# 线性现代CPU缓存敏感哈希优化型内存数据库

黄宇岳 黄增士？？张亚？？林传文？？

**摘要：**随着芯片制造技术的提升及其成本的降低，当今许多数据库都可以完全在内存中维护。然而，为了在内存中管理数据而重新使用基于硬盘存储的索引型数据结构不能达到最佳性能，其原因在于并未充分利用缓存和现代CPU的性能。因而，许多基于内存优化的索引型数据结构相继出现。这些数据结构还只关注于单键值查找，忽略了同样重要的范围查询。本文提出一种线性缓存敏感型跳表（Line Cash Sensitive Skip List, LCS-SL），其在范围查询中表现最佳且能够充分利用现代CPU的特性。LCS-SL基于一种缓存友好的数据布局和一种特别的遍历算法，该算法能最小化缓存失配和分支预测失误，并且可以将SIMD指令集用于搜索。在本文的实验中，LCS-SL在范围查询上的表现优于所有其他主要对比数据结构。因而可以将LCS-SL视为一种在内存中进行键值范围混合查找工作时的可选数据结构。

**关键词：**线性，直接寻址，缓存敏感，内存数据库，哈希索引优化，并行查找，缓存

## 引言

过去几年，不同的基于索引的数据结构被设计出来用于在主存中高效迅速得查找数据，如ART树、缓存敏感的B+树。为了减少缓存失配，改进Cache Line的使用，和利用向量指令，这些数据结构比传统索引结构的数据库数据结构（如B+树）更为优越迅速。当然B+树是为尽可能减少磁盘读取设计出来的。这些新奇的索引方法关注于单键值查找，在范围查找上使用了次优的解决方案。但无疑范围查找在各种实际业务和科研领域中有着数量可观的使用场景。

现代多处理器的硬件架构，用增加处理器核数的方法提升程序并发性能来代替原来的单纯增加处理器频率的做法，这种趋势使传统算法面临着多核并行处理的挑战。以NUMA架构为例，在多线程情况下，每一个CPU核中非共享缓存的数据的更新，会导致其他核的非共享的缓存数据同时保持更新。

跳表是一个有趣的可选的数据结构，它提供了对数级时间复杂度，把存储的元素存放在连续的内存空间中，但是没有提供一个全局的重新平衡的方法，这在某些时候会造成性能瓶颈。频繁的指针跳跃寻址会造成大量的缓存失配问题。

此篇论文中，我们引入一个由跳表启发的线性缓存敏感数据结构。这是一个主存索引数据结构，为在现代CPU上进行范围查找进行了特殊的设计。它采用了一个非常不同的内存布局策略，以获得在现代CPU的优势，诸如CPU Cache Lines，SIMD指令，流水线操作，来进行更快的查找。在我们的研究中，我们专注于读操作，并且也提供了独有的更新操作。我们能看到这个数据结构很好得与更注重读和范围读操作性能的历史数据数据库相结合。尤其是生物和金融领域，他们都有巨大的数据集，并且要对一段范围的数据进行研究。

我们将其和以下几种数据结构进行了比较，B树，红黑树，哈希表，B+树，缓存敏感的B树，和使用二分查找的数组。我们也使用了真实的金融数据用于测试。无论是单键值查找和范围查找，我们的数据结构比竞争对手普遍快了两个数量级。

论文以下的部分将具体介绍我们的数据结构，我将它命名为线性缓存敏感跳表（Line Cache Sensitive Skip List，aka LCS-SL）。第三部分介绍了LCS-SL，作为我们的主要内容，第四部分描述了查找和范围查找的算法。在第五部分，我们使用真实的金融交易数据比较了LCS-SL和其他索引数据结构。在第六部分讨论了相关的工作，第七部分包含了此篇论文的结论。

