# 面向金融数据的高效内存数据库索引算法研究①

黄宇岳1, 任谈1, 许红伟1

1(上海华信证券 上海 675000)

通讯作者: 黄宇岳, E-mail: [wangenda@163.com](mailto:wangenwang@163.com)

摘　要: 近年来, 服务器内存不断增加. 内存数据库由于其较高的吞吐量和较低的延迟获得了广泛的关注与研究. 在金融数据领域, 如股票交易等, 范围查询等大规模高频数据检索对索引算法提出了更高要求. 本文在现有跳表及哈希算法的基础上, 提出针对现代CPU体系结构的缓存敏感型索引算法, 充分利用缓存特性来构建和存储数据索引结构, 并基于CPU硬件特性对索引结构进行优化, 研究证明, 在证券类金融数据领域, 该索引算法可以充分利用CPU特性, 获得更高的吞吐量, 为金融证券交易提供可靠支撑.

关键词: 内存数据库; 索引算法; 体系结构; 金融数据

引用格式:  王恩达,王恩旺.运动目标检测中的光流扰动效应.计算机系统应用,xxxx,xx(x):x–x. http://www.c-s-a.org.cn/1003-3254/xxxx.html

Optical Flow Disturbance Effect in Moving Object Detection

WANG En-Da1, WANG En-Wang2

1(CEFC ShangHai Securities, ShangHai 675000, China)

**Abstract**: In recent years, the memory of the servers has been increasing rapidly. In-memory databases have gained widespread attention and research due to their high throughput and low latency. In the field of financial data, large-scale and high-frequency data retrieval, such as stock exchange, range inquiry, put forward higher requirements on the indexing algorithm. Based on the existing skiplist and hashing algorithm, this paper proposes a cache-sensitive indexing algorithm based on modern CPU architecture. It takes full advantage of the cache to construct and store data indexing structure and optimizes the indexing structure based on CPU hardware features. Research shows that in the field of securities financial data, the index algorithm can make full use of CPU characteristics, obtain higher throughput and provide reliable support for financial securities trading

**Key words**: optical flow disturbance; motion detection; frame difference method; binarization; moving object judgment

随着内存的增加与数据的增长, 内存数据库在工业界和学术界迅速发展[6-18]. 多个开源内存数据库, Redis[17], Memcached[24], levelDB[18]等获得了广泛的研究与应用. 数据索引算法作为内存数据库引擎中最核心的内容, 直接影响整个数据库的性能[9,10,13]. 现有开源内存数据库的索引结构, 没有充分利用现代CPU体系结构的特征, 如缓存, 硬件指令等, 因此再实际的应用中不能够充分发挥现代CPU的性能, 获得最佳的效果. 此外, 现有的索引结构更多的强调随机查询的性能, 对于金融证券交易中常用的范围查询等, 并没有做出针对性的解决方案. 本文在现有研究的基础上, 面向金融数据需求提出了结合跳表与哈希表的混合型数据索引算法, 研究了针对缓存的跳表优化模型与多算法混合方法. 针对哈希算法, 充分利用PCU硬件特性提高哈希算法性能. 在华信证券提供的真实金融交易数据集上, 针对不同服务器的测试研究表明, 所提出的模型可以有效的指导跳表针对缓存的优化过程. 优化后的混合索引算法在随机查询, 范围查询等金融交易常用操作上综合性能明显优于现有的平衡树, 跳表, 哈希等索引算法.

## 1 数据索引算法

　　目前, 现有的数据索引算法按照其数据结构可以分为三类, 树结构, 哈希索引结构以及跳表结构. 三类算法各有特点.

　　树结构. 树结构可以分为平衡树与字典树两类.平衡树是一种特殊的树结构, 其通过严格的平衡策略来限制树的高度发展, 保证了最坏情况下的查询复杂度. 以常用的平衡二叉树为例, 其左右子树的高度差不超过1.常用的平衡树有B+树[2], 红黑树[24]等. 然而在大规模数据集下, 平衡树复杂的再平衡操作会限制其性能. 不同于平衡树以键值作为树节点, 字典树用一个或多个字符表示树节点, 以搜索路径上节点字符的顺序连接来构成键值. 理想情况下, 字典树的复杂度仅与键值的长度有关, 而与键值的规模无关. 然而, 在实际使用中, 完整的字典树会带来巨大的空间开销, 以64位键值为例, 则需要（256）^64的存储空间. 自适应字典树[3]将多个字符存储在一个树节点中, 同时, 删除没有分支的树节点, 从而实现对字典树的压缩, 实现存储空间与性能的平衡, 获得了非常好的性能.

