计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2016.20148040
53(2):443-448,2016

基于频繁序列挖掘的预取算法研究与实现

王 芳[1,2] 王培群[2] 朱春节[1]
[1](武汉光电国家实验室(华中科技大学) 武汉 430074)
[2](华中科技大学计算机科学与技术学院 武汉 430074)
(wangpeiqun@hust.edu.cn)

Study and Implementation of Frequent Sequences Mining Based Prefetching Algorithm

Wang Fang[1,2], Wang Peiqun[2], and Zhu Chunjie[1]
[1](*Wuhan National Laboratory for Optoetectronics* (*Huazhong University of Science and Technology*), *Wuhan* 430074)
[2](*School of Computer Science and Technology*, *Huazhong University of Science and Technology*, *Wuhan* 430074)

Abstract Prefetching technology is widely used as an efficient means to improve the performance of storage systems. However, traditional prefetching algorithms are mostly based on detecting sequential access features, which makes them hard to work in the environment with less or no sequential access features. What's worse, the storage system may even suffer from negative effects with poor prefetching accuracy. Whereas the proposed prefetching algorithm based on frequent sequences mining can make some contributions to the storage system in such environment by analyzing the behavior of the data accessing to find the potential rules. Meanwhile, in some application scenarios where the cache capacity may be limited, such as the embedded system, the proposed prefetching algorithm improves the prefetching accuracy to avoid some adverse impacts which may be caused by prefetching. The new proposed prefetching algorithm is based on the frequent sequences mining technology, and the prefetching rules derived from the mined frequent sequences are organized in a Trie tree. To improve the accuracy of the prefetching, the multistep matching technology and the subtree partitioning technology are introduced, which can subtly control the using of prefetching rules, so that the prefetching algorithm with relatively high prefetching accuracy can efficiently improve the performance of the storage system.

Key words frequent sequences mining; prefetching algorithm; Trie tree; multistep matching; subtree partitioning

摘 要 预取作为一种提升存储系统性能的有效手段被广泛使用,然而传统的预取算法大多基于顺序性访问特征的探测,这使得它们在非顺序数据访问环境下很难奏效,甚至可能因为预取准确率较低而对存储系统的性能带来负面影响.而基于频繁序列挖掘的预取算法则能够通过分析数据的访问行为找出潜在规律,从而能在非顺序访问模式下也取得一定的性能提升.同时,为了应对某些缓存受限的应用场景,如嵌入式系统,预取算法通过提高分析的准确率减少预取可能对缓存带来的不利影响.新提出的预取算法基于频繁序列挖掘技术,并使用字典树组织预取规则,通过多步匹配和子树分割技术精细地控制规则的使用,提升预取的准确率,从而使得预取算法能够有效提升存储系统的性能.

收稿日期:2014-09-19;**修回日期**:2015-05-14
基金项目:国家"八六三"高技术研究发展计划基金项目(2013AA013203);华中科技大学自主创新研究基金项目(2014QN010)
This work was supported by the National High Technology Research and Development Program of China (863 Program) (2013AA013203) and the Innovation Research Foundation of Huazhong University of Science and Technology (2014QN010).

关键词 频繁序列挖掘;预取算法;字典树;多步匹配;子树分割

中图法分类号 TP302

频繁序列是指在一组有序的数据列组成的数据集合中,出现次数不小于阈值的序列.频繁序列挖掘算法属于数据挖掘的一个分支,它于 1993 年由 Agrawal 等人[1]提出,至今已经有 20 多年历史.期间出现了各种各样的频繁序列挖掘算法,具有代表性的有 Apriori[2],PrefixSpan[3],CloSpan[4]等.

