Learned Cache

机器学习驱动下，面向真实场景的缓存设计

演讲者：尚卓燃

演讲时间：2021.10.15

PART.01
内　容　速　览

PART.02
LeCaR - LRUxLFU

PART.03
AutoCache for DFS

PART.04
LearnedCache x Databend?

LEARNED-CACHE

PART.01

内 容 速 览

ML & Cache

- 机器学习正在广泛应用，并影响到 cache, index 的设计与实现
- 基于机器学习的算法往往在真实场景有优势
- 机器学习可以用来学习 “最好” 的缓存替换策略
- 机器学习可以用来改进现有的缓存替换策略

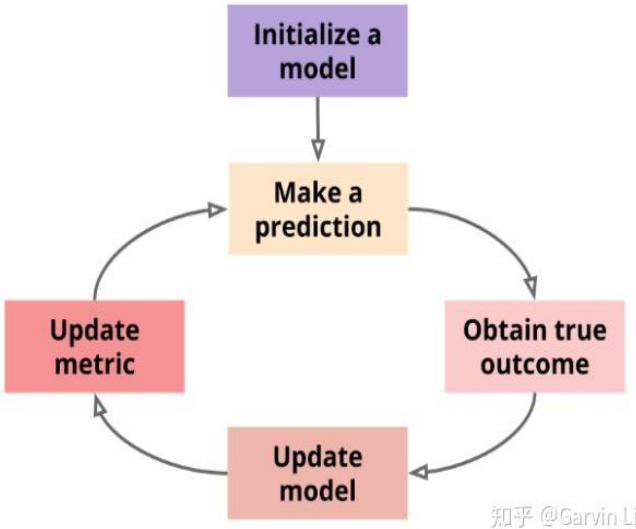
LEARNED - CACHE

PART.02

LeCaR - LRUxLFU

LeCaR

- 探索路径：单一(LRU) -> “适应”(ARC) -> 机器学习(NN) -> 在线学习(LeCaR)
- 遗憾最小化 + 强化在线学习 + 一组几乎正交的缓存替换策略(LRU + LFU)
- 极简主义(相对专家策略集而言)，学习系统每个状态下的最优概率分布
- 命中率 -> 遗憾函数，缓存替换问题 -> 遗憾最小化的在线学习问题



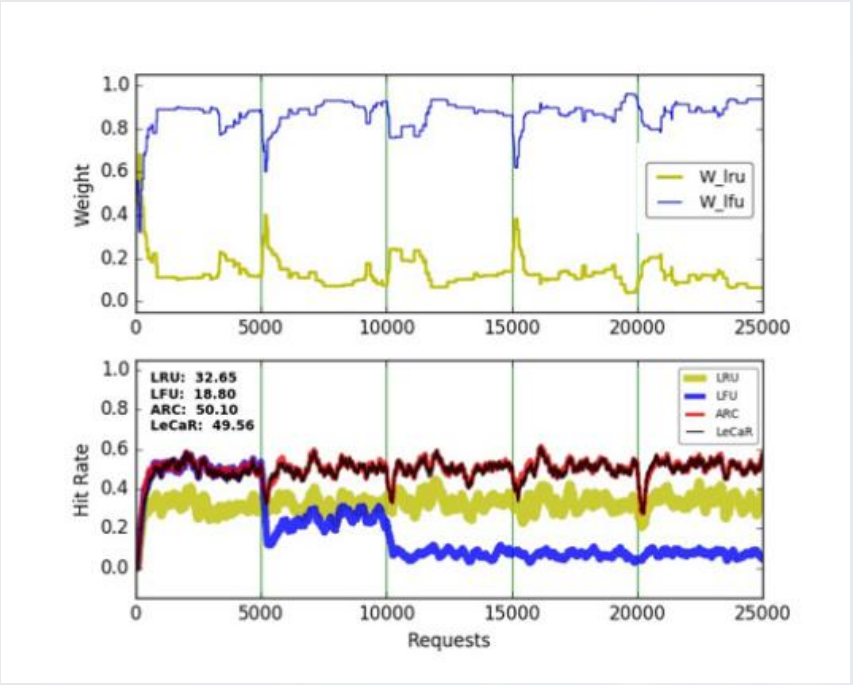
Algorithm 2: UPDATEWEIGHT(q, λ, d)