　　哈希索引结构利用哈希算法的实现索引key值到目标位置的一一映射[22]. 在理想情况下, 哈希索引在随机查询时可以获得O（1）的时间复杂度.然而, 哈希索引结构会打乱数据顺序, 因而不能够提供金融领域常用的范围查询等复杂操作.

跳表结构可以看做一种平衡树的替代结构[1]. 它通过随机的平衡策略来代替平衡树严格的强制平衡策略. 跳表具有多个层次, 在构建索引的过程中, 基于概率将每个数据存储在各个层次中. 虽然理论上并不能严格保证平衡性, 但是这种随机的平衡策略在实际应用中显示了巨大作用, 其具有和平衡树相似时间复杂度. 跳表广泛应用于Redis,levelDB等内存数据库中, 在最近的研究中, 缓存敏感型跳表在范围查询等操作上获得了极大的性能优势 [5].

在现有研究的基础上, 针对金融数据高频随机查询, 范围查询的需求, 提出结合跳表与哈希表的混合索引算法, 充分利用跳表在范围查询, 哈希表在随机查询的性能优势来获得最佳综合性能. 此外, 基于现代CPU的体系结构本文提出针对跳表的缓存优化模型, 指导缓存敏感型跳表的优化实现. 研究表明, 该优化模型可以有效指导跳表的设计与实现. 在真实的金融数据集上, 本文的混合索引算法在随机查询, 范围查询等操作获得最佳性能.

1.1 传统跳表

传统跳表（如图1）提供了一个基于概率的查找数据结构. 跳表包含了数条包含了键值的分层的跑道. 在最顶层的跑道中, 一个跳表包含了一个已按键值排序的链表. 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+1跑道中的元素出现在i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.

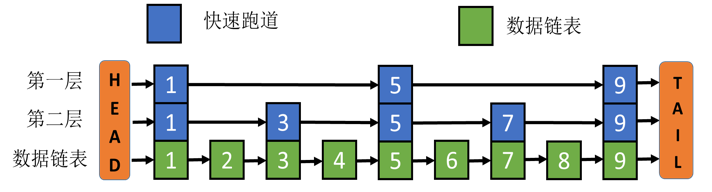


图1 传统跳表结构图

平衡跳表具有更加确定的数据结构。在平衡跳表中，第i层跑道的节点数目与第i+1层的节点数目比例为p，即底层跑道每1/p个元素会有一个在上层跑道中. 如图1所示，当p为0.5时，跳表可以看做一个平衡搜索树. 在平衡跳表中, 第i层跑道中的元素个数Ni可以由公式一计算, 其中N为总的元素个数, p为概率值, 0<p<1, L为跑道总的层次数.

 （1）

在跳表中，p的设置直接影响了跳表的结构, 在p值较低是, 跑道会跳过更多元素, 因此, 跑道能被认为是稀疏的. 在p值较高时, 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. p的设置直接影响跳表的性能. 如果p过高, 虽然在查找时每条跑道比较次数变少, 但是需要更多的跑道层级来构建平衡跳表. 如果p值过低, 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. 本文将提出针对缓存的跳表优化模型来指导p参数的设置.

