Computer Science and Application 计算机科学与应用, 2023, 13(3), 358-368
Published Online March 2023 in Hans. https://www.hanspub.org/journal/csa
https://doi.org/10.12677/csa.2023.133034

Hans汉斯

基于HBase的数据高效读技术研究

闵继勇，史爱武*，武　俊，田贞才

武汉纺织大学计算机与人工智能学院，湖北　武汉

收稿日期：2023年2月16日；录用日期：2023年3月15日；发布日期：2023年3月23日

摘　要

在大数据时代，关系型数据库面临着海量数据存储的挑战。HBase是一种基于列存储的NoSQL数据库，广泛应用于大数据存储。HBase在数据的检索方面仍然存在着不足之处，本文对HBase的数据检索技术进行分析和研究，针对目前存在的问题提出了相应的改进和优化。针对HBase在查询数据时需要访问磁盘，查询速度慢的问题，本文提出使用Redis索引HBase的热点数据，并综合考虑数据的查询频率、更新频率和历史积热对缓存的影响，设计了一种基于数据查询频率和更新频率的热值缓存驱逐策略，提高了Redis的缓存命中率。针对HBase在检索非行键字段时需要全表扫描，检索效率低的问题，本文提出了为非行键字段建立二级索引的策略，设计了一种基于协处理器和Redis的二级索引方案。实验结果表明改进后的缓存驱逐策略的命中率高于LRU策略，在查询模块引入Redis缓存热点数据并且为非行键字段建立二级索引后，改进后的查询模块的数据检索性能提升显著，极大地提高了查询速度。

关键词

HBase，Redis，缓存，二级索引

Research on Efficient Data Reading Technology Based on HBase

Jiyong Min, Aiwu Shi*, Jun Wu, Zhencai Tian

College of Computer and Artificial Intelligence, Wuhan Textile University, Wuhan Hubei

Received: Feb. 16th, 2023; accepted: Mar. 15th, 2023; published: Mar. 23rd, 2023

Abstract

In the era of big data, relational databases face the challenge of massive data storage. HBase is a NoSQL database based on column storage, which is widely used in big data storage. HBase still has

*通讯作者。

文章引用: 闵继勇, 史爱武, 武俊, 田贞才. 基于 HBase 的数据高效读技术研究[J]. 计算机科学与应用, 2023, 13(3): 358-368. DOI: 10.12677/csa.2023.133034

shortcomings in data retrieval. This paper analyzes and studies the data retrieval technology of HBase, and puts forward corresponding improvements and optimization for the existing problems. In view of the problem that HBase needs to access the disk when querying data and the query speed is slow, this paper proposes to use Redis to index the hot data of HBase, and comprehensively considers the impact of data query frequency, update frequency and historical heat accumulation on the cache, and designs a calorific cache eviction strategy based on data query frequency and update frequency, which improves Redis cache hit rate. Aiming at the problem that HBase needs full table scanning when retrieving non-row key fields, and the retrieval efficiency is low, this paper proposes a strategy of establishing secondary index for non-row key fields, and designs a secondary index scheme based on coprocessor and Redis. The experimental results show that the hit rate of the improved cache eviction strategy is higher than that of the LRU strategy. After the Redis cache hot data is introduced into the query module and the secondary index is established for non-row key fields, the data retrieval performance of the improved query module is significantly improved, and the query speed is greatly improved.

