# 基于时序参数的重复数据删除索引优化技术研究

周 斌,李 波
(中南民族大学 计算机科学学院,武汉 430074)

**摘 要** 针对系统中存在的索引检索效率问题,提出了一种基于时序参数的快速索引优化算法,该算法通过时间参数和序数参数获取数据块的热度值,将高热度值的数据块指纹组合成了一个高优先度的快速索引. 快速索引与主索引组成了重复数据删除中的两层索引结构,从而提高了系统的检索性能. 通过实验验证了基于时序参数的索引优化算法的优越性.

**关键词** 重复数据删除; 索引优化; 时序参数
**中图分类号** TP311 **文献标识码** A **文章编号** 1672-4321(2017)02-0133-05

# The Research of Data Deduplication Index Optimization Based on Time and Cites Parameters

*Zhou Bin*, *Li Bo*
(College of Computer Science, South-Central University for Nationalities, Wuhan 430074, China)

**Abstract** A fast index optimization algorithm based on time and cites parameters was proposed for the efficiency of index retrieval. It obtained the hot values of data chunks through the time and cites parameters and then the fingerprints of data chunks with high hot values were combined to a fast index with high priority. Two layer index was formed with the fast index and the main index which improved the retrieval performance of the system. Finally, the advantages of the index optimization algorithm based on time and cites parameters were verified by experiments.

**Keywords** data deduplication; index optimization; time and cites parameters

据国际数据公司(IDC)统计,2009 年的全球数据总量为 0.8ZB,而到 2010 年底,全球数据总量已经增长到 1.2ZB,2011 年,全球数据总量达到 1.82ZB,并且数据总量还将继续高速增长. 为了应对信息量的不断增长,重复数据删除作为一种较好的处理重复数据的方式,能够保证存储系统中数据块的惟一性,从而完成重复数据的检索和消除. 随着系统容量的不断扩大,重复删除系统面临新问题:即随着索引体积不断增大,已无法完整存放在内存中. 为了将新存入的数据特征值和已保存的数据特征值进行比较,需要通过系统索引进行检索,从而带来大量的硬盘访问开销. 由于硬盘的访问速度严重滞后,索引检索的效率降低,开销增大.

面对重复数据删除系统的检索瓶颈,本文主要针对重复数据删除系统的索引进行优化,提出了一种基于时序参数的重复数据删除索引优化算法. 该算法是一种基于数据块热度值的高优先度的快速索引算法,通过使用两层索引检索的方式优化索引检索的性能. 最后,本文对索引优化算法的性能进行了测试,验证了该索引优化算法的优越性.

## 1 相关研究工作

重复数据删除索引作为制约系统性能的重要环节,一直是国内外专家研究的重点. 目前被广泛使用的重复数据删除索引主要有:DDFS Index[1,2]、Sparse Index[3]和 Extreme Binning[4]等.

DDFS 是美国普林斯顿大学提出的一种针对海量数据情况下的数据指纹检索技术. DDFS 认为应当将数据作为一个整体来看待,同一数据的不同数

收稿日期 2017-02-17
作者简介 周 斌(1971-),男,副教授,博士,研究方向:大数据存储与处理,E-mail:binzhou@mail.scuec.edu.cn
基金项目 湖北省自然科学基金资助项目(2016CFB650)

据块很可能被先后存储或使用. 因此,DDFS 使用一种定长的容器来存放同一数据块及其指纹,保证这些数据块的集中存放,当 DDFS 读取某一个数据块的指纹时,会读取该数据块所在容器的所有指纹. 如果该容器中其他数据块接下来被读取,就能直接在内存中进行比较,以减少系统的 I/O 开销. 同时,DDFS 将 Bloom Filter[5] 引入到主索引中,并利用数据的局部性原理减少了系统索引检索开销. 但由于 DDFS 的 Bloom Filter 直接构建在主索引上,受制于 Bloom Filter 在删除和更新上的缺陷,DDFS 的更新较为困难.

Sparse Index 是由 Lillibridge 等人在 2009 年提出的一种稀疏矩阵抽样索引. 通过将数据块组合为数据段,并对数据段中的哈希值进行抽样,形成稀疏索引. Sparse Index 本质上是一种局部数据重复性检测,虽然能提高系统的处理效率,但由于缺乏全局性的数据判断,系统的重删率不高.

