电子科技大学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

硕士学位论文

MASTER THESIS

论文题目 面向冷热数据的存储机制研究与实现

学 科 专 业 计算机科学与技术

学 号 201821080904

作 者 姓 名 许佳欣

指 导 教 师 侯孟书 教 授

分类号 _____ 密级 _____

UDC [注1] _____

学 位 论 文

面向冷热数据的存储机制研究与实现

(题名和副题名)

许佳欣

(作者姓名)

指导教师 侯孟书 教 授

电子科技大学 成 都

(姓名、职称、单位名称)

申请学位级别 硕士 学科专业 计算机科学与技术

提交论文日期 2021.3.18 论文答辩日期 2021.5.12

学位授予单位和日期 电子科技大学 2021 年 6 月

答辩委员会主席 _____

评阅人 _____

注 1：注明《国际十进分类法 UDC》的类号。

The Research and Implementation of Storage Mechanism for Hot and Cold Data

A Master Thesis Submitted to

University of Electronic Science and Technology of China

Discipline: **Computer Science and Technology**

Author: **Jiaxin Xu**

Supervisor: **Prof. Mengshu Hou**

School: **School of Computer Science & Engineering**

摘 要

数据量激增带来的存储需求不断提高，企业通过建立分层存储架构来优化存储空间。为了提高分层存储系统中的空间利用率，企业将数据分为高频访问的热数据与不常访问冷数据，并将数据访问特征与存储设备性能相匹配，以避免冷数据驻留于高性能存储层造成的存储污染。而冷热数据的识别精度会直接影响到分层存储系统的数据存取效率。因此，开展面向冷热数据的存储机制研究具有重要意义。

对冷热数据的研究起源于计算机高速缓存的分级设计，缓存替换中的经典算法大多是基于数据的访问时间、访问频率等数据的单一特征来决定数据是否需要被换出。这些算法对数据访问特征的考虑都具有一定的局限性，且无法很好的适应数据访问模式的变化。

论文针对以上传统缓存替换策略的不足，进行深入分析，提出了基于数据多维度特征的冷热数据判定方法，并基于该方法进一步设计了冷热数据的存储策略。针对传统策略的局限性问题，论文基于数据的访问时间、访问频率、数据关联性三个方面的特征，对数据的温度进行量化，依据此温度对数据未来的访问模式进行估计，实现对冷热数据的判定。针对数据存储无法适应数据访问模式变化的问题，论文提出了基于数据温度的数据迁移策略，根据热数据库中数据温度分布情况来动态调整该库的存储阈值，将超过阈值的数据迁移至冷数据库以完成冷热数据的分离存储。

最后，论文基于 Redis 与 HBase 数据库对所设计的冷热数据存储策略进行了原型实现，并完成了对冷热数据判定策略的精度测试与整体存储策略的功能、性能测试。测试结果验证了论文所提出策略较于LFU与LRU算法有更好的识别精度，保证了嵌入冷热数据迁移策略后的 Redis 数据库的读写可用性，且针对多种数据访问模式其读写性能均有所提升，达到了预期目标。

关键词：冷热数据，缓存替换，数据迁移，分级存储

ABSTRACT

The proliferation of data volumes has led to increasing storage requirements, enterprises are optimizing storage space by creating tiered storage architectures. To improve space utilization in a tiered storage system, enterprises divide data into hot data and cold data depending on the frequency of access, and match data access characteristics to storage device performance to avoid storage pollution caused by cold data residing in the high-performance storage medium. The identification accuracy of hot and cold data will directly affect the data access efficiency of a tiered storage system. Therefore, it is of great significance to study the storage mechanism for hot and cold data.

The study of hot and cold data originated from the hierarchical design of computer caches. Most of the classical algorithms in cache replacement decide whether data needs to be swapped out based on a single characteristic of the data, such as the access time and access frequency. These algorithms are partial in considering the characteristics of data access and cannot well adapted to changes in data access patterns.

This paper presents an in-depth analysis of the shortcomings of the traditional cache replacement strategy, proposes a method for determining hot and cold data based on the multidimensional characteristics of the data, and further designs a storage strategy for hot and cold data based on this method. In view of the limitations of the traditional strategy, this paper quantifies the temperature of the data based on the characteristics of three aspects: access time, access frequency and data relevance, and estimates the future access patterns of the data based on this temperature to realize the determination of hot and cold data. To address the problem that data storage cannot adapt to changes in data access patterns, this paper proposes a data migration strategy based on data temperature, which dynamically adjusts the storage threshold of the hot database according to the distribution of data temperature in the database, and migrates data exceeding the threshold to the cold database to complete the separation of hot and cold data storage.

Finally, based on Redis and HBase database, this paper implements the prototype of the cold and hot data storage strategy designed, and completes the precision test of the cold and hot data determination strategy as well as the function and performance test

of the storage strategy. The results verify that the proposed strategy has better identification accuracy than LFU and LRU algorithms, ensures the read and write availability of Redis database after embedding hot and cold data migration strategy. In addition, the read-write performance of Redis database has been improved for various data access modes, reaching the expected goal.

Keywords: Hot and cold data, cache replacement, data migration, tiered storage

目 录

第一章 绪论

1.1 研究背景与意义

随着 4G 网络的普及以及智能手机的发展，人们每天的工作生活、娱乐消遣都伴随着网络应用的使用。在享受网络智能带来的便利同时，人们也在无意识间产生着大量的数据。互联网企业为了给用户带来更好的用户体验，利用大数据等技术手段在应用背后做着支撑，而这些技术都基于海量的用户数据才得以实现。这庞大的数据量早已达到 PB 级以上的规模，且每日还在以数百 TB 的速度不断增长，存储成本成为了许多企业不得不关注的现实问题。

为了给用户提供更加优质的服务，企业除了对软件及后台进行技术优化外，最直接的办法就是使用更高性能的存储设备来存放数据，以提高数据的读取速度实现业务性能提高。HDD（磁盘）作为最常用的存储设备，其机械结构由磁臂和旋转的盘片组成，在数据读取方面采用 LBA（逻辑块地址）寻址，其访问时延为寻道、旋转与传输三步用时相累加，这导致 HDD 的成本低廉但同时其存取速度难以满足高速访问需求。SSD（固态硬盘）采用 NAND 闪存芯片作为存储介质，突破了 HHD 的机械延迟所导致的性能瓶颈，其随机读取速度几乎能达到与顺序读取性能相近。基于闪存的 SSD 根据其存储单元又分为两类：基于高成本小容量的 SLC（单层单元）和基于低成本大容量的 MLC（多层单元），SLC 相比于 MLC，虽然存储容量较小，但小容量所避免的纠错开销也使其在读取速度方面更快。由此 SSD 在存储容量与性能上也存在差异，所适用的存储场景也有所不同。总体来说，SSD 在性能上要远优于 HDD，但相对的问题是其价格较高、存储容量较小，且存在读写寿命，因此目前用户还是没有办法将 SSD 代替 HDD 来大面积使用。基于性能和价格的综合考虑，企业采用混合存储池是比较好的解决办法，将物理设备添加到存储池中，并按照需要从共享池里分配存储空间。混合存储架构[2]如图 1-1 所示，主存储系统采用如 SSD 高性能磁盘系统支持，混合其他高容量低成本的 HDD 甚至磁带库辅助存储。