Keywords

HBase, Redis, Cache, Secondary Index

1. 引言

随着互联网在日常生活中地位的不断提高，业务系统和各个网站所需处理的业务量和数据量快速增长，例如短视频软件或者社交软件需要为用户存储和计算大量数据。由于人们对计算资源利用效率、存储与处理数据能力以及资源集中化整合日益迫切的需求，云计算与大数据应运而生[1] [2]。随着网络技术的发展，特别是移动通信和社交网络的快速发展，需要处理 TB 级别甚至 PB 级别的数据。常用的关系型数据库没有办法处理这样的海量数据。海量数据的检索也对传统的关系型数据库[3]提出了严峻的挑战。由于传统数据库无法满足高并发请求和高存储空间，非关系型数据库应运而生，如 MongoDB，Cassandra 等，其中 HBase 应用最为广泛。HBase 是一种分布式非关系型面向列的开源数据库，广泛应用于大数据的存储。HBase 使用 HDFS 作为文件数据存储系统，使用 MapReduce 处理海量数据，使用 Zookeeper 作为协作服务[4]。HBase [5]已经应用于许多大型网络，如 Facebook、淘宝、Twitter。HBase 源自谷歌提出的 Bigtable，它的底层利用 HDFS 来存储数据，HDFS 是 Hadoop 生态系统的一部分[6]。HBase 不断地对磁盘进行读写请求，这使得其查询性能受到硬件的限制[7]。利用内存缓存技术缓存 HBase 的热点数据已成为提高其查询性能的常用方法，HBase 的内存缓存技术采用了第三方工具(如 Redis、Memcache 等)和一组辅助工具[8]。按照 LSM-Tree 模型编写缓存辅助工具的方式会降低 HBase 的写入性能。第三方工具提供的缓存驱逐策略很少，而且它们无法支持缓存更新。目前，面向 HBase 的数据缓存技术主要从硬件和软件两个方面进行考虑。TBF [9]提出了一种基于固态硬盘的缓存驱逐策略。该算法结合了 Clock 和 Bloom 过滤器，减少了元数据的空间开销。microsoft [10]使用指数平滑方法来区分记录的数据查询的频率以及冷数据和热数据之间的差异，从而动态地调整冷数据和热数据的边界。然而，该方法主要用于日志数据查询的分析。Zhang 等人[11]提出了一种使用 Redis 指数平滑来预测记录查询频率的方法。该方法比 LRU 算法有更高的命中率，但该方法直接使用 Redis 作为持久数据库，不适合大数据处理。HiBase [12]

使用 Redis 索引 HBase 的热点数据，并采用基于热积累的缓存驱逐策略。但是，它没有考虑数据更新频率对缓存的影响。Zhai 等人在[13]中引入了 TwemProxy，这是一种 Redis 中间件代理服务，可以提高 Redis 的性能。然而，该解决方案必须维护一组额外的 TwemProxy 集群，这会带来额外的开销。Li 等人[14]提出了一种基于查询和更新频率的热累积 Redis 缓存驱逐算法。该算法考虑了数据更新的频率，但忽略了历史数据的影响。HBase 只能通过行键检索数据，在未命中索引时，HBase 会全表扫描，这样效率低下，所以在查询非行键字段时，为非行键字段建立二级索引非常重要，能极大的提高数据查询的效率。

HBase 在数据高效读方面还存在需要改进的地方，主要体现在以下两点：HBase 在查询数据时需要访问磁盘，查询速度慢，HBase 在检索非行键字段时需要全表扫描，检索效率低。因此，本文主要针对以上两点，分析并研究了 HBase 的数据高效读技术，针对不足的地方进行了优化和改进，具体的优化和改进体现在本文提出使用 Redis 索引 HBase 的热点数据，并综合考虑数据的查询频率、更新频率和历史积热对缓存的影响，设计了一种基于数据查询频率和更新频率的热值缓存驱逐策略，提高了 Redis 的缓存命中率，减少了磁盘访问开销和访问时间，提出了为非行键字段建立二级索引的策略，设计了一种基于协处理器和 Redis 的二级索引方案。这些优化和改进对提高 HBase 的性能有着非常重要的意义。

2. HBase 数据检索的研究与分析

2.1. 数据检索研究

HBase 的默认检索方式大致分为以下三种：

1) 通过指定的行键进行查询，返回指定对应的数据。

2) 通过设置起始行键和结束行键的边界值，利用 scan 语句查询数据。

3) 通过扫描整个表获取相应的全部数据。

在数据快速检索过程中，Redis 常被用作缓存，Redis 有持久化策略，缓存在 Redis 中的数据可以周期性地存储到硬盘上，以解决由于意外断电或节点停机导致的数据丢失问题。同时，HBase 是一种 NoSQL 数据库，实现了数据索引的持久化，因此，两种技术的结合具有很好的应用价值。每个对 HBase 的读写请求都涉及磁盘的访问，这限制了数据读写的性能。Redis 可以将经常访问的数据存储到 Redis 缓存服务器，并将数据持久化到 HBase 数据库，可以提升数据检索效率。数据的检索分为行键字段和非行键字段的检索。

1) 行键字段的查询

为了加快数据的检索速度，可以采用 Redis 和 HBase 结合来存储和查询数据，Redis 实现了热数据的缓存，并根据缓存驱逐策略对缓存数据进行更新，从而提高检索效率。

