《libnindex，常用数据索引技术浅析》



libnindex推荐语：有了索引库，妈妈再也不用担心我的程序啦！

发表者简介：nickwu(吴一飞)，任职OMG微博事业部，主要负责广告平台数据监控等系统的开发与维护工作。

1. **libnindex是什么？**

libnindex是一款通用索引库，包括：多阶HashTable、红黑树、最大堆、最小堆、三元搜索树、KD树、位图、布隆过滤器等。并且支持共享缓存和文件映射存储方式。其内部采用C++模板设计，对业务无缝支持，具有良好的适配型。函数库API较简单，在保证执行效率的同时，大大降低开发成本，提升开发效率。

1. **常用数据索引结构**

在构建后台数据搜索服务时，我们经常会遇到如下几种情况，接下来我们将逐层分析其中通常使用的数据索引结构。

1. 基于Key ＝> Value的数据查询服务：

通常我们使用的NoSQL数据库都是基于HashTable构建的KeyValue服务，在通用索引库中提供两种常用的HashTable数据结构：HashTable和TimerHashTable。

HashTable：采用多阶哈希表实现。通常使用场景为，定长KeyValue索引。使用前我们先确定好元数据结构和系统容量，再通过Seed类构建素数表，创建或加载哈希表时传递进去，就可以进行KeyValue索引了。

TimerHashTable：同HashTable，但加入了元素生命周期特性，每个索引都可以设置一个过期时间。当索引过期后，元素所在的桶会被系统回收，以便下次继续分配。通常使用场景为各种营销活动，用户只能在24小时内参与X次活动。也可以用作热点数据缓存，即访问频繁数据延长元素有效期，冷数据系统自动过期释放内存资源。

**HashTable / TimerHashTable**

**struct** MyKey {

**uint64\_t** ddwUserId;

**uint32\_t** dwSubId;

};

**struct** MyValue {

**char** sName[16];

} \_\_attribute\_\_((packed));

*// load hash table;*

HashTable**<**MyKey**&**, MyValue**>** ht **=** HashTable**<**MyKey**&**, MyValue**>::**LoadHashTable(...);

*// initialize key*

MyKey key;

memset(**&**key, 0, **sizeof**(MyKey));

key.ddwUserId **=** 2657910038;

key.dwSubId **=** 2000;

*// find value*

MyValue**\*** pItem **=** ht.Hash(key);

**if**(pItem **!=** NULL)

printf("name: %s\n", pItem**->**sName);

1. 基于布尔值判断的白名单等数据查询服务：

布尔值判断是我们最常见的业务需求，例如：白名单、标志位等等。当然我们也可以用哈希表实现所有功能，不过Bitmap和BloomFilter在内存占用率方面更有优势。

**Bitmap**

Bitmap**<uint64\_t>** bm **=** Bitmap**<uint64\_t>::**CreateBitmap(4096);

**uint64\_t** ddwTweetId **=** 312590019750805;

*// test for the presence*

**if**(bm.Contains(ddwTweetId))

**return**;

*// set the bit*

bm.Set(ddwTweetId);

bm.Delete();

**BloomFilter**

BloomFilter**<**std**::**string**>** bf **=** BloomFilter**<**std**::**string**>::**LoadBloomFilter(...);

std**::**string strCrawlUrl **=** Scheduler.Next();

*// determine whether the url being crawled*

**if**(**!**bf.Contains(strCrawlUrl))

{

*// crawl the url*

.....

bf.Add(strCrawlUrl);

}

1. 字典查询、前缀搜索服务：

在使用不同业务的搜索功能时，智能提示已经成为系统标配。三元搜索树能够帮助我们有效构建相关数据索引，首先离线计算数据字典，例如利用最大堆、最小堆统计热榜TopN，然后将结果数据添加到三元搜索树构建字典表索引。

TernarySearchTree：三元搜索树支持动态构建，内部采用红黑树实现动态平衡。从算法角度来看，三元搜索树解决类型串的前缀搜索需求。默认类型是char，可以自定义成任意类型，例如加入权重值，并通过定义比较函数完成在指定前缀下按照权重值进行数据排序（默认按照字符顺序排序）。

**TernarySearchTree**

**struct** Value {

**uint32\_t** TweetID;

} \_\_attribute\_\_((packed));