Extreme Binning 是由 Bhagwat 等人在 2009 年提出的一种基于分层思想的相似性检索. 它将相似的文件块集中起来,将文件通过滑动窗口分块技术分成数据块,使用哈希函数求出这些数据块的对应指纹,然后从中选出这个文件的具有代表性的关键数据块指纹; 接着,将关键数据块的指纹组合成一个常驻内存的体积较小的主要索引. 而剩下的数据块指纹则组成存放于此盘中的二级索引. Extreme Binning 的最大特点是将每个文件的硬盘访问次数降低到了一次. 与 Sparse Index 类似,由于 Extreme Binning 使用了数据空间的局部性及相似性特征,系统的重复数据删除被局限在局部删除,因此 Extreme Binning 的重删率不高.

## 2 基于时序参数的索引优化技术

为解决系统检索瓶颈,提高索引检索在内存中的命中率,本文构建了一种基于时序参数的索引优化技术,即快速索引.

### 2.1 热度值的计算

为了实现快速索引的高命中率,快速索引中存放的应该是能在接下来反复被检索的高价值的数据块的指纹. 系统中需要一种衡量数据价值的参数,用于判断哪些数据是高价值数据. 为了实现这种价值判断,需要使用系统清理中引入的序数参数 *Cites* 和时间参数 *Time*.

重复数据删除索引中用于标明每个数据块的引用次数的参数,被称为序数参数. 重复数据删除索引中用于标明每个数据块的最后访问时间的参数,被称为时间参数. 重复数据删除系统中,数据块在第 $t$ 期的热度值表示该数据块在时间范围 $t$ 内的系统数据价值,该值记为 $Hotkey_t$.

在重复数据删除系统中,一个数据块的热度值越高,则其数据价值越高. 高热度值数据块拥有如下特征: 在系统中被反复引用、最近多次被使用等. 由于系统拥有时间局部性,即如果一个数据块正在被访问,那么它在近期很可能还会被访问. 因此,系统能通过热度值找出当期被频繁访问的数据块,依据时间局部性原理,这些数据块在下一期中也很可能被频繁访问,因此应该将这些高热度值的数据块的指纹放入快速索引中.

考虑到数据块的热度值不应是固定不变的,应该是动态的、随时间和具体访问情况变化的,因此,数据块的热度值应该是在系统的使用过程中积累并实时生成的,定义如下:

$$Hotkey_{t+1} = (1 + Cites)\sqrt{\frac{tC_1 + Time}{tC_1}} \cdot \frac{Size}{C_2}. \quad (1)$$

公式(1)是该数据块在未考虑历史影响的条件下在第$(t+1)$期的热度值,$t$ 表示系统运行的期数(每期都会重新生成快速索引),$Hotkey_{t+1}$ 表示快速索引第$(t+1)$期某数据块的热度值,*Cites* 为该数据块的序数参数值,*Time* 为该数据块的时间参数值,*Size* 为该数据块的大小,$Hotkey_t$ 为该数据块的上期热度值,$C_1$ 和 $C_2$ 为给定的经验常数.

$(1 + Cites)$ 为该数据块的序数参数对于热度值的影响因子,表明不同数据块的不同引用次数对于数据块热度值的影响. 由于系统存在局部性原理,因此数据块的 *Cites* 越大,热度值也应越大. 同时,由于当期 *Cites* 为 0 的数据块并不会被当期清理,因此这些数据块也应该拥有对应的热度值. 在计算热度值时,使用$(1 + Cites)$而不是直接使用 *Cites*,能避免直接使用 *Cites* 参数导致的归零问题,保证所有的数据块都有可能被加入到快速索引中.

$\sqrt{\frac{tC_1 + Time}{tC_1}}$ 为该数据块的时间参数对于热度值的影响因子,表明不同数据块的不同最后访问时间对于数据块热度值的影响. $C_1$ 是一个时间常量,表示一期中时间总量. *Time* 越大,表示数据块最近被访问的时间越近,依据系统的时间局部性原理,数据块的热度值也应越大. 为了避免时间参数对热度值的影响过大,同时兼顾数据的时间局部性原理,热

度值的计算中未直接使用 *Time* 参数.

$\frac{Size}{C_2}$ 为该数据块的尺寸参数对于热度值的影响因子,表明不同数据块的尺寸对于数据块热度值的影响. $C_2$ 为数据块尺寸的最大值,在本文中为 8 KB. *Size* 越大,表示数据块的尺寸越大,重复引用时节约的空间就越大,因此热度值也就越大; 同时,在重复数据删除系统中 *Size* 参数不应该发挥过大的影响,因此用$\frac{Size}{C_2}$,即该数据块的尺寸除以数据块的最大尺寸来表示尺寸参数对数据块热度值的影响. 由于本系统中数据块的范围在4 ~ 8KB 之间,因此,这个值的结果介于 0.5 ~ 1.