Redis 的默认缓存驱逐策略是 noeviction，即当内存使用超过配置的时候会返回错误，不会驱逐任何键。显然不进行缓存驱逐的话，Redis 内存到达上限后，直接返回错误，不能实现热数据的缓存。Redis 有 8 种缓存驱逐策略，其中 LRU 策略是最常用的。LRU 策略的思想是将缓存中最久未使用的数据驱逐出缓存，逻辑上就是先进入缓存的数据先离开。然而，它没有考虑数据查询的频率，如果现在缓存里有一个经常被访问的页面，它的访问频率很高，所以不希望它被逐出缓存。但此时有一个 scan 操作访问了非常多的数据，依据 LRU 的原理，经常被访问的那个数据会被挤出缓存，即使新加入的数据仅仅被访问了一次。LFU 策略的思想是将缓存中最不常用的数据驱逐出缓存。

2) 非行键字段的查询

行键可以满足大多数查询的需求，但是在查询非行键字段时，HBase 只能进行全表扫描来检索出结果，这显然不能满足实际需求。

2.2. 问题分析

由于采用默认的缓存驱逐策略和非行键字段的存在，数据检索存在如下两个问题：

1) 查询行键字段时，提出了利用 Redis 来缓存热数据，即经常被查询访问的数据，但是在默认的缓存驱逐算法方面，没有同时考虑到查询频率、更新频率和历史积热对数据热值计算的影响。

2) 查询非行键字段时，必须扫描全表进行查询，检索效率非常低。

3. 改进优化策略与算法

3.1. 数据缓存策略

Redis 是一个完全开源的高性能键值数据库，它的数据保存在内存中。此外，Redis 支持丰富的数据结构，提供了极高的读写性能，每秒读取高达 10 万次，每秒写入高达 8 万次，因此在执行行键查询和非行键查询时提供了很高的效率和性能。Redis 根据缓存驱逐策略对缓存数据进行更新，从而加快了检索速度。通过研究并分析 LRU 和 LFU 缓存驱逐策略，本文提出，查询频率越高的数据在未来被查询的可能性越大，更新频率越高的数据被查询的可能性越小。根据查询和更新频率这两个因素，设计了一种基于时间平滑法计算热值的缓存逐出算法。本文设计了以下方程式来计算记录的热值：

$$heatValue_n = w \times \frac{queryf}{updatef} + (1-w) \times heatValue_{n-1}$$

$$heatValue_n = w \times \frac{\frac{queryCount}{T}}{\frac{updateCount}{T}} + (1-w) \times heatValue_{n-1}$$

$$heatValue_n = w \times \frac{queryCount}{updateCount} + (1-w) \times heatValue_{n-1}$$

queryCount 是一个周期内的访问次数，*updateCount* 是一个周期内的更新次数，*T* 是热值计算周期，*queryf* 是查询频率，*updatef* 是更新频率，历史值 $heatValue_{n-1}$ 反映历史记录的热值。参数 *w* 是衰减系数，$0 < w < 1$，用于确定热值中累积热量和历史热量的权重。最近访问的权重越大，历史查询记录对数据热值的影响越小，反之亦然。缓存中数据的历史热量以 $1 - w$ 的速率衰减，周期越早的数据的的累积热量将经历更多的衰减。因此，早期累积热量对数据热量的影响逐渐减小。数据缓存组件的思想是计算查询和更新缓存数据的次数，并根据以上公式计算每条记录的热值，将其存储在有序的 ZSet 热值集中。在数据缓存开始时，缓存是空的，数据存储到 HBase 和 Redis。当缓存的数据量达到预设阈值时，将调用缓存逐出算法，该算法将删除热量值最低的 *n* 条记录。缓存逐出的基本步骤如算法所示：

缓存驱逐算法：

输入：(key,value)类型的字符串数据

输出：操作结果(true/false)

1) 首先根据以上公式计算每条记录的热值，并将热值集存放到 Zset 有序集合里面。

2) 查看操作方式是更新操作还是查询操作，如果是查询操作，则将访问次数 *queryCount* + 1，如果是更新操作，则将更新次数 *updateCount* + 1。