## 基本概念

传统跳表提供了一个基于概率的查找数据结构，类似于B树。跳表包含了数条包含了键值的分层的跑道。在最低层跑道中，一个跳表包含了一个已按键值排序的链表。在第i层的跑道平均包含 n\*p^i个元素，n是键值的数量，0<p<1。跳表被设计成一个基于概率的数据结构，因为存储在更高跑道层的元素是被从底层跑道中随机选择的。每一个i跑道中的元素出现在i+1跑道中的概率为p。这就允许了更高效的更新和插入，也造成了数据结构更难被预测。

在我们的工作中，我们使跳表更加的确定，所以可以被称为完美平衡的跳表。在平衡的跳表中，第i+1层的跑道包含了第i层的1/p个元素。于是，在p=0.5时，一个i+1的跑道包含了在第i层跑道中每2个元素中开始的那个，这使跳表变成了一个平衡搜索树。配图1表示了一个包含了9个整数并且p=0.5的平衡跳表。

在p值比较低的情况下，跑道会跳过许多元素，因此， 跑道能被认为是稀疏的。在高P值的情况下，跑道仅跳过相对较少的元素，因此，跑道可以被认为是稠密的。跑道被用于缩小数据链表的可能的查找范围，避免遍历。例如，在配图1的跳表中搜索键值为6的数据，首先查找在最高层的跑道中使用头结点定位到5，接着向下进入下一层跑道，依旧定位到5，进入到下一层，开始查找6，直到找到，或者遍历结束。

在一个p=0.5完全构建好的平衡跳表中，最坏查找效率为O(log(n))，p直接影响了跳表的结构，应该根据期望的元素数目谨慎选择。如果p过高，在查找时虽然每条跑道比较次数变少，但是需要更多的跑道层级来完全构建一个平衡的跳表。如果p值太低，查找时每层跑道的比较次数会增加，但是构建平衡跳表的层级会减少。

除了单键值查找，跳表也提供非常有效的范围查找，因为数据链表以顺序保存，实现一个范围查找要求两步，1.查找范围内的第一个元素，2.遍历数据链表查找所有范围内的元素。

传统跳表每个跑道维护了许多指针，这样的内存布局并不适合现代的CPU，因为这会因为在两个非连续内存之间跳转而产生大量的内存失配。例如，在一个存储了32位整数键值，有5层跑道的跳表中，每一个节点需要 4 bytes + 6 \* 8 bytes = 52 bytes。在64字节的Cache Line里，每一步的操作需要几乎一整个cache line，虽然只有很少的部分会用到。基本上，跑道中的每一步操作只需要键值和一个指针去找到下一个元素，也就是说 4 bytes + 8 bytes = 12 bytes。

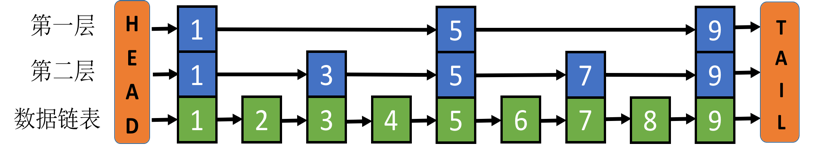


图1跳表数据结构示意图

## 线性缓存敏感跳表

在此篇论文中，我们引入线性缓存敏感跳表（Line Cache Sensitive Skip List，aka LCS-SL）作为一个可选的平衡跳表的实现。它使用了一个不同的激进的内存布局策略，以最有效率的方式利用现代CPU的架构。首先也是最为明显的想法是保持跑道在一个逻辑上被分割成不同部分的稠密数组中。这导致了很少的缓存失配，改进了Cache Line的使用率。并且使用SIMD指令。配图2表示了一个维护了32个整数键值、p=0.5的LCS-SL。查找键值7的搜索路径用红色点线表示。