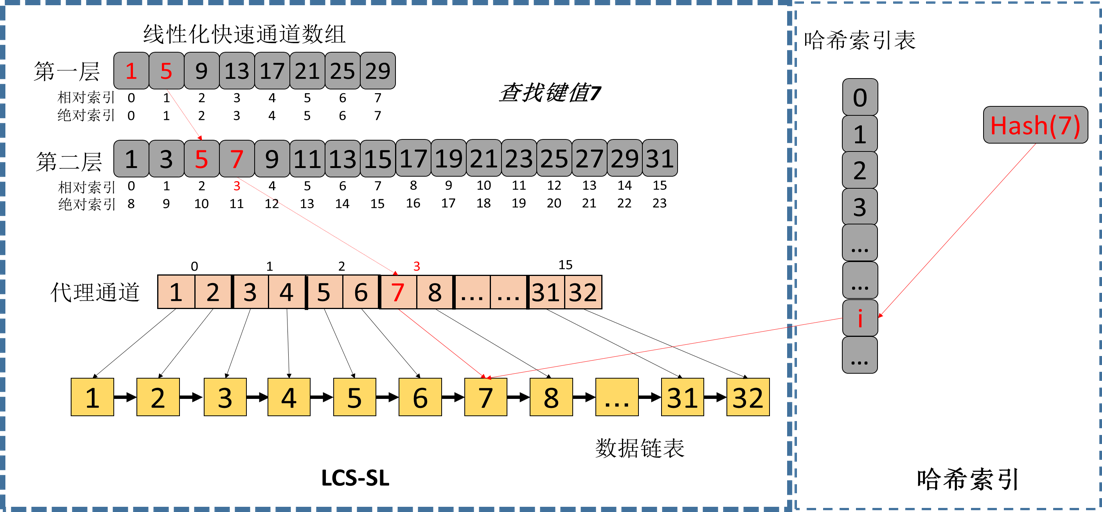
此外, 传统跳表对现代CPU体系结构的利用率较低. 传统跳表为每个元素维护多个指针, 这样的内存布局并不适合现代的CPU, 因为这会因为在两个非连续的内存之间跳转而产生大量的Cache 失效. 在传统的跳表中, 每个元素占用的空间M可以由公式2计算, 其中K为键值大小, L为跑道(图1蓝色部分)层数, V为指针大小.

 (2)

例如, 在一个存储了64位键值, 有5层跑道的跳表中, 每一个节点需要 8bytes + (5+1) \* 8 bytes = 56 bytes. 在典型的64 bytes的Cache Line中, 每一步的操作需要一个Cache Line. 然而, 在实际操作中, 跑道中的每一步操作只需要键值和一个指向下一个元素的指针, 即 8 bytes + 8 bytes = 16 bytes. 额外的存储信息导致每一步操作均需要重新读取Cache Line, 从而带来不必要的性能开销. 基于此，本文提出线性缓存敏感跳表（Line Cache Sensitive Skip List, LCS-SL），以提高Cache Line 利用率.

## 2 混合索引算法

如图2所示, 混合索引算法采用一套数据集, 两套索引结构的索引策略. 在进行范围查询时, 使用左侧的LCS-SL进行范围查询. 针对现代CPU体系结构, 我们提出了基于缓存及特殊硬件指令的LCS-SL跳表优化模型, 基于该模型指导p 值，跑道层数等相关参数调优，以在现代CPU体系结构上获得最佳的范围查询效果. 在进行随机查询时, 则使用右侧的哈希表索引算法, 在实现中, 哈希表与LCS-SL使用现代CPU特殊硬件指令如SSE4.2来进行硬件加速, 从而实现高效的范围查询与随机查询.

图2 基于LCS-SL与哈希表的混合索引结构

2.1 线性缓存敏感跳表

在此篇论文中, 我们提出了针对缓存的跳表优化模型, 并基于此模型指导LCS-SL在现代CPU体系结构上的优化. LCS-SL使用线性的内存布局策略, 以更好的利用现代CPU的缓存特点, 降低访存开销. 图2是一个简单的LCS-SL, 其中p=1/2, 跑道层次为2, 键值为32位整数. 不同于传统跳表使用链表来维护快速跑道, LCS-SL将跳表不同层次的跑道维护在一个连续的数组空间中.

LCS-SL的设计具有如下特点.

1) 跑道存储在线性的数组空间中, 通过跑道进行快速的数据检索时, 每个Cache Line中可以包含多个关键字, 进行多次比较时不需要重复的载入新的Cache Line. 因此LCS-SL可以有效减少缓存失配, 提高Cache Line利用率.

2) 取消传统跳表中的指针. 给定数据库中元素数目N, 则由公式一, 每个层次的跑道中元素数目Ni是确定的, 因此可以通过线性映射来确定跑道层次为l, 偏移为j的元素在线性数组中的下标k, 如公式三所示.