3) 重新计算记录的热值并更新 ZSet 热值集。

4) 如果 ZSet 集合内存被热值集占满，则执行 delete 方法删除 *n* 个最低热值的数据。

5) 返回结果。

3.2. 查询模块

3.2.1. 查询行键字段

查询行键字段时，可以直接命中索引，通过行键索引来检索数据，查询模块是在上述 HBase 和 Redis 改进的基础上设计的，以加快数据的查询检索速度。结合 HBase 和 Redis 进行批量数据检索时，缓存的数据从 Redis 中获取，如果缓存的数据不存在，则从 HBase 中获取持久化数据。首先在 Redis 中根据行键匹配 key 进行搜索，当数据在 Redis 中被命中时，从 Redis 的缓存中返回与行键匹配的数据记录，查询任务将在不访问磁盘的情况下结束，这大大减少了磁盘开销和查询时间。如果 Redis 中没有，则在 HBase 中通过行键进行检索。改进后的方案支持批量数据查询。基本步骤如下：

1) 批量读取查询条件，使用算法 1 生成行键。

2) 遍历行键并向 Redis 发起查询请求。如果 Redis 中存在与行键对应的 key，则从 Redis 的缓存中返回与行键匹配的数据记录值，并结束查询。

3) 否则，将向 HBase 的指定区域发起请求。如果 HBase 中存在查询数据记录，则将查询记录写入 Redis 并返回记录。查询结束。否则，返回一个标志，表示查询记录不存在。

3.2.2. 查询非行键字段

查询非行键字段时，为了避免全表扫描，引入了二级索引的方案，即基于协处理器和 Redis 的 HBase 二级索引方案。协处理器拥有直接在区域服务器上执行自定义代码的机制。协处理器类似于关系型数据库中的触发器，它在特定事件(比如 Get 或 Put)发生之前或之后执行代码，协处理器在特定事件发生之前或之后触发。协处理器可以在执行 Put 操作时更新索引。索引的构造由 MyCorprocessor 类完成，它继承 BaseRegionObserver 类。重写这个类中的钩子方法，以确保在 Put 操作之后，相关的索引部分会更新。该方案通过协处理器实现 HBase 和 Redis 中二级索引的快速创建和自动更新。检索时快速获取相应的行键，并根据行键查询数据表中相应的数据。前者存储数据表的索引，后者存储数据表中最新数据的索引，提高数据检索的实时性。Redis 通过协处理器实现了索引的自动构建和更新，Redis 只在数据表中存储最新数据的索引，并及时删除旧的数据索引。例如，最近一周数据的二级索引信息，以减少磁盘访问开销并改善对新数据的实时访问。在检索过程中，通过 Redis 索引表或 HBase 索引表获取符合过滤条件的行键，然后根据行键快速查询 HBase 数据表中对应的行键数据，提高多条件非行键字段的检索效率和性能。

当客户端发出插入、删除或更新数据的请求时，数据表的区域将添加这条记录，协处理器将根据建立的策略捕获相应的插入、删除或更新操作。根据制定的策略对索引表插入、删除或更新对应的索引项，例如：插入、删除或更新数据为新的数据，协处理器同时对 Redis 二级索引表和 HBase 二级索引表插入、删除或更新对应的索引项，并扫描 Redis 二级索引表中的数据，将过期的索引数据删除。当删除或更新的数据不是最新数据时，协处理器只删除或更新 HBase 二级索引表中对应的索引项。

在本文设计的二级索引方案中，为了充分利用 HBase 数据库的存储特性，尽可能减少 HBase 二级索引表的数量，提高索引性能，将所有索引表融合为一个索引表，并通过索引识别对索引进行划分，数据表和索引表具有以下特点：

1) 一个数据对应一个 HBase 数据表。

2) 数据表对应于二级索引的数据部分。

3) 数据表中的检索字段对应于二级索引的索引标识符。

首先，在 HBase 二级索引表中，数据表的区域的起始行键需要合并到索引表的行键中，当数据表的记录不断增加，某个区域的数据超过了该区域预设的大小时，将进行分割操作并进行区域分片，相应索引表的区域也必须进行相应的操作。将二级索引的索引部分和数据部分放在同一区域，并使用不同的列