LCS-SL的主要贡献在三个方面：

1. 首先跑道是线性的，并且由一个稠密的数组维护，我们叫它线性跑道数组，来代替把节点存在链表中。这显然在执行查找和范围查找时改进了Cache Line的使用率。
2. 使用线性跑道数组我们再也不用指针了。对于一个给定的n，跑道中的元素数量是已知的，因为我们构建的是一个平衡跳表。因此我们可以简单得在数组中计算位置获得需要的元素位置，这使指针成为多余。在配图2中，跑道中无指针的查找方式以点虚线箭头表示。在我们目前的实现中，我们始终为每个跑道基于假设的最大的键值数量t预分配一个确定大小的内存。只要n < t，所有的插入都能在数据结构中。一旦n超过了t，我们重建跑道，并且以一个确定的值增加t。（详见下文）
3. LCS-SL利用SIMD指令在范围查找中来比较符合的键值。这在大范围的查找中尤其有用。LCS-SL应该是第一个大量采用了SIMD指令用于范围搜索的索引数据结构。

我们在LCS-SL上的成果对比传统的跳表有如下好处：

1. LCS-SL需要的内存更小。设k为键值的大小，r为指针的大小，忽略数据对象的大小。传统跳表需要n \* (m \* r + r + k)的空间，然而LCS-SL只需要 n\*(r+k)+zgama(m->i=1)p^i \* n \* k。
2. 线性跑道数组的每一次操作有更好的cache line使用率，因为我们始终使用一整条cache line的内容，直到我们取消搜索并且跳转到下一层。在32位键值的情况下，16个元素适合64字节的cache line。
3. 数组结构的跑道存储允许SIMD指令的使用，启用了数据层面的并行。设s为SIMD寄存器的大小，k是键值的大小，s/k个元素可以并行比较。通常现代CPU的SIMD寄存器有128或者256位，因此4个或者8个32位整数能同时被指令处理。此篇论文中，我们使用Intel的AVX指令，它支持256位的SIMD寄存器。

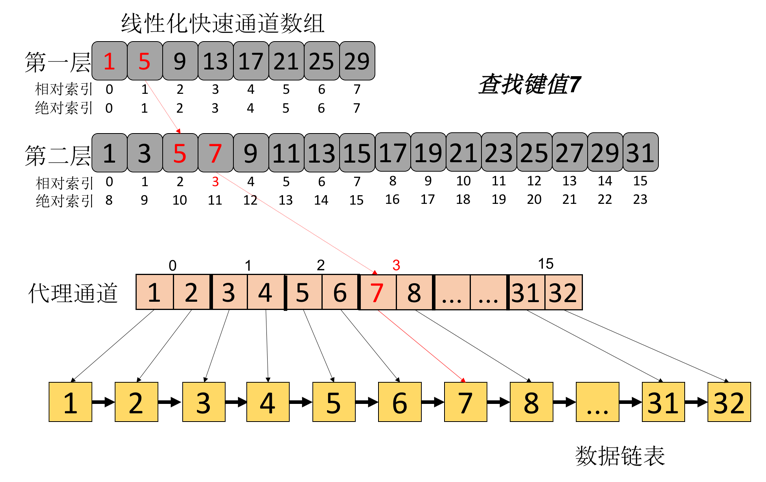


图2 LCS-SL单键值查询算法

### 优化

除了这些主要的概念以外，我们在现代CPU上采取了更多的优化。

首先，我们总是使跑道的大小为CPU Cache Line大小的整数倍。

第二，我们在跑道和数据链表中间引入了另外一条跑道，称为代理跑道。对于每一个键值代理跑道维护了一个指向其关联数据对象的指针。代理跑道也是一个数组型的数据结构，并且下层跑道已经表明了，跑道中第i个的元素能在代理跑道中第i-1个位置中找到。我们使用代理跑道来连接最底层的跑道和数据链表。

第三，在实践中我们观察到若最高层包含了大量的元素，则在最高层查找时非常耗费CPU，在最坏的情况下，我们可以能会遍历整条跑道，即使搜索下面的跑道时每条跑道的比较次数不会超过1/p。所以我们在最高层跑道采用二分查找法来代替序列遍历。