预取作为一种提高性能的有效手段已被广泛用于各种存储系统中,而通常提到的预取大部分是基于顺序访问特征,例如 Linux 内核使用一种带有预取窗口的顺序预取方法[5], DiskSeen[6]采用一种基于历史访问信息分析的顺序预取方法,它们的时间和空间开销小,很多情况下对性能的提升也非常明显.然而顺序预取并不适用所有情形,特别是当数据布局是非顺序时,例如文件系统中元数据信息的访问、数据库文件内部的索引结构等.这种非顺序的频繁访问模式在存储系统中是比较常见的,于是出现了针对频繁序列访问模式的预取算法,最有代表性的是 C-Miner[7]和 C-Miner*[8],它们通过频繁序列挖掘算法生成规则集,然后根据规则判断需要预取的数据块,但是它们挖掘过程中的时间和空间开销比较大,而且对规则的利用不够准确高效.本文提出的 Trie 预取算法针对频繁序列访问模式,改进了频繁序列挖掘算法,并采用字典树组织规则集,通过在树中匹配序列判断预取的数据块,预取的准确度更高.

1 算法架构

如图 1 所示,Trie 预取算法包括 3 部分,即缓存管理模块,频繁序列挖掘模块和预取模块.

缓存管理模块用于缓存块设备上的部分数据,如果上层用户请求的数据在缓存管理模块中,即缓存命中,则直接将缓存管理模块的数据返回给上层用户;如果不在,即缓存不命中,则需要向底层块设备发出读请求,并将取到的数据拷贝一份放入缓存模块中,同时将数据返回给上层用户.缓存模块用 LRU(least recently used)算法管理.

频繁序列挖掘模块收集到达缓存的用户请求元数据信息,并把它作为样本序列,当样本序列达到预先设定的规模时,执行频繁序列挖掘算法,输出频繁序列数据集,即关联规则.

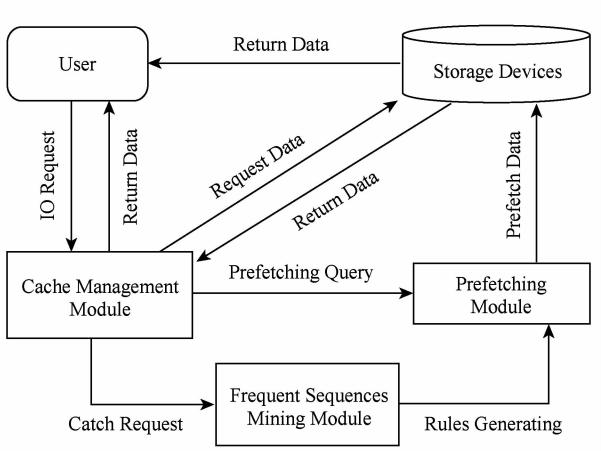


Fig. 1 Algorithm architecture.

图 1 算法架构

预取模块产生预取请求.当用户请求在缓存模块不命中时,预取模块会搜索关联规则,若找到与该请求相关的关联规则,则根据这些关联规则产生预取请求.

2 频繁序列挖掘算法

2.1 请求序列预处理

存储系统中用户请求序列是单个长序列,无法直接适用频繁序列挖掘算法,通常的做法是把这个长序列分割成多个等长的短序列.算法参考 C-Miner 的预处理方法,采用非重合式等长分割.同时,短序列的长度根据经验预先设置一个合适的值,因为如果短序列的长度太长会导致挖掘算法的开销剧增,长度太短则可能丢失大量的频繁序列信息.

2.2 频繁序列挖掘

频繁序列挖掘算法的核心数据结构包括一棵字典树和一个散列表,其中字典树用于保存所有的频繁序列信息,散列表用于临时保存已处理的结点的信息.算法开始时,字典树只有一个空的根结点指针 $head$,散列表为空.为了提高挖掘效率,算法一方面根据频繁条目集合对初始的数据集进行修剪,从而大大减小后续操作中数据集的大小;另一方面,利用散列表快速判断等价结点,从而减小递归深度.具体处理步骤描述如下:

算法 1. 频繁序列挖掘算法.

输入:字典树根结点 $head$,初始的数据集 S,筛选的阈值 min_sup;

输出:以字典树组织的频繁序列集.

步骤 1. 对初始数据集 S 进行统计,得到所有长度为 1 的频繁条目,组成频繁条目集合.如果此集合为空,则整个算法结束;否则创建 *head* 结点,将集合中的条目存放在 *head* 结点中,将 *head* 结点作为当前结点,并进入步骤 2.