族来保证物理隔离。这是为了确保数据表和索引表的分布同步，保证索引及其相关数据可以存储在同一区域，这样可以显著减少区域间的网络流量，从而使数据检索过程尽可能本地化，使用更少的 RPC 调用。HBase 索引表的行键设计为：区域起始行键 + 索引标识符 + 索引值。通过这样的设计，我们可以将索引和数据放在同一个区域中，以最小化跨区域的 RPC 操作延迟，并使用不同的列族来确保物理隔离。表 1 为 HBase 中的数据表，为该日期字段设计 HBase 二级索引。例如，该数据表的标识符为 idx1，则第一个数据对应的索引的行键为 001idx120230101。二级索引被设计为反向索引，将键值对转换为值键对，将要查询的数据字段作为键，即二级索引的行键部分，并将相应的数据的行键作为值，即二级索引的数据部分。表 2 是表 1 数据表对应的 HBase 二级索引表，将原数据表的主键与值互换。原值作为主键，即二级索引的行键，原主键放在值的位置。其次，Redis 存储最新的数据索引信息，可以以键值对的形式实现二级索引，为实现更高的检索效率，其键的形式可以表示为：索引标识符+索引值，值存储相应的行键。例如行键为 001 的数据表中的数据为日期字段建立二级索引，键是 idx120230101，值为 001。生成的 Redis 二级索引表如下表 3 所示。

Table 1. Data table
表 1. 数据表

rowkey	date	content:dir
001	20230101	dir1
002	20230102	dir2
003	20230103	dir3

Table 2. HBase secondary index table
表 2. HBase 二级索引表

rowkey	date
001idx120230101	001
002idx120230102	002
003idx120230103	003

Table 3. Redis secondary index table
表 3. Redis 二级索引表

key	value
idx120230101	001
idx120230102	002
idx120230103	003

当客户端向 Hbase 集群发送查询数据的请求时，请求首先被协处理器捕获，以确定正在查询的数据是否是最新的数据。如果是最新的数据，通过 Redis 二级索引表中的 Redis 多条件查询，快速得到对应的行键。如果不是最新的数据，则从 HBase 二级索引表中查询相应的行键并返回，然后根据获得的行键，通过行键对 HBase 数据表进行快速查询，提高查询效率，及时减少磁盘访问开销，提高系统性能。基于协处理器和 Redis 的 Hbase 二级索引方案的数据查询流程如下图 1 所示。

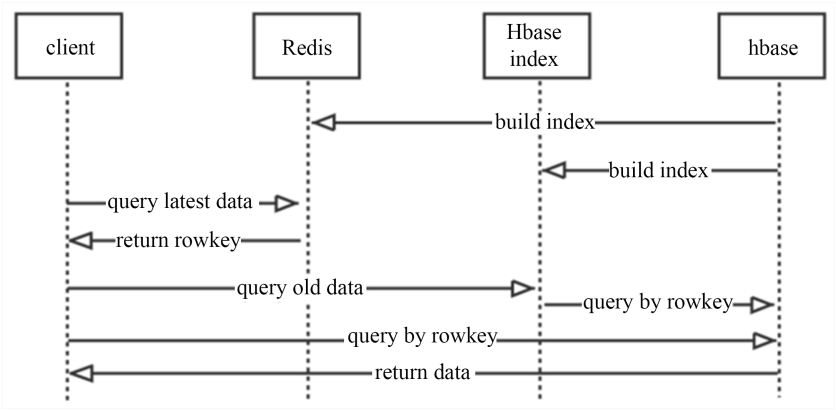

Figure 1. Data query process
图 1. 数据查询流程

4. 实验与分析

4.1. 实验环境

为了体现本次 HBase 数据高效读优化策略的可用性和有效性，实验利用 ZStack 部署了一个具有 5 个节点的 HBase 集群。集群中每台机器的配置为 8G 内存，100G 硬盘。操作系统和集群上安装的软件环境如下表 4 所示，硬件配置如下图 2 所示。

Table 4. Redis secondary index table
表 4. Redis 二级索引表

软件名称	版本号
Linux system	CentOS-7
Hadoop	3.2.3
Zookeeper	3.7.1
HBase	2.4.12
Redis	4.0.7
JDK	1.8.0

名称	CPU	内存	默认IP	物理机IP	集群	启用状态	所有者	高可用级别
centos7-3	1	8 GB	192.168.31.22	192.168.31.188	colony01	运行中	admin	None
centos7-2	1	8 GB	192.168.31.23	192.168.31.188	colony01	运行中	admin	None
centos7-4	1	8 GB	192.168.31.29	192.168.31.188	colony01	运行中	admin	None
centos7-5	1	8 GB	192.168.31.26	192.168.31.188	colony01	运行中	admin	None
centos7-1	1	8 GB	192.168.31.30	192.168.31.188	colony01	运行中	admin	None