**struct** dict {

**const** **char\*** str;

} dicts[] **=** { {"alpha"}, {"abs"}, {"a"}, {"bay"},

{"baby"}, {"大家好"}, {"大学"}, {"大小"} };

TernaryTree**<**Value**>** tt **=** TernaryTree**<**Value**>::**CreateTernaryTree(**…**);

**for**(**size\_t** i**=**0; i**<sizeof**(dicts)**/sizeof**(dict); **++**i) {

Value**\*** pValue **=** tt.Hash(dicts[i].str, true);

**if**(**!**pValue)

**break**;

pValue**->**TweetID **=** i;

}

**const** **char\*** pPrefixStr **=** "大";

TernaryTree**<**Value**>::**TernaryTreeIterator iter **=** tt.PrefixSearch(pPrefixStr);

**char** buffer[10];

memset(buffer, 0, 10);

**size\_t** size **=** 0;

Value**\*** pValue **=** NULL;

**while**((pValue **=** tt.Next(**&**iter, buffer, 10, **&**size)) **!=** NULL)

{

printf(" %s, TweetID:%u\n", buffer, pValue**->**TweetID);

memset(buffer, 0, size);

}

1. kNN、LBS等多维数据搜索：

上述我们所讨论的主要是一维空间的数据索引问题。除此之外我们还有可能遇到多维空间的数据搜索问题。例如：LBS的查看附近的人，kNN算法匹配与你最相似的人等等。

KDTree：k-d树，是k-维树的缩写，常用于多维空间的搜索服务，例如：图像拼接应用中，用于匹配SIFT特征点，也可作为kNN（k-Nearest Neighbor）算法中的落地索引，LBS中“附近的人”可以看作是二维空间中的特例。为了保证树的平衡，通用索引库采用静态构建的方式创建一个棵多维索引树（准备好所有数据样本，执行build方法进行索引构建），这样在一定程度上保证索引的查询效率。

**KDTree**

**struct** Store {

**uint64\_t** StoreId;

];

Store**\*** pBuffer **=** (Store**\***)malloc(**sizeof**(Store)**\***10);

gps **=** {x, y};

**uint32\_t** range **=** 100;

kdtree.Range(gps, range, pBuffer, 10);

kdtree.Delete();

1. **构建复杂数据查询业务**

在这里我们来看一个实际中的案例：

在线上业务运行中，我们不仅仅要实时监控系统的PV/UV，互动量等数据，还要根据不同属性进行分类统计和查询，以便于运营人员可以在第一时间得到数据反馈。

对于上面这种问题，复合索引就是一个比较简单和便捷的解决方案。就像数据库索引设计一样，在建立复合索引时，我们需要考虑我们的查询条件是什么？数据的排序规则是什么？是否可以合并索引？

首先我们来看下面这张表结构：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| UIN | Name | Age | Gender | Exp |
| 10000 | User1 | 18 | 0 | 100 |
| 21000 | User2 | 25 | 1 | 98 |
| 22010 | User3 | 22 | 1 | 76 |
| ………. | | | | |
| 22500 | UserN | 35 | 0 | 85 |

当我们要获取“22岁男性用户经验值Top10的用户”以及“统计男性用户的数量”，转换成SQL语句其实很简单：

SELECT `UIN` FROM `t` WHERE `Age`=22 AND `Gender`=1 ORDER BY `Exp` DESC LIMIT 10;

SELECT COUNT(\*) FROM `t` WHERE `Gender`=1;

如果我们需要按照上面搜索条件建立索引的话，需要建立两个索引：

ALTER TABLE `t` ADD INDEX(`Age`, `Gender`, `Exp` DESC);

ALTER TABLE `t` ADD INDEX(`Gender`);

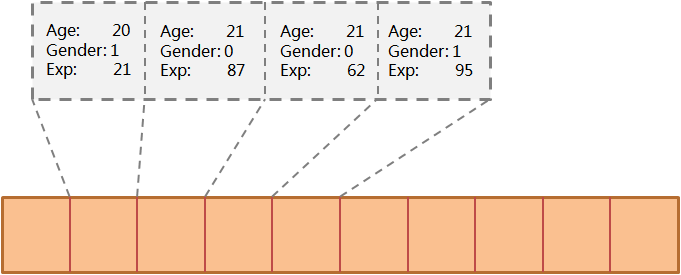
当数据库有写操作时，系统会更新两条索引结构。其实我们可以通过修改SQL查询语句达到合并索引的效果，以提升数据库写入性能。第一条SQL就是简单翻译“22岁男性用户经验值Top10的用户”的语句，他的执行结果与“男性22岁经验值Top10的用户”没有区别，转换成SQL语句为