步骤 2. 根据步骤 1 中得到的频繁条目集合对初始的数据集 S 进行修剪,除去 S 中的非频繁条目,得到精简的数据集 S',将 S'作为当前数据集,并进入步骤 3.

步骤 3. 根据当前数据集的大小快速判断当前结点在散列表中是否存在等价的结点.若存在等价的结点,则算法对当前结点的递归调用结束,并进入步骤 5,否则进入步骤 4.

步骤 4. 取当前结点的 1 个条目生成相应的数据集,如果该条目的后缀数据集不为空,则为此条目创建 1 个新结点,并把新结点作为当前结点,跳入步骤 3,否则继续对当前结点的下 1 个条目作步骤 4 的操作;如果当前结点的条目全部处理完,则进入步骤 5.

步骤 5. 判断当前结点在字典树中是否存在未遍历的兄弟结点,若存在,则把兄弟结点作为当前结点,并转入步骤 3,否则将当前结点的父结点作为当前结点;如果当前结点是 *head*,则整个算法结束,否则重复步骤 5.

挖掘算法结束后,规则树中有很多并不是闭合的频繁序列,需要对它们进行删除.由于散列表中保存的结点在具有等价后缀数据库的结点中,均是最长的序列,因此对散列表中所有的结点进行扫描,查找到所有叶子结点,将它们标记为 FLAG_CLOSET,并将它们的所有祖先结点标记为 FLAG_INUSE.散列表扫描结束后,对之前生成的频繁序列集进行整理,仅保留标志位 FLAG_INUSE 的结点,新生成的树则为最终的频繁序列规则字典树.如果把挖掘的频繁序列的长度限制成 1 个常量,则此算法在最坏情况下与 CloSpan 算法的时间复杂度相当,即为 $O(n^2)$[9],同时,由于在 CloSpan 算法的基础上做了上述优化,所以新算法的计算开销更小.

3　基于字典树匹配的预取算法

为了能够有效存储和使用挖掘算法输出的关联规则,预取算法采用字典树组织规则.任何一条频繁序列都是作为一个整体出现的,一旦检测到某个频繁序列的第 1 个请求时,预取算法就将该频繁序列中的所有其他请求预取到内存.

图 2 描述了一棵简单的频繁序列规则树,它包含 5 条频繁序列{$abcd$,$aefg$,$aefh$,b,$cfai$}.其中,所有规则的前缀都分布于规则树的第 1 层,那么当请求到来时仅需要从规则树的第 1 层查找访问请求是否存在.如果存在,则表示在接下来的访问可能对应于请求子树中的某一条频繁序列,需要将这棵树的子树预取到内存中.例如,如果探测到请求条目 a,则将 a 的子树的所有条目{b,c,d,e,f,g,h}预取到内存中.虽然在频繁序列 $cfai$ 中,条目 a 与条目 i 也存在关联关系,但是由于没有探测到条目 c 的到来,i 不会被预取.采用这样的方式,可以大量减少无关数据的预取,从而减少预取的失效率.

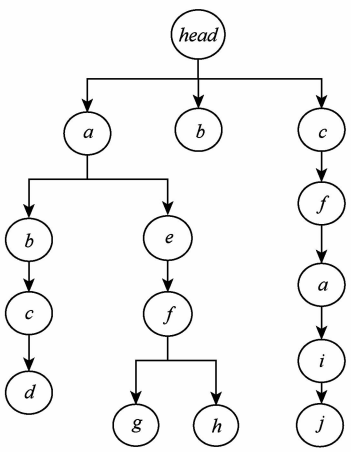


Fig. 2　Prefetching tree of rules.

图 2　规则预取树

然而,当字典树具有较多分支且树的深度较深时,上述预取方式仍然会导致过度预取,特别是当系统缓存比较小时,这种预取方式可能给系统带来更大的负担.因此,为了进一步提高预取的精度,提出了多步匹配和子树分割的方法.

3.1　多步匹配

上述提到的基于字典树匹配的预取算法仅仅通过匹配频繁序列前缀的首个条目,就预取该条目的所有子树上的条目.如果某条目有多棵子树(如图 2 中的条目 a 有 2 棵子树),那么预取时将会把这 2 棵子树上的所有条目都预取到内存中,但实际上可能只有 1 棵子树将被访问,特别是当子树很多时,这种预取的正确率更低,带来的开销也更大.