但是,如果简单地使用 $Hotkey_{t+1}$ 来表示该数据块在第$(t+1)$ 期的热度值就难免会发生抖动. 为了减少抖动,避免高热度数据在某一期中的意外情况,本文引入一个热度历史系数 $\lambda$. $\lambda$ 表示上一期的热度值在当期热度值计算中所占的比重. 因此在 $Hotkey_{t+1}$ 中加入热度历史系数 $\lambda$,就得到了完整的第$(t+1)$ 期的热度值:

$$Hotkey_{t+1} = (1-\lambda)\left((1+Cites)\sqrt{\frac{tC_1+Time}{tC_1}}\cdot\frac{Size}{C_2}\right)+\lambda Hotkey_t. \quad (2)$$

## 2.2　基于时序参数的快速索引

基于数据块热度值的定义,根据公式(2),主索引对应的每个数据块都能求出在某一期中的热度值,并将这些热度值按顺序排列起来.

将主索引中热度值从大到小排列中的前 $N$($N$ 为指定的快速索引的尺寸) 个指纹组合成一个新的小规模的高优先度索引称为快速索引,其中 $N$ 的取值应当依赖于数据块的数量以及内存的容量. 由于这个快速索引中存放的都是满足在系统中被反复引用、最近多次使用等条件的数据块指纹,依据数据局部性原理,这些数据块在系统接下来的运行中也将被多次使用,从而保证了快速索引的高命中率.

当 $N$ 的取值增大时,即快速索引的容量增大,快速索引的命中率也将显著提高,但同时会增加快速索引的检索开销; 当 $N$ 的取值减小时,即快速索引的容量缩小,快速索引的命中率将有所降低. 因此,当 $N$ 过大或过小时,都会使得快速索引的性能降低,甚至失去作用. 为了充分发挥快速索引的作用,必须根据重复数据删除系统的情况,在保证命中率的前提下,确定 $N$ 的合理取值,这样才能提高系统的整体性能.

引入快速索引后的系统索引检索流程如图 1 所示,新数据块的指纹将首先在快速索引中进行检索,如果在快速索引中检索成功,则用检索到的指纹的指针代替数据块,并删除新数据块; 如果在快速索引中检索失败,则回到主索引进行二次检索.

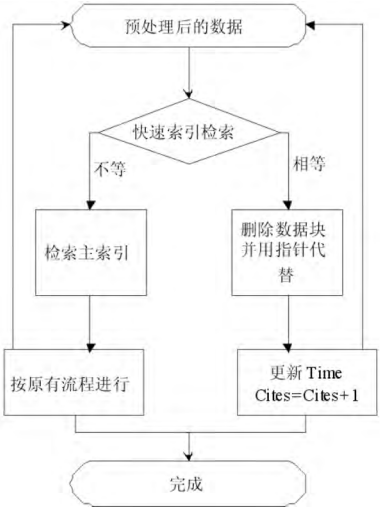


**图 1　基于快速索引的检索流程**

Fig. 1 Retrieval process based on fast index

## 2.3　快速索引的更新

数据块的热度值是随着期数 $t$ 不断变化的,因此,为了保证快速索引始终能够适应重复数据删除系统的实时需求,要对快速索引进行更新操作,将当期的高热度数据块的指纹加入快速索引中,同时清理热度值降低后不符合要求的数据块指纹.

重复数据删除系统的运行有周期性规律,可以按照工作强度分为忙时和闲时. 当系统处于忙时状态,重复数据删除系统不进行清理和更新,而是通过更新 *Cites* 参数和 *Time* 参数对需要清理的数据块进行标记,并为热度值的计算积累参数,从而节省 CPU 与存储空间. 当系统处于闲时状态,重复数据删除系统将执行清理,计算所有数据块的热度值,同时依据新的热度值重新生成快速索引,充分利用系统的 CPU 与存储资源.

# 3　快速索引的改进

如图 1 中的快速索引的检索流程,在包含快速索引的两层索引检索中,新的数据块指纹将优先在快速索引中进行检索,当快速索引中检索未命中时,返回主索引进行检索. 用 $t_1$ 来表示快速索引的平均检索开销,$t_2$ 为主索引的平均检索开销,快速索引中

的命中率为 $n$,则包含快速索引的两层索引检索平均开销 $T_a$ 为:

$$T_a = nt_1 + (1-n)t_2, \quad (3)$$

又由于在本系统中, $t_2 = 30t_1$. (4)

将式(4)带入式(3)中得: $T_a \leqslant t_2$,即包含快速索引的两层索引检索平均开销小于主索引. 因此,包含快速索引的两层索引检索的平均性能要优于主索引.