Figure 2. Hardware configuration
图 2. 硬件配置

4.2. 缓存命中率优化实验

4.2.1. 实验设计

为了验证数据缓存组件的性能，本文使用了 7 个数据集，具体数据集如表 5 所示。每个数据集将执

行 10000 次读操作，分别记录每个数据集的实验结果并与理论结果对照，内存中设置集合容量为 1000，即缓存的记录的数量为 1000。基本步骤如下：

1) 计算查询和更新缓存数据的次数，得到一个周期内的查询和更新频率。

2) 根据公式计算每条记录的热值，并将其存储在一个有序的热值集中。

3) 在数据缓存开始时，缓存是空的，数据存储到 HBase 和 Redis 中。当缓存的数据量达到预设阈值 1000 时，调用改进后的缓存驱逐算法，删除热值最低的 n 条记录。使缓存的数据都是热值比较高的热点数据。

Table 5. Cache datasets
表 5. 缓存数据集

数据集	可缓存记录的个数	可缓存数据百分比
5000	1000	20%
3333	1000	30%
2500	1000	40%
2000	1000	50%
1667	1000	60%
1429	1000	70%
1250	1000	80%

4.2.2. 实验结果与分析

如图 3 所示，在存储容量百分比不同的情况下，改进后的缓存驱逐策略的缓存命中率高于 LRU 策略的缓存命中率。随着存储容量百分比的增加，两者的缓存命中率也增加，改进后的缓存驱逐策略的缓存命中率比 LRU 策略高约 10%。

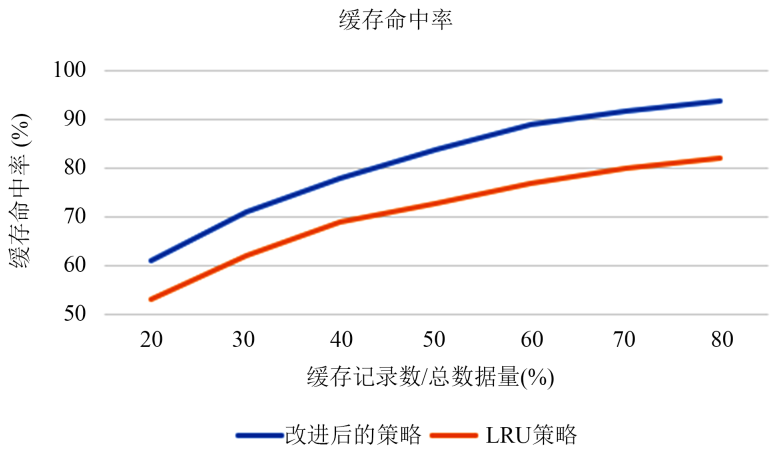

Figure 3. Cache hit rate
图 3. 缓存命中率

4.3. 查询模块优化实验

4.3.1. 实验设计

为了分别验证查询模块的性能，查询条件分为行键字段查询和非行键字段查询。

1) 行键字段查询

为了验证引入 Redis 并改进缓存驱逐策略后的方案的查询性能，预先存储的记录数为 100 万条，设置了 3 个批量查询数据集，分别为 5000、10000、15000。

2) 非行键字段查询

测试所提出的基于协处理器和 Redis 的 HBase 二级索引方案，先验证其在构建 HBase 二级索引过程中对数据写入性能的影响，再验证构建 HBase 二级索引后的数据检索性能是否有所提高。数据写入性能实验使用 insert 语句进行测试，分别插入 1W、10W、100W 的数据，并对非主键字段进行 HBase 全表扫描检索和二级索引检索测试，为了最大限度地减少测试误差，保证实验的准确性，对 insert 操作进行了 10 次测试，并取结果的平均值。数据检索性能实验使用查询语句检索非行键字段，先利用二级索引找到行键，然后再回表查询，分别从数据库中查询 10 条数据、100 条数据和 1000 条数据，并取 10 次测试结果的平均值。

4.3.2. 实验结果与分析

1) 行键字段查询

图 4 表明了引入 Redis 并改进缓存驱逐策略后的查询性能远远高于 HBase。随着查询记录数的增加，HBase 与改进后设计的查询模块的数据查询时间差距越来越大。实验表明利用 Redis 来缓存热点数据，然后改进原有的缓存驱逐策略后，极大地提高了数据检索速度。