Input: page q, learning rate λ, discount rate d
$t :=$ time spent by page q in History
$r := d^t$
if q is in H_{LRU} **then**
  $w_{\text{LFU}} := w_{\text{LFU}} * e^{\lambda * r}$ // increase w_{LFU}
else if q is in H_{LFU} **then**
  $w_{\text{LRU}} := w_{\text{LRU}} * e^{\lambda * r}$ // increase w_{LRU}
$w_{\text{LRU}} := w_{\text{LRU}} / (w_{\text{LRU}} + w_{\text{LFU}})$ // normalize
$w_{\text{LFU}} := 1 - w_{\text{LRU}}$

Algorithm 1: LeCaR(LRU,LFU)

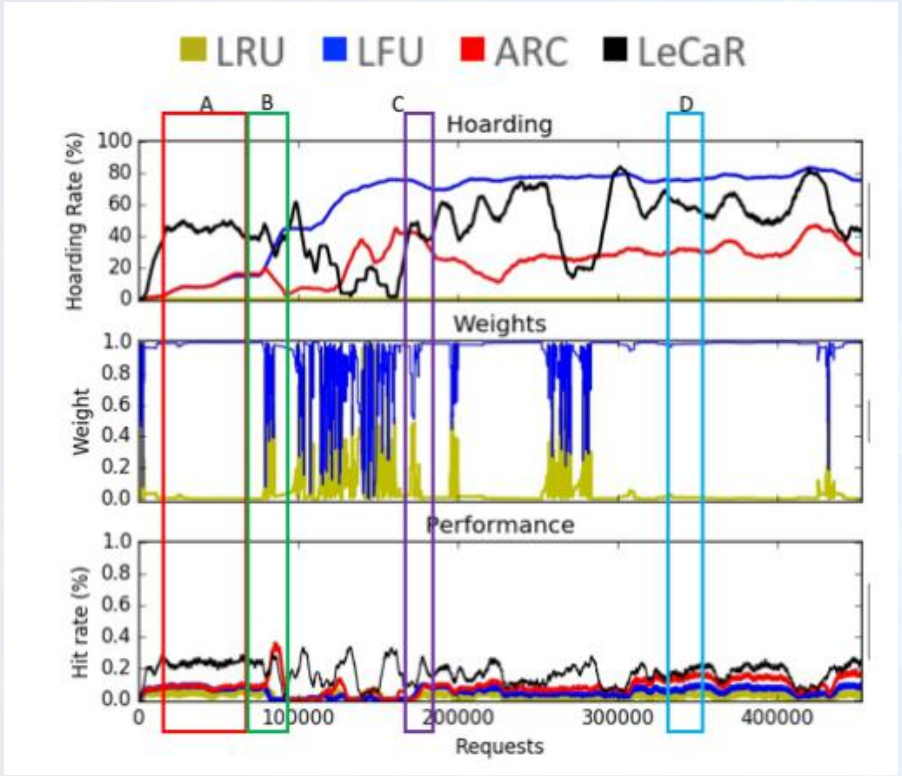
Input: requested page q
if q is in C **then**
  C.UPDATEDATASTRUCTURE(q)
else
  if q is in H_{LRU} **then**
    H_{LRU}.DELETE(q)
  else if q is in H_{LFU} **then**
    H_{LFU}.DELETE(q)
  UPDATEWEIGHT(q, λ, d)
  if (Cache C is full) **then**
    action = (LRU, LFU) w/ prob $(w_{\text{LRU}}, w_{\text{LFU}})$
    if (*action* == LRU) **then**
      if H_{LRU} is full **then**
        H_{LRU}.DELETE(LRU(H_{LRU}))
      H_{LRU}.ADD(LRU(C))
      C.DELETE(LRU(C))
    else
      if H_{LFU} is full **then**
        H_{LFU}.DELETE(LRU(H_{LFU}))
      H_{LFU}.ADD(LFU(C))
      C.DELETE(LFU(C))
  C.ADD(q)