当数据块指纹在快速索引中检索未命中时,两层索引检索的平均开销 $T_b$ 为:

$$T_b = t_3 + t_2, \quad (5)$$

其中 $t_3$ 为完整检索快速索引的开销,在本系统实验环境中有:

$$t_3 \approx 3t_1, \quad (6)$$

将式(4)、(6)带入式(5)中,此时相对于直接检索主索引的额外开销为 $\frac{T_b - t_2}{t_2} = 10\%$.

为了减少这种快速索引未命中时的开销,本文将 Bloom Filter 引入到快速索引中,其未命中时平均检索时间变为 $T_b$,则:

$$T'_b = P_s(t_q + t_3) + (1-P_s)t_a + t_2, \quad (7)$$

其中 $t_q$ 为快速索引 Bloom Filter 检索时间,且在实验环境中:

$$t'_3 > 100000t_q, \quad (8)$$

将式(8)代入式(7)中, $P_s < 99.9\%$ 时, $T'_b < T_b$.

当系统中 Bloom Filter 误判率 $P_s = 20\%$ 时: $\frac{T'_b}{T_b} = 0.816$; 当误判率 $P_s$ 减小到 10% 时: $\frac{T'_b}{T_b} = 0.792$.

即在快速索引中加入 Bloom Filter 后,能明显提升快速索引未命中时两层索引检索的效率,改善重复数据删除系统的用户体验,其两层索引的整体平均检索时间为:

$$T'_a = n(t_q + t_1) + (1-n)[P_s(t_q + t_3) + (1-P_s)t_a + t_2]. \quad (9)$$

比较 $T_a$ 与 $T'_a$,即比较式(4)与式(9)可得,在实验环境下: $\frac{T_a}{T'_a} \approx 1$,即在快速索引中加入 Bloom Filter 对两层索引未增加太大的开销,但能大幅优化快速索引未命中时的索引检索开销.

## 4 性能测试与分析

本节主要测试快速索引的大小为何值时,系统拥有最高的检索效率. 实验着重测试在不同大小的快速索引的情况下,新文件加入已运行的重复数据删除系统所花费的时间.

首先将重复数据删除系统存储空间清空,接着存储 500 MB 的数据库文件; 然后,按照快速索引占主索引大小的 5%、10%、15%,即 $N = 5\%$、$N = 10\%$、$N = 15\%$ 分别生成快速索引,并进行后续实验. 从被选取的数据库文件集中分别选 10,20,…,100MB 的数据库文件,见表 1,分别通过 3 种包含不同大小快速索引的两层索引系统以及主索引系统,将其存入重复数据删除系统中,观察并记录不同索引条件下的重复数据删除系统的处理时间.

表1 快速索引性能测试

Tab. 1 Fast index performance test

| 快速索引和主索引 | 检索时间/s | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 10MB | 20MB | 30MB | 40MB | 50MB | 60MB | 70MB | 80MB | 90MB | 100MB |
| $N=5\%$ | 2.23 | 4.02 | 6.07 | 9.48 | 11.38 | 14.97 | 18.35 | 20.33 | 24.25 | 28.25 |
| $N=10\%$ | 2.12 | 3.67 | 6.25 | 9.19 | 11.32 | 14.27 | 16.95 | 20.02 | 23.63 | 26.87 |
| $N=15\%$ | 2.17 | 3.99 | 6.82 | 9.63 | 12.05 | 14.63 | 17.23 | 20.57 | 24.21 | 28.72 |
| 主索引 | 2.67 | 4.58 | 7.32 | 10.95 | 12.74 | 15.25 | 17.89 | 21.93 | 25.48 | 29.67 |

如表 1 所示,当 $N = 5\%$ 时,快速索引带来的存储及检索的额外开销较小,根据第 2 部分中提及的 $N$ 的大小对于快速索引命中率的影响,此时快速索引检索的命中率不高,导致系统检索效率不稳定.

当 $N = 15\%$ 时,快速索引带来的存储及检索的额外开销较大,同时快速索引的命中率有所提升. 在这种情况下,系统检索的性能较为稳定,但效率不是最优.

当 $N = 10\%$ 时,快速索引带来的存储及检索的额外开销较为适中,实验中系统检索的稳定性接近 $N = 15\%$,而效率优于 $N = 5\%$ 和 $N = 15\%$.

在实验环境下,当 $N = 5\%$ 时,包含快速索引的两层索引的检索效率较单纯的主索引检索提高了 16%; 当 $N = 10\%$ 时,检索效率提高了 20%; 当 $N = 15\%$ 时,检索效率提高了 18%. 因此,当快速索引的大小为主索引的 10%($N = 10\%$)时,包含快速索引

的两层索引的性能为最优.