 (3)

其中p为跳表的概率. 在实现中, 基于假设的最大键值数N与公式1, 每层跑道在线性数组中获得大小为Ni个键值的空间. 当实际键值数T小于N时, 所有的插入都能在数据结构中. 当N大于T时, 则重建跑道.

3) LCS-SL利用SIMD指令在范围查找中来比较符合的键值. 通过扩展SIMD指令, 可以一次比较多个数据, 有效降低大范围的查询中的开销.

相对于传统跳表，LCS-SL具有如下优势

1) LCS-SL需要更小的内存. 设K为键值的大小, R为指针的大小, L为跳表层次数, 忽略数据对象的大小. 传统跳表需要的空间M0可以由公式（4）获得

 （4）

LCS-SL的空间M1可以由公式5获得

 （5）

以64位体系结构为例, R的大小为64位, 以金融数据中常用的64位键值为例, M1<M0, LCS-SL相对于传统跳表更加节省空间.

2) 线性跑道数组的每次操作一整条Cache Line的数据, 直到我们取消搜索并且跳转到下一层跑道, 因而可以有效提高Cache Line 利用率. 以CPU Cache Line大小为C, 键值大小为K, 则每次操作的键值数目F可以由公式6获得.

 (6)

以Intel CPU通常CPU为例, C为64 bytes, 当键值为8 bytes时, 每次操作数F为8.

3) 线性数组存储的跑道可以充分利用现代CPU的SIMD指令集. 如公式四所示, 使用SIMD指令可以同时比较H个关键字从而实现数据并行. 其中D为SIMD寄存器大小, K为键值大小. 以Intel的AVX[21]指令为例, 其支持256位的SIMD寄存器. 当键值为64位时, 可以并行比较四个键值.

** （7）

2.2 线性缓存敏感跳表优化模型

针对现代CPU可以基于如下模型对LCS-SL进行优化.

1．数据对齐. 跑道的大小为CPU Cache Line大小的整数倍并进行数据对齐. 在对跑道上的数据进行检索操作时, 保证数据可以在整数次读取中完成.

2. 在检索跑道和数据链表间引入代理通道. 对于每一个键值代理通道维护了一个指向其关联数据对象的指针. 代理通道也是一个数组型的数据结构, 并且最下层下层跑道已经表明了, 跑道中第i个的元素能在代理跑到中第i-1个位置中找到. 我们使用代理通道来连接最底层的跑道和数据链表.

3. 在实践中我们观察到若最顶层跑道包含了大量的元素, 在最坏的情况下, 查询操作会遍历整条跑道, 导致较高的计算开销, 因此在最顶层跑道采用二分查找法来代替序列遍历.

4在LCS-SL中, 完成顶层跑道的二分查找后, 在搜索下面的跑道时每条跑道的比较次数至多为1/p. 为充分利用CPU Cache Line, 保证这些比较在一条Cache Line中完成, 因而跳表参数p的设置满足公式8, 其中F为每条Cache Line 中键值的数目, 由公式6计算.

 （8）

5 基于硬件特性的参数设置.在每次查询中, 最顶层跑道由于采用二分查找的方式, 如果其占用空间较大, 可能会带来多次Cache失效从而导致性能降低. 在进行参数p和跑道层数level的设置时, 始终保持最顶层跑道的数据规模小于CPU的L1 Cache大小,如公式6所示.其中M为L1 Cache大小, K为key值的大小, L为level层数. 以N为256M, L1 cache 1MB为例, 则p为1/8,level至少为3.

 (9)

2.3 混合索引算法实现

在这个部分中, 我们将描述查找和范围查找的算法细节. 我们实现了基于哈希表与优化后的LCS-SL的混合型索引算法, 以实现高效的单键值查询与范围查询操作.