LEARNED
- Cache
PART.02
LeCaR - LRU/LFU



当缓存足够大时，并不能体现出缓存策略的优势，因为工作集已经装入缓存。而当缓存减少（或者工作量扩大后），缓存之间的差异就能够更明显体现。

LeCaR 的缓存所需空间是 ARC 的 2 - 3 倍。当缓存是工作量的 1/1000 时，LeCaR 相较 ARC，在 IO 方面提高 18 倍。



LEARNED - CACHE

PART.03

AutoCache for DFS

AutoCache

- 数据密集型工作的大部分时间花费在 IO 上，缓存(LRU)无法适应负载
- 考虑数据访问模式对系统的影响，以提高集群效率和负载性能
- 机器学习跟踪和预测文件访问模式，轻量级梯度增强树学习如何根据工作量访问文件，并使用生成的模型决定元素的插入和逐出
- 自动管理文件缓存，并动态移动文件（从/到缓存）

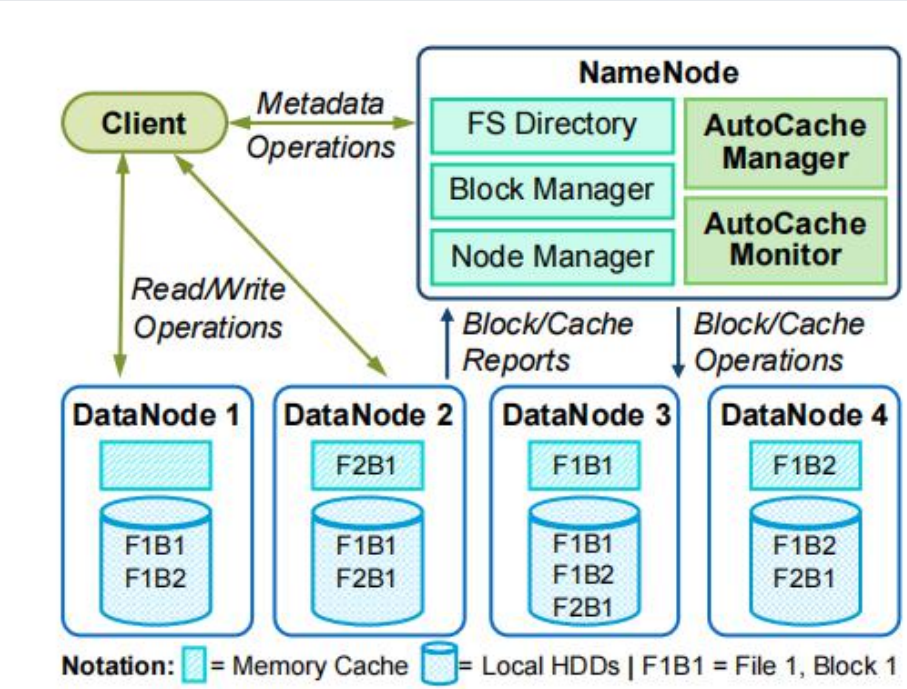


Fig. 1: HDFS Architecture with AutoCache

AutoCache----------Manager & Monitor

Manager 负责根据可插入的存入和逐出策略的决定来协调文件的缓存和解压。
决定基于系统维护的文件和节点统计数据以及文件创建、访问、修改或删除后收到的通知产生。
Monitor 负责处理来自 Manager 的缓存和解压请求，以及监视分布式缓存的总体状态。
在文件写入/读取的同时启用缓存，以及直接将文件块存储在内存中提高效率。

NameNodes: 1. 文件系统目录：文件层级组织和操作 2. 块管理：维护文件块到节点的映射
3. 节点管理：网络拓扑和节点统计信息
DataNodes: 1. 在本地磁盘或内存中存储/管理文件 2. 为客户端的文件请求提供服务
3. 根据 NameNodes 的指示管理块的创建/删除等
Clients: 为所有典型的文件系统操作（如创建目录或读 /写文件）公开 API。