### 更新

在我们的实现中，一个LCS-SL是从一组被预排序的键值中建立的，尽管如此，我们也支持实时更新操作。下面我们将描述如何插入新的键值、更新已经存在的键值以及删除键值。

1. 插入键值：因为LCS-SL的跑道是由稠密数组维护的，所以为了维护键值的有序性，直接插入键值会导致很多拷贝操作，因为这个原因，新的键值只被插入到由链表维护的数据链路层。我们创造一个新的数据节点并且把它添加到一个合适的位置。跑道被重新建立分配更多的空间时，新的键值会自动添加进跑道。此外我们依然可以找到新添加的键值，如果没有在跑到中找到键值，搜索操作会进入数据链表，并且遍历它直到找到所搜索的数据。插入算法可以使用CAS技术来获得原子性。
2. 删除键值：我们不能直接从数据链表中删除数据，但是我们可以把跑道中的键值置为无效的。在删除键值的第一步中，我们需要把它从跑到中删除，只需要把相关的单元设置成null就能避免数组中的大量的内存拷贝。因此我们把需要删除的键值的单元中的值用前面的单元的元素的值代替，这就使键值删除的同时保证了数组不需要拷贝每一个后面的单元。最后一步，更改数据层指向被删除的数据的节点的后继指针。
3. 更新键值：更新操作以删除后插入操作实现。

## 算法

在这个部分中，我们将描述查找和范围查找的算法细节。我们从单键值查找开始。

查找：单键值查找的伪代码在Algorithm 1.中提供。如果搜索成功，元素的键值将被返回，如果不成功，将返回一个INT\_MAX。这个算法可以分成以下几个部分。第一，在最高层的跑道中使用二分查找。第二，剩余的跑道将逐个查找来缩小可能包含搜索数据的数据层的范围。我们使用逐个遍历的方法来查找除了最上层跑道以外的其他跑道，因为在这些跑道中我们只需要比较 1/p个元素。第三，如果快速跑道包含被查找的元素，返回，否则相关的代理节点所包含的数据层的节点将会逐一被比较，若没有找到，返回INT\_MAX。

Algorithm 1: lookup(key)

1: pos = binary\_search\_top\_lane(flanes, key);

2: for (level = MAX\_LEVEL - 1; level > 0; level--) {

3: rPos = pos - level\_start\_pos[level];

4: while (key >= flanes[++pos])

5: rPos++;

6: if (level == 1) break;

7: pos = level\_start\_pos[level-1] + 1/p \* rPos;

8: }

9: if (key == flanes[--pos]) return key;

10: proxy = proxy\_nodes[pos - level\_start\_pos[1]];

11: for (i = 1; i < 1/p; i++)

12: if (key == proxy->keys[i]) return key;

13: return INT\_MAX;

范围查找：Algorithm 2是范围查找算法的伪代码。返回第一个匹配数据和最后一个匹配数据的指针。第一，使用算法1查找第一个元素位置；第二，返回最低的一层跑道，使用向量指令搜索匹配结束元素的位置。使用 AVX，LCS-SL能同时处理8个32位整数。第三，根据第二步获得的代理节点的位置获得最后数据节点的位置。第四返回最后一个匹配查找数据的数据节点指针。

Algorithm 2: searchRange(start, end)

1: RangeSearchResult res;

2: pos = binary\_search\_top\_lane(flanes, start);

3: for (level = MAX\_LEVEL - 1; level > 0; level--) {

4: rPos = pos - level\_start\_pos[level];

5: while (start >= flanes[++pos];

6: rPos++;

7: if (level == 1) break;

8: pos = level\_start\_pos[level-1] + 1/p \* rPos;

9: }

10: proxy = proxy\_nodes[rPos];

11: res.start = proxy->pointers[1/p - 1]->next;

12: for (i=0; i < 1/p; i++) {

13: if (start <= proxy->keys[i]) {

14: res.start = proxy->pointers[i]; break;