SELECT `UIN` FROM `t` WHERE `Gender`=1 AND `Age`=22 ORDER BY `Exp` DESC LIMIT 10;

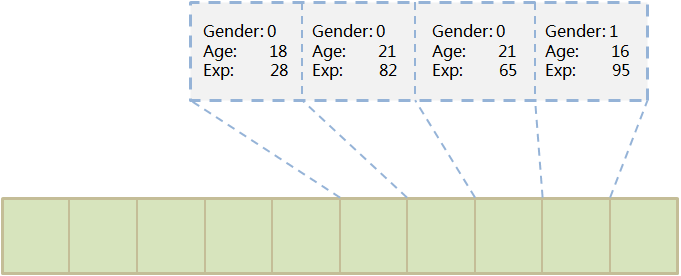
ALTER TABLE `t` ADD INDEX(`Gender`, `Age`, `Exp` DESC);

这两条语句执行结果虽然没有差别，但是索引内部构造却有天壤之别。第一个索引是以年龄为优先排序字段其次是性别和经验值，而第二个索引是以性别为优先排序字段之后是年龄和经验值。

以年龄为优先排序的索引：



以性别为优先排序的索引：



从上图我们可以看出索引内部排序规则是根据复合索引指定字段的先后顺序和字段内顺序决定的。

在通用索引库中，RBTree能对这类复杂索引结构提供良好的支持。首先我们需要定义索引Key和Value的数据结构：

*struct Key {*

*uint8\_t Gender;*

*uint8\_t Age;*

*uint8\_t Exp;*

*};*

*struct Value {*

*uint32\_t Uin;*

*};*

编写索引Key的比较函数以确定索引顺序：

*template<>*

*struct KeyCompare<Key> {*

*static int Compare(Key key1, Key key2) {*

*if(key1.Gender > key2.Gender) return 1;*

*else if(key1.Gender < key2.Gender) return -1;*

*else {*

*if(key1.Age > key2.Age) return 1;*

*else if(key1.Age < key2.Age) return -1;*

*else {*

*if(key1.Exp > key2.Exp) return -1;*

*else if(key1.Exp < key2.Exp) return 1;*

*else return 0;*

*}*

*}*

*}*

*};*

在比较函数中，我们优先比较性别字段，如果性别字段相等再按照年龄字段排序，最后则是经验值字段，并且经验值字段是按照DESC排序。这样我们在遍历红黑树时就会按照经验值由大到小排序。

*RBTree<Key, Value> rbtree = RBTree<Key, Value>::LoadRBTree(…);*

*Key keyBegin = {1, 22, 255};*

*Key keyEnd = {1, 23, 255};*

*RBTree<Key, Value>::RBTreeIterator iter = rbtree.Iterator(keyBegin);*

*RBTree<Key, Value>::RBTreeIterator endIter = rbtree.Iterator(keyEnd);*

*int i = 0;*

*Value Top10Buf[10];*

*Key key;*

*Value\* pValue = NULL;*

*while(iter != endIter && (pValue = rbtree.Next(&iter, &key))) {*

*if(i >= 10) break;*

*memcpy(&Top10Buf[i], pValue, sizeof(Value));*

*++i;*

*}*

在上述代码中，我们只需要确定查找条件的取值范围，就可以通过迭代器遍历符合条件的红黑树节点，得到最终的用户数据。

1. **libnindex的适用性**

libnindex通用索引库虽然支持mmap这样的落地解决方案，但是其主要数据结构和算法更适合于内存操作，索引库正在开发btree、rtree这样基于磁盘特性的数据结构和算法，以便后续支持更多的应用场景。

索引库目前并没有支持处理并发的安全机制，建议采用数据crc校验或者全表锁来控制并发写带来的风险，此外也可以通过数据垂直和水平拆分来降低服务负载。