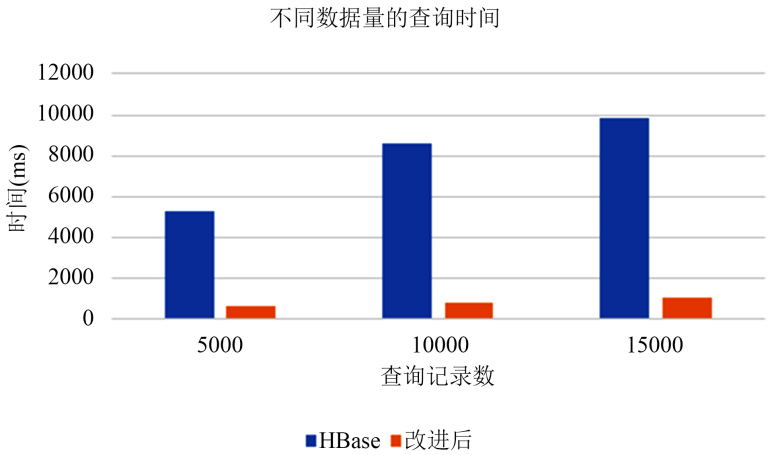

Figure 4. Query time of different data set
图 4. 不同数据量的查询时间

2) 非行键字段查询

从表 6 可以看出，在有二级索引和没有二级索引的 HBase 表中插入数据时，有二级索引的插入需要更多的时间，并且随着插入记录数量的增加，两种情况下的插入时间也会增加。因为向数据库写入数据时，由于要建立二级索引，需要分别创建和更新 HBase 和 Redis 数据的索引。因此，写入带有二级索引的数据会有点耗时。但是在向 HBase 数据库中插入数据时，协处理器在 HBase 索引表和 Redis 索引表中建立二级索引并更新 Redis 索引表中的索引，提高了检索效率，在 HBase 查询非主键字段时，通过二级索引间接获取行键，避免了全表扫描。由于新数据的索引存储在内存数据库 Redis 中，查询频率比较高的新数据的查询性能进一步提高。

根据表 7 的结果，在查询过程中使用 HBase 的二级索引可以显著提高查询的性能。这是因为 HBase

在查询非主键字段时需要全表扫描，这很耗时。改进后的 HBase 在查询非主键字段时，首先通过 HBase 二级索引表获取数据表的行键，然后根据行键进行查询，来实现更高的查询效率。并且由于 Redis 中存在新的数据索引表，在检索新数据时将极大地提高查询性能。HBase 在查询非行键字段时只能全表扫描来检索，所以数据检索时间很长，在引入并设计二级索引后，数据查询效率得到了显著的提升，这都意味着改进后的查询模块更适合海量数据的查询。

Table 6. Data table
表 6. 数据表

数据写入	1w	10w	100w
不构建二级索引	5047.5 ms	10128.9 ms	113558.7 ms
构建了二级索引	6094.2 ms	11368.7 ms	127961.5 ms

Table 7. Data query performance
表 7. 数据查询性能

查询数据	10	100	1000
全表扫描	2079.5 ms	3573.6 ms	5164.7 ms
构建了二级索引	217.9 ms	627.3 ms	1287.9 ms

5. 结论

本文主要针对 HBase 的数据高效读技术进行分析和研究，针对数据检索效率低的问题，提出了两个优化方案，最后搭建实验环境进行对比实验验证优化方案的可行性和可靠性并得出实验结论。本文主要的研究工作包括：

1) 针对 HBase 在查询数据时需要访问磁盘，查询速度慢的问题，为了加快数据的检索速度，本文提出使用 Redis 索引 HBase 的热点数据，并综合考虑数据的查询频率、更新频率和历史积热对缓存的影响，设计了一种基于数据查询频率和更新频率的热值缓存驱逐策略，提高了 Redis 的缓存命中率，减少了磁盘访问开销和访问时间。

2) 针对 HBase 在检索非行键字段时需要全表扫描，检索效率低的问题，本文提出了为非行键字段建立二级索引的策略，设计了一种基于协处理器和 Redis 的二级索引方案，极大地提高了检索速度。