1数据准备

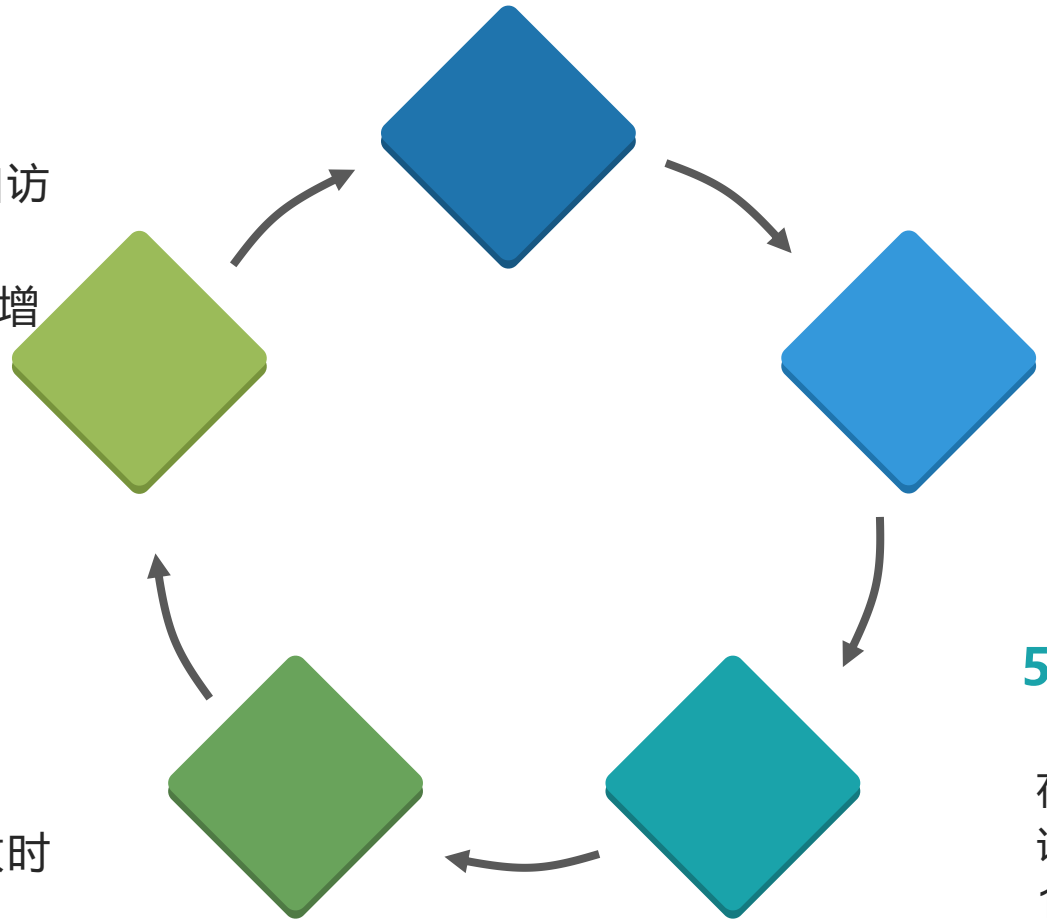
1 文件的文件大小、创建时间和访问时间
2. 二分类，将时间离散为固定/增长的间隔

2学习模式

XGBoost
1.准确
2.开销小

3访问预测

1 将参考点设置为当前时间；
2 使用固定或增长间隔方法离散时间
3 根据文件大 小、创建时间和访问时间创建特征



4 逐出策略

考虑到等待IO会造成更大的延迟，所以设定上下阈值，一旦超过上界就开始逐出，触及下界则停止

5 存入策略

存入成本同样很高（涉及访问大量数据）
1. 与最近文件相关 2. 根据权重考虑（无则缓存，高于阈值缓存，高于要执行文件缓存和将来高概率访问缓存）

LEARNED-CACHE

PART.04

LearnedCache x Databend?

参 考 资 料

- Driving Cache Replacement with ML-based LeCaR
https://www.usenix.org/conference/hotstorage18/presentation/vietri
- AutoCache: Employing Machine Learning to Automate Caching in Distributed File Systems
https://www.researchgate.net/publication/334152920_AutoCache_Employing_Machine_Learning_to_Automate_Caching_in_Distributed_File_Systems