多步匹配能够较好地解决这个问题,其基本思路是当一个条目有多棵子树时,先把此条目保存在一个队列结构中,当该条目的某个子树的根条目被

访问时,则将这个子树的根条目也加入到此队列中,直到一个频繁序列的前 n 个条目都出现时,才把此频繁序列的后续条目全部预取到内存.具体的做法是在缓存中创建一个队列,每当一个新的请求到来时,通过对比队列中已经匹配的前缀,决定是否进行匹配层级增加或者进行预取操作.

图 3 模拟一个正在运行的多步预取队列,设定的预取步长为 4.在条目 f 到达之前,匹配队列中维护着若干个已经匹配的条目,队列的每个结点保存一个条目的数据块信息及当前已经匹配的层级,同时还有一个结构指向规则树中已经匹配到的结点.当 f 到达时,从 LRU 队列的顶端查找是否所指向的树结点的子结点中存在 f,如果存在,则将结构放到 LRU 的顶端,并且匹配层级自增 1.

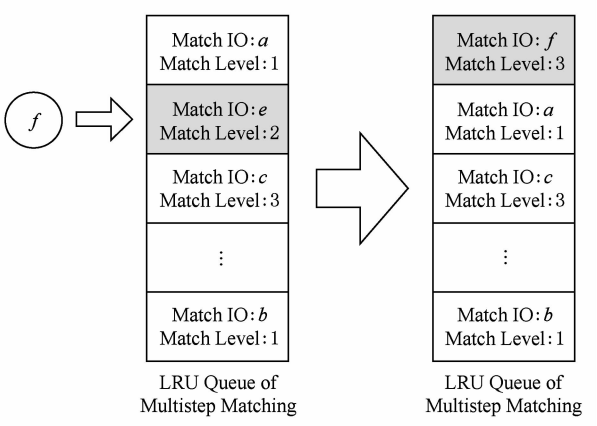


Fig. 3 Prefetching cache pools based multistep matching.

图 3 多步匹配预取缓存池

例如,从图 2 可以看出,条目 e 所在的结点的子树中有 f,所以当 f 到达时,用 f 替代 e,并且将此结点的匹配层级变为 3.如果经过变化的匹配层级达到多步预取的步长 4,则进行预取操作,并且将该结点删除;否则没有达到预取的要求,继续进行后面的操作.

3.2 子树分割

多步匹配策略提高了预取的准确度,而且步长越长,预取的准确度也越高,然而如果步长设置过长,则预取的数据会大大减小,使得预取的效果被削弱,而且匹配带来的开销也会增大,故在使用时需要设置一个比较合适的步长.但是在某些情况下,超过设定步长的子树仍然有很多分支,此时多步匹配对精度的提高就比较有限.为了应对这种情况,提出子树分割的策略,即将规则生成树进行预处理,使得规则树的高度维持在一定范围内,而分割后靠近叶子结点的部分挂载到另外一个线性表中,只有在实际访问到某棵子树的父结点时,才将其预取到缓存中.

图 4 描述了对前面描述的例子中的子树进行分割的过程,设定子树分割的深度为 2,那么将初始的规则树深度为 2 的子树全部切割,挂载到一个数组下面.当条目 a 的请求到来时,根据规则树将 b 和 e 预取到内存中,但却不预取它们各自的子树 cd 和 fgh.而且随着请求的继续,若某一时刻,条目 e 在内存中命中,并检测到该条目 e 是某一棵子树的根结点,那么通过其 ID 查找到相应的子树,从而将子树预取.

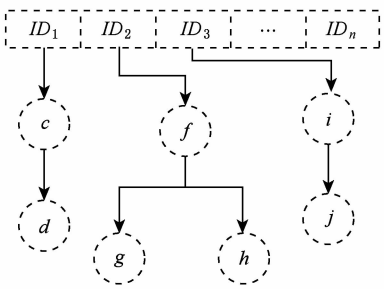


Fig. 4 Subtree partitioning.