15: }

16: }

17:sreg=\_mm256\_castsi256\_ps(\_mm256\_set1\_epi32(end));

18: while (rPos < level\_items[1] - 8) {

19: creg = \_mm256\_castsi256\_ps(

20: \_mm256\_loadu\_si256((\_\_m256i const \*) &flanes[pos]));

21: res = \_mm256\_cmp\_ps(sreg, creg, 30);

22: bitmask = \_mm256\_movemask\_ps(res);

23: if (bitmask < 0xff) break;

24: pos+=8;rPos+=8;

25: }

26: pos--; rPos--;

27: while (end >= flanes[++pos] && rPos < level\_items[1])

28: rPos++;

29: proxy = proxy\_nodes[rPos];

30: res.end = proxy->pointers[1/p - 1];

31: for (i=1; i < 1/p; i++) {

32: if (end < proxy->keys[i]) {

33: res.end = proxy->pointers[i - 1]; break;

34: }

35: }

36: return res;

## 评估

我们使用了其他索引结构的数据结构进行比较。我们比较了单键值查找，负载，空间效率。使用的数据是真实的交易所报单数据。经过评估我们使用64位整数作为键值，p = 1/7， level = 8。理论上层数的最优值应该和CPU的L1 Cache大小一致。在当前的实现中，我们采用了参数的方式来设置p和level，以综合考量不同设置所带来的改变。

我们使用了以下数据结构进行比较：B+树，B树，CSB+树，art树，使用二分查找法的数组。

我们测试的系统包含了如下的硬件：

CPU：

型号：Intel(R) Xeon(R) CPU E7-4820 v3 @ 1.90GHz

核数：63

缓存队列：64

内存：32931928 kB

硬盘：116G

gcc版本：gcc version 7.2.0 (Ubuntu 7.2.0-8ubuntu3)

所有的测试都是单线程的，所有的数据结构包括LCS-SL均采用GCC 7.2的O3等级编译，并使用folly的benchmark模块和papi收集相关信息。

### 范围查找

LCS-SL的主要目标是在使用一种根据现代CPU cache line大小进行设计的数据结构来获得高效的范围查找操作，当然也使用SIMD指令来访问跑道。这部分中，我们使用100万条真实的金融交易数据随机生成范围查找的开始和结束的64位整数键值。图3显示了各种数据结构对三个金融数据库进行范围查找的结果，其中LCS-SL的表现相对其他数据结构具有明显优势。

### 单键值查找

图5是各种数据结构对三个金融数据库进行单键值查找的结果。可以看到，LCS-SL的查找速度明显优于其他数据结构。

### 空间效率

图5为各种数据结构在单值查找时的内存占用情况，可以看出LCS-SL的空间效率明显优于其他数据结构。其中数组结构优于存在内存预分配，导致其空间效率不为最高。

图3 单键值查找性能对比

a) order数据库范围查询速度对比

b) transaction数据库范围查询速度对比

c) time-share数据库范围查询速度对比

图4 范围查询性能对比

图5 空间复杂度对比

## 结论

我们介绍了 LCS-SL，一个适用于内存数据库的用于在现代CPU上高效查找数据的索引数据结构。LCS-SL的线性跑道使用了一个CPU友好的数据布局，用于减少缓存失配，并使用了SIMD指令。我们把LCS-SL和其他数据结构进行了比较，诸如B+树，CSB+树，还有二分查找法。LCS-SL在一个真实的金融交易数据集上进行范围和单键值查找的效率上均胜过其他所有的竞争对手。LCS-SL查找性能和内存消耗受到每条跑道上的元素数量和p值影响。

## 感谢

(1) 任谈-上海华信证券，提供了BenchMark中的测试数据

(2) 许红伟-上海华信证券，收集了用于测试的数据，论文的校对

## 参考资料：

项目github地址：https://github.com/detective7th/kn\_db.git