## 5　总结

面对系统存储压力不断增大的现状,重复数据删除系统需要解决系统索引检索效率的问题. 为此,本文通过在索引中引入时序参数,并依据数据块的热度值构建了一种包含快速索引的两层索引. 实验结果证明基于时序参数的重复数据删除索引能提供高效的索引检索效率.

### 参 考 文 献


[1] Zhu B, Li K, Patterson H. Avoiding the disk bottleneck in the data domain deduplication file system [C] //USENIX. Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08). San Jose: USENIX, 2008: 269-282.

[2] Jain R, Rawat M, Jain S. Data optimization techniques using bloom filter in big data [J]. International Journal of Computer Applications, 2016, 142(3): 23-27.

[3] Lillibridge M, Eshghi K, Bhagwat D, et al. Sparse indexing: large scale, inline deduplication using sampling and locality [C] //USENIX. Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09). San Francisco: USENIX, 2009: 111-123.

[4] Bhagwat D, Eshghi K, Long D D E, et al. Extreme binning: scalable, parallel deduplication for chunk-based file backup [C] // IEEE/ACM. Proceedings of the 17th IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'09). London: ACM, 2009: 1-9.

[5] Zhang P, Huang P, He X, et al. Resemblance and mergence based indexing for high performance data deduplication [J]. Journal of Systems and Software, 2017, 128(6): 11-24.


(上接第 114 页)


[4] Talouki R N, Motameni H. Modeling sequence diagram in Fuzzy Uml to Fuzzy Petri-net for calculating reliability parameter [J]. Research Journal of Applied Sciences Engineering & Technology, 2013, 6(20).

[5] 赵俊峰, 周建涛, 邢冠男. UML 活动图到 Petri 网的转换方法及实现研究 [J]. 计算机科学, 2014, 41(7): 143-147.

[6] 王 雷, 姜久雷, 王晓峰. 基于 Petri 网的设计模式形式化描述 [J]. 计算机工程, 2016, 42(7): 33-36.

[7] Belghiat A, Chaoui A, Maouche M, et al. Formalization of mobile UML statechart diagrams using the π -calculus: an approach for modeling and analysis [J]. Communications in Computer & Information Science, 2014, 465: 236-247.

[8] Dingel J, Paen E, Posse E, et al. Definition and implementation of a semantic mapping for UML-RT using a timed pi-calculus [C] // ACM. International Workshop on Behaviour Modelling. USA: ACM, 2010: 1-8.

[9] Belghiat A. Formalization of UML Communication Diagrams using π-Calculus [C] // University of Souk Ahras. Symposium of Complex Systems and Intelligent Computing. Algeria: University of Souk Ahras. 2015: 12-17.

[10] 柳 毅, 麻志毅, 何 啸,等. 一种从 UML 模型到可靠性分析模型的转换方法 [J]. 软件学报, 2010, 21(2): 287-304.

[11] Evans A, France R, Lano K, et al. Developing the UML as a formal modelling notation [J]. Computer Science, 2014, 19(98): 297—307.

[12] Snook C, Savicks V, Butler M. Verification of UML models by translation to UML-B [J]. Formal Methods for Components & Objects, 2011, 6957: 251.

[13] Chavez H M, Shen W, France R B, et al. An approach to checking consistency between UML class model and its Java implementation [J]. IEEE Transactions on Software Engineering, 2016, 42(4): 322-344.

[14] Ekanayake EMNK, Kodituwakku SR. Consistency checking of UML class and sequence diagrams [C] // IEEE. International Conference on Ubi-Media Computing. Brazil: IEEE, 2015: 24-31.

[15] Vieweg I, Werner C, Wagner K P, et al. Unified modeling language (UML) [M]. Wiesbaden: Gabler Verlag, 2012: 367-377.

[16] Merris R. Wiley-Interscience series in discrete mathematics and optimization [M]. New Jersey: John Wiley & Sons, 2011: 409-419.

[17] Choi J, Jee E, Bae D H. Timing consistency checking for UML/MARTE behavioral models [J]. Software Quality Journal, 2016, 24(3): 835-876.

[18] Herchi H, Abdessalem W B. From user requirements to UML class diagram [J]. Computer Science, 2012.

[19] Griff E R, Matter S F. Evaluation of an adaptive online learning system [J]. British Journal of Educational Technology, 2013, 44(1): 170-176.

[20] 杨放春,龙湘明. 软件非功能属性研究 [J]. 北京邮电大学学报,2004,27(3): 1-12.