图 4 子树分割

4 实验及结果分析

本文的负载采用惠普实验室的 HP Cello96 数据集,它是 1996 年在惠普 Cello 系统上跟踪采集的 2 级磁盘 IO 请求,而 Cello 是由一组研究者用来作仿真、编译、编辑文档和收发邮件的分时共享系统.为了验证算法,搭建了模拟实验原型系统,原型系统的架构与图 1 基本吻合.实验平台是 1 台浪潮 NF5280 服务器,具体的硬件配置如表 1 所示.原型系统运行于 64 位 Red Hat Enterprise Linux Server Release 5.3 操作系统上,它是自主实现的一个用户态仿真程序,包括缓存管理模块、频繁序列挖掘模块和预取模块,其中缓存管理模块不存储真实的数据,只存储请求的元数据信息,因为通过元数据信息就可以判断是否命中.

Table 1 Hardware Configuration of the Server

表 1 服务器硬件配置

Hardware	Model
CPU	Intel Xeon E5560
Mainboard	Supermicro X6DHE-XB
Memory	DDR ECC RG 12 GB
Disk	8×SAS Disks (15000RPM, 146 GB) configured RAID5

实验方法分 2 个阶段,第 1 阶段是频繁序列挖掘阶段,此阶段尚没有关联规则可用,所以预取模块不可用. 当关联规则树生成后,程序进入第 2 阶段,此阶段预取模块是可用的. 分段的标准是 trace 文件中前半部分的 IO 请求属于第 1 阶段,后半部分属于第 2 阶段. 这种实验方法是参考 C-Miner 中的做法,方便与之对比.

实验中关注的参数主要有:*hit_num* 为缓存命中次数,*replace_num* 为缓存替换次数,*prefetch_hit_num* 为预取命中次数,*prefetch_num* 为预取次数.

定义 1. 预取准确率=预取命中的请求数/预取的次.

图 5 和图 6 分别描述了不同算法的命中率和预取准确率的比较情况,缓存大小为 64MB. 其中 LRU 是标准的缓存替换算法;C-Miner 是目前比较好的一个频繁序列预取算法;Trie 是单步匹配的频繁序列预取算法,是本文的核心算法;Trie(3)是 3 步匹配的频繁序列预取算法,是对 Trie 的一种改进;Trie(3)&CutTree 是同时使用 3 步匹配和子树分割策略优化的频繁序列预取算法,是在 Trie(3)的基础上做进一步的优化.

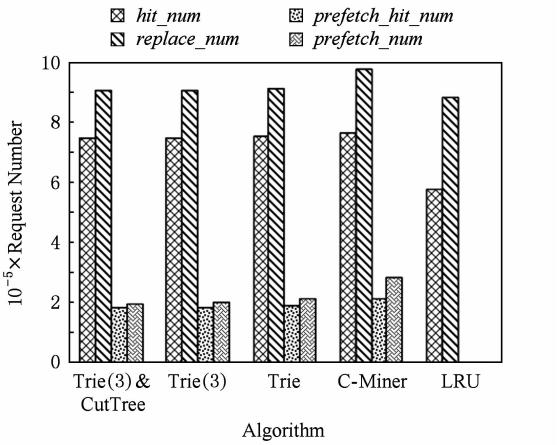


Fig. 5　Comparison of hit rate.

图 5　命中率比较

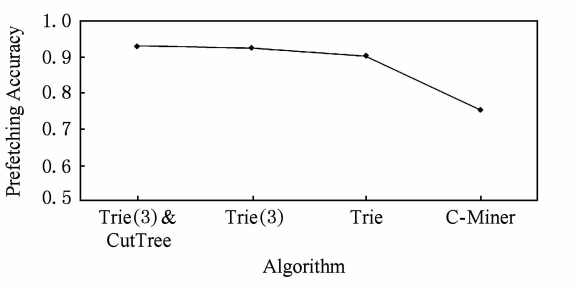


Fig. 6　Prefetching accuracy.