**单键值查找**：算法1提供了单键值查找的伪代码. 单键值查询基于哈希表索引结构来实现. 在理想情况下, 哈希索引结构具有O（1）的时间复杂度. 在本文的实现中, 哈希算法采用基于硬件特殊指令加速（第4行）.哈希表采用基于链表的实现方式. 当发生冲突时, 通过线性遍历存储冲突键值的链表进行查询. 如果搜索成功, 则返回数据节点; 如果不成功, 将返回空指针.

|  |
| --- |
| 算法1 基于哈希表的单键值查询算法算法 |
| lookup(key)  1: uint64\_t \*data64;  2: data64 = (uint64\_t \*) &key;  3: uint32\_t index=0;  4: index = \_mm\_crc32\_u64 (index, \*data64++);  5: Index=index%hashtablesize;  6: list=hashtable[index];  7: return listkeysearch(list,key); |

**范围查找**：范围查找返回第一个匹配数据和最后一个匹配数据的指针. 由于范围查找的起始元素与终止元素可能并不在整个数据列表中, 因此不能够直接使用单键值的查找算法来确定元素指针. 本文采用LCS-SL来实现范围查询并终返回范围的start和end对应的数据节点的指针.

|  |
| --- |
| 算法2 基于LCS-SL的范围查找[start,end]算法 |
| 1) 使用二分查找在最上层的跑道中查找start;若未找到, 则在顶层跑到中确定小于start的最大值;  2) 依次在下层的跑到中搜索start节点, 我们使用逐个遍历的方法来进行查找, 在这些跑道中, 每层跑道至多需要比较 1/p个键值;  3） 如果快速跑道包含被查找的元素, 则返回start元素对应的数据节点.若快速跑到中未找到对应的start元素, 则会进入代理通道和数据链表进行顺序查找, 并返回不大于start元素的最大值的数据节点;  4）返回最底层的跑道中start元素的位置, 使用向量指令搜索匹配end元素的位置. 以 256位AVX指令为例[21], LCS-SL可以同时比较4个64位键值;  5) 根据第4步获得的代理通道中end节点的位置并通过顺序查询获得end数据节点的在数据链表中的位置;   1. 返回start 与 end元素对应的数据节点的指针; |

2.4 数据更新

混合索引结构的数据更新实行以LCS-SL为主, 哈希表为辅的更新策略. 哈希表仅保留键值索引, 并保存指向数据节点的指针, 所有的数据保存在LCS-SL的数据链表中.

插入：插入操作实行先LCS-SL, 后哈希表的插入策略. 因为LCS-SL的跑道是由稠密数组维护的, 在插入数据时, 如要维护线性数组, 会带来过多内存拷贝等操作, 因而新的键值只插入最底层的数据链表中（图2中的黄色链表）. 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. 在基于LCS-SL的查询中, 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. 在LCS-SL插入操作结束后, 新插入的键值会在哈希表中建立索引, 并将指针指向数据链表中的数据节点.

删除：删除键值采用先哈希后LCS-SL的删除策略. 首先删除哈希表中要删除键值的索引, 数据节点的删除操作由LCS-SL完成. LCS-SL的删除操作分为两个部分, 删除跑道中的键值索引和删除数据链表中的数据节点.在删除跑道中的键值索引时, 为保证数组的空间连续性, 这里仅将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.这种方法在删除的同时保证了数组不需要拷贝每一个后面的单元, 避免大量内存拷贝带来的开销. 在删除跑到中的键值后, 删除数据链表中的数据节点.

更新：更新操作通过先删除, 后插入来实现.

## 3 实验分析

我们使用了其他索引结构的数据结构进行比较对象. 我们比较了单键值查询, 范围查询的等金融数据常用的查询操作. 数据集采用华信证券提供的真实的交易所报单数据. 其中Transaction数据是成交数据, Order数据为委托报单数据, Timeshare为分时交易数据.所有数据均采用64位数作为键值, 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[2], B树, 缓存敏感型B+树CSB+[4], 使用二分查找法的数组. 实验使用folly[19]的benchmark模块和papi[20]收集相关信息. 表一给出了测试所用的软硬件环境：

表1 系统测试软硬件环境

|  |  |
| --- | --- |
|  | 相关参数 |
| CPU型号 | Intel(R) Xeon(R) CPU E7-4820 v3 @ 1.90GHz |
| L1 Cache | 32KB |
| L2 Cache | 25KB |
| L3 Cache | 25600KB |
| 核数 | 64 |
| 缓存队列 | 64 |
| 内存 | 32931928 kB |
| 硬盘 | 116GB |
| SSE 4.2 支持 | 是 |
| 操作系统 | Ubuntu 7.2.0-8ubuntu3 |

5.1 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.