3) 利用 ZStack 平台搭建部署了一个具有 5 个节点的 HBase 集群环境。设计实验方案并进行了对比实验验证其可用性和高效性，实验结果表明改进后的缓存驱逐策略的命中率也高于 LRU 策略，在查询模块引入 Redis 缓存热点数据并且为非行键字段建立二级索引后，改进后的查询模块的数据检索性能提升显著，极大地提高了查询速度。

本文提出的优化策略依旧存在着不足之处，需要在以后的工作中进一步地改进。本文提出的基于协处理器和 Redis 的二级索引方案没有考虑数据的热值，未来，将继续研究数据的冷热预测算法，在插入数据时进行冷热预测，并将其索引存储在相应的缓存索引表中，进一步提高内存数据库的利用率，提高 HBase 数据的检索效率。

参考文献

[1] Medel, V., Rana, O. and Bañares, J.Á. (2016) Modelling Performance & Resource Management in Kubernetes. *Proceedings of the 9th International Conference on Utility and Cloud Computing*, Shanghia, 6-9 December 2016, 257-262. https://doi.org/10.1145/2996890.3007869

[2] Li, Z.H., Zhang, Y. and Liu, Y.H. (2017) Towards a Full-Stack Dev Ops Environment (Platform-as-a-Service) for

Cloud-Hosted Applications. *Tsinghua Science and Technology*, **22**, 1-9. https://doi.org/10.1109/TST.2017.7830891

[3] Chi, Y.P., Yang, Y.T., Xu, P. and Yang, J.X. (2008) Design and Implementation of Monitoring Data Storage and Processing Scheme Based on Hadoop. *Computer Applications and Software*, **35**, 58-63+157.

[4] Xia, C.J. (2015) Research on HBase retrieval Speed Improvement based on Coprocessor mechanism. Master's Thesis, Hunan University, Changsha.

[5] Konstantinou, I., Tsoumakos, D. and Mytilinis, I. (2013) DBalancer: Distributed Load Balancing for NoSQL Data-Stores. *Proceedings of the* 2013 *ACM SIGMOD International Conference on Management of Data*, New York, 22-27 June 2013, 1037-1040. https://doi.org/10.1145/2463676.2465232

[6] Chang, F., *et al*. (2008) Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, **26**, 1-26. https://doi.org/10.1145/1365815.1365816

[7] Xia, L., Chen, H. and Sun, H. (2014) An Optimized Load Balance Based on Data Popularity on HBASE. *Proceedings of* 2*nd International Conference on Information Technology and Electronic Commerce*, Dalian, 20-21 December 2014, 234-238. https://doi.org/10.1109/ICITEC.2014.7105609

[8] Dang, P. (2019) Design and Implementation of HBase Hierarchical Auxiliary Index System. Master's Thesis, Xidian University, Xi'an.

[9] 丁飞, 陈长松, 张涛, 等. 基于协处理器的 HBase 区域级第二索引研究与实现[J]. 计算机应用, 2014(Z1): 181-185.

[10] Levandoski, J.J., Larson, P.Å. and Stoica, R. (2013) Identifying Hot and Cold Data in Main-Memory Databases. 2013 *IEEE* 29*th International Conference on Data Engineering* (*ICDE*), Brisbane, 8-12 April 2013, 26-37. https://doi.org/10.1109/ICDE.2013.6544811

[11] Zhang, C., Li, F. and Jestes, J. (2013) Efficient Parallel kNN Joins for Large Data in MapReduce. *Proceedings of the* 15*th International Conference on Extending Database Technology*, Berlin, 26-30 March 2012, 38-49. https://doi.org/10.1145/2247596.2247602

[12] Wei, G., *et al*. (2016) HiBase: A Hierarchical Indexing Mechanism and System for Efficient Hbase Query. *Chinese Journal of Computers*, **39**, 140-153 .

[13] Qu, L. and Li, X. (2017) A HBase index buffer solution based on TwemProxy. *Information Technology and Management*, **10**, 103-107+117.

[14] Li, K., Guo, K. and Guo, H. (2019) Financial Big Data Hot and Cold Separation Scheme Based on Hbase and Redis. 2019 *IEEE Intl Conf on Parallel & Distributed Processing with Applications*, *Big Data & Cloud Computing*, *Sustainable Computing & Communications*, *Social Computing & Networking* (*ISPA*/*BDCloud*/*SocialCom*/*SustainCom*), Xiamen, 16-18 December 2019, 1612-1617. https://doi.org/10.1109/ISPA-BDCloud-SustainCom-SocialCom48970.2019.00237