图 6　预取准确率

从图 5 可以看出,LRU 算法的缓存命中次数在 580 000 左右,其他 4 种算法的缓存命中次数均在 750 000 左右,即 Trie(3)&CutTree 算法的缓存命中率比 LRU 算法提高 29.3%,与 C-Miner 算法的缓存命中率基本一致,但 Trie(3)&CutTree 的预取条件更严苛,所以它的预取次数比 C-Miner 减少 40%左右,缓存替换次数比 C-Miner 减少 8%左右.

从图 6 可以看出,Trie(3)&CutTree 的预取准确率是最高的,达到 92.9%;C-Miner 的预取准确率是最低的,只有 75.4%;另外,LRU 算法不存在预取,所以不考虑它的预取准确率. 从而,根据图 5 和图 6 可以得出,Trie(3)&CutTree 在缓存命中率与 C-Miner 达到相同效果的情况下,预取的准确率更高,从而能够有效减轻磁盘负担,并降低缓存替换的频度.

5　结　　论

本文提出一种基于频繁序列挖掘的预取算法,它能够挖掘存储系统中数据块的关联性,并通过合理使用这些关联特性,提高缓存的命中率. 与通用的 C-Miner 算法相比,本文方法使用关联规则的条件更加严苛,使得它能在命中率与 C-Miner 基本保持一致的情况下,表现出更高的预取精度,这使得它在存储系统内存资源有限的情况下比 C-Miner 算法更加适用.

参　考　文　献

[1] Agrawal R, Imielinski T, Swami A. Mining association rules between sets of items in large databases [C] //Proc of the 19th ACM SIGMOD Int Conf on Management of Data. New York: ACM, 1993: 207-216

[2] Agrawal R, Srikant R. Fast algorithms for mining association rules [C] //Proc of the 20th Int Conf on Very Large Data Bases. San Francisco, CA: Morgan Kaufmann, 1994: 487-499

[3] Pei J, Han J, Mortazavi-Asl B, et al. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth [C] //Proc of the 17th Int Conf on Data Engineering. Washington, DC: IEEE Computer Society, 2001: 215-224

[4] Yan X, Han J, Afshar R. CloSpan: Mining closed sequential patterns in large datasets [C] //Proc of the 3rd SIAM Int Conf on Data Mining. Philadelphia, PA: SIAM, 2003: 166-177

[5] Wu Fengguang. Prefetching algorithm in Linux kernel [D]. Hefei: University of Science and Technology of China, 2008 (in Chinese)
(吴峰光. Linux 内核中的预取算法[D]. 合肥: 中国科学技术大学, 2008)

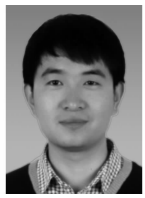
[6] Ding Xiaoning, Jiang Song, Chen Feng, et al. DiskSeen: Exploiting disk layout and access history to enhance I/O prefetch [C] //Proc of the 2007 USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2007: 261-274

[7] Li Zhenmin, Chen Zhifeng, Sudarshan M, et al. C-Miner: Mining block correlations in storage systems [C] //Proc of the 3rd USENIX Conf on File and Storage Technologies. Berkeley, CA, USA: USENIX Association, 2004: 173-186

[8] Avichai G, Dan P, Eran R, et al. Using machine learning technique to enhance the performance of automatic backup and recovery system [C] //Proc of the 3rd Annual Haifa Experimental Systems Conf. New York: ACM, 2010: 1-10

[9] Li Zhenmin, Lu Shan, Myagmar S, et al. CP-Miner: Finding copy-paste and related bugs in large-scale software code [J]. IEEE Trans on Software Engineering, 2006, 32 (3): 176-192

Wang Fang, born in 1972. Professor in the Huazhong University of Science and Technology. Member of China Computer Federation. Her main research insterests include computer architecture, computer energy, distributed file system and parallel I/O storage system.



Wang Peiqun, born in 1990. Received his master degree from the Huazhong University of Science and Technology. His main research interests include mass network storage system and parallel I/O storage system.

Zhu Chunjie, born in 1988. PhD in the Huazhong University of Science and Technology. Student member of China Computer Federation. His main research interests include mass network storage system and parallel I/O storage system.