发……【查看全文】

阅读：853 关键词： 大型网站 网站架构 网站系统 日期：2017-03-02

计算机的开机启动原理

计算机从打开电源到开始操作，整个启动可以说是一个非常复杂的过程。总体来说，计算机的整个启动过程分成四个阶段：第一阶段：BIOS；第二阶段：主引导记录；第三阶段：硬盘启动；第四阶段：操作系统；直至执行/bin/login程序，跳出登录界面，等待用户输入用户名和密码。……【查看全文】

阅读：3039 关键词： 计算机 计算机启动 计算机原理 开机启动原理 日期：2014-01-16

今日头条的个性化推荐算法

互联网给用户带来了大量的信息，满足了用户在信息时代对信息的需求，但也使得用户在面对大量信息时无法从中获得对自己真正有用的那部分信息，对信息的使用效率反而降低了，而通常解决这个问题最常规的办法是推荐系统。推荐系统能有效帮助用户快速发现感兴趣和高质量的信……【查看全文】

阅读：12908 关键词： 今日头条 日期：2016-01-20

HTTP、SSL/TLS和HTTPS协议的区别与联系

HTTPS是为了安全性而设置的，要验证很多的信息，相对应http请求的速度肯定有点慢，如果使用HTTPS的话很麻烦的，无意给服务器和客户端增加了很大的压力，所以平时最好不要使用HTTPS，如果牵扯到个人隐私或者是其他的什么重要信息就一定要这么做了，很多的时候你感觉有点问题，……【查看全文】

阅读：14035 关键词： http ssl https https协议 日期：2016-05-13

HTTPS建设使用的方案教程解析

百度已对部分地区开放HTTPS加密搜索服务，随后，百度实行全站化HTTPS安全加密服务，百度HTTPS安全加密已覆盖主流浏览器，旨在用户打造了一个更隐私化的互联网空间、加速了国内互联网的HTTPS化。同时也希望更多网站加入到HTTPS的队伍中来，为网络安……【查看全文】

阅读：42 关键词： seo https 日期：2018-02-01

详解内存数据库中的索引技术

传统的数据库管理系统把所有数据都放在磁盘上进行管理，所以称作磁盘数据库（DRDB:Disk-Resident Database），磁盘数据库需要频繁地访问磁盘来进行数据的操作，磁盘的读写速度远远小于CPU处理数据的速度，所以磁盘数据库的瓶颈出现在磁盘读写上，基于此，内存数据库的概……【查看全文】

阅读：3257 关键词： 内存数据库 索引技术 数据库 日期：2015-01-09

基于贝叶斯推断应用原理的过滤垃圾邮件研究

随着电子邮件的应用与普及，垃圾邮件的泛滥也越来越多地受到人们的关注。而目前正确识别垃圾邮件的技术难度非常大。传统的垃圾邮件过滤方法，主要有关键词法和校验码法等。前者的过滤依据是特定的词语；后者则是计算邮件文本的校验码，再与已知的垃圾邮件进行对比。它们……【查看全文】

阅读：855 关键词： 贝叶斯推断 贝叶斯应用 贝叶斯原理 过滤垃圾邮件 垃圾邮件 日期：

如何开启苹果系统的两步验证机制，避免iCloud帐号遭到攻击

首先，你需要登录至苹果的网页版Apple ID管理系统，你需要点击“管理你的Apple ID”，随后输入帐号密码信息。在登录之后，你需要从左侧导航栏中选择“密码和安全”选项，在这里，你将需要验证安全问题，随后下拉至“两步验证”区域，点击蓝色的“开始”链接并阅读其中的……【查看全文】

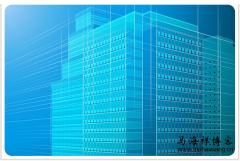
阅读：1407 关键词： 苹果系统 验证机制 icloud攻击 icloud帐号 icloud 日期：2014-09-

HTTP服务的七层架构技术解析及运用

一般来说，计算机领域的体系结构普遍采用了分层的方式，从最底层的硬件往高层依次有：操作系统->驱动程序->运行库->系统程序->应用程序等等。从网络分层模型OSI来讲，由上至下为：应用层->表示层->会话层->传输层->网络层->数据链路层->物理层。当然实际应用的TCP/IP协……【查看全文】

阅读：4386 关键词： 七层架构解析 七层架构运用 七层架构技术 http服务 日期：2014-09-

关于大型网站架构的负载均衡技术详解



负载均衡是将负载（工作任务，访问请求）进行平衡、分摊到多个操作单元（服务器，组件）上进行执行，是解决高性能，单点故障（高可用），扩展性（水平伸缩）的终极解决方案。面对大量用户访问、高并发请求，海量数据，可以使用高性能的服务器、大型数据库，存储设备，高性能W……【查看全文】

阅读：809 关键词： 大型网站 网站架构 负载均衡 日期：2016-08-05

↓ 点击查看更多 ↓