图3 单键值查找性能对比

5.1 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. 这次部分中, 我们评估使用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构, 其中LCS-SL的表现相对其他数据结构具有明显优势.

5.3 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。

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

## 4 结论与展望

本文通过结合缓存敏感型跳表与哈希表的的混合索引算法来满足金融数据的高频索引需求，并基于当代CPU的体系结构提出了针对缓存敏感型跳表的优化模型。实验结果表明，该优化模型可以有效指导缓存敏感型跳表针对硬件特性的优化过程。在真实的金融交易数据集上，基于此模型设计的混合索引算法在范围查询，随机查询等方面获得明显的性能提升。接下来的主要工作将研究将不断完善该模型，并针对不同类型的金融数据进行优化。此外，基于该混合索引算法，研究面向当代CPU多核体系结构的并行模型也是一个有价值的方向.

参考文献

1 Pugh W. Skip lists: a probabilistic alternative to balanced trees[M]. ACM, 1990.

2 B+ tree source code. <http://www.amittai.com/prose/bpt.c>

3 Leis, Viktor, Alfons Kemper, and Thomas Neumann. "The adaptive radix tree: ARTful indexing for main-memory databases." Data Engineering (ICDE), 2013 IEEE 29th International Conference on. IEEE, 2013.

4 Rao, J., & Ross, K. A. (2000, May). Making B+-trees cache conscious in main memory. In ACM SIGMOD Record (Vol. 29, No. 2, pp. 475-486). ACM.

5 Sprenger, S., Zeuch, S., & Leser, U. (2016, September). Cache-sensitive skip list: Efficient range queries on modern cpus. In International Workshop on In-Memory Data Management and Analytics (pp. 1-17). Springer, Cham.

6 郎泓钰, 任永功. 基于 Redis 内存数据库的快速查找算法[J]. 计算机应用与软件, 2016, 33(5):40-43.

7周翔宇, 程春玲, 杨雁莹. 基于分布式内存数据库的移动对象全时态索引[J]. 计算机科学, 2016, 43(7):203-207.

8石磊, 黄高攀, 乔雄. 基于内存数据库的索引算法研究[J]. 信息技术, 2016, 40(11).

9 薛忠斌, 周烜, 张延松,等. 内存列存储数据库中优化的混合自适应索引[J]. 计算机科学, 2015, 42(11):28-31.

10王澜. 内存数据库中B+树和CSB+树的性能比较[J]. 通讯世界, 2015(12):277-277.

11李诗云. 基于内存数据库Redis的众包系统性能优化[D]. 浙江大学, 2016.

12颜昌盛, 范娟娟, 海洋,等. 基于内存数据库提升铁路货车追踪应用性能的研究[J]. 铁路计算机应用, 2015(6):26-30.

13丁正, 王红熳, 段綦,等. 基于内存数据库的HBase二级索引构建与查询的装置及方法:, CN 106294814 A[P]. 2017.

14胡俊, 傅春霞, 彭立. 应用于核电站的高效内存数据库设计方案[J]. 工业控制计算机, 2017, 30(1):80-81.

15彭宇. 高速缓存技术在内存数据库中的应用分析[J]. 技术与市场, 2016, 23(11):111-111.

16王芬,顾乃杰,黄增士等.基于Bully算法的Redis集群选举方案优化[J].计算机科学,2017,44(10)

17 Zawodny, Jeremy. "Redis: Lightweight key/value store that goes the extra mile." *Linux Magazine* 79 (2009).

18 Ghemawat, S., and J. Dean. "LevelDB, A fast and lightweight key/value database library by Google." (2014).

19 Folly. <https://github.com/facebook/folly>

20 Papi. <http://icl.utk.edu/papi/>

21 Firasta, Nadeem, et al. "Intel AVX: New frontiers in performance improvements and energy efficiency." Intel white paper 19 (2008): 20.

22 Ratnasamy, Sylvia, et al. "GHT: a geographic hash table for data-centric storage." Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications. ACM, 2002.

23 Fitzpatrick, Brad, and Anatoly Vorobey. "Memcached: a distributed memory object caching system." (2011): 1-4

24 葛瑶, 李晓风, 孔德光. 基于红黑树的堆内存泄漏动态检测技术[J]. 计算机工程, 2008, 34(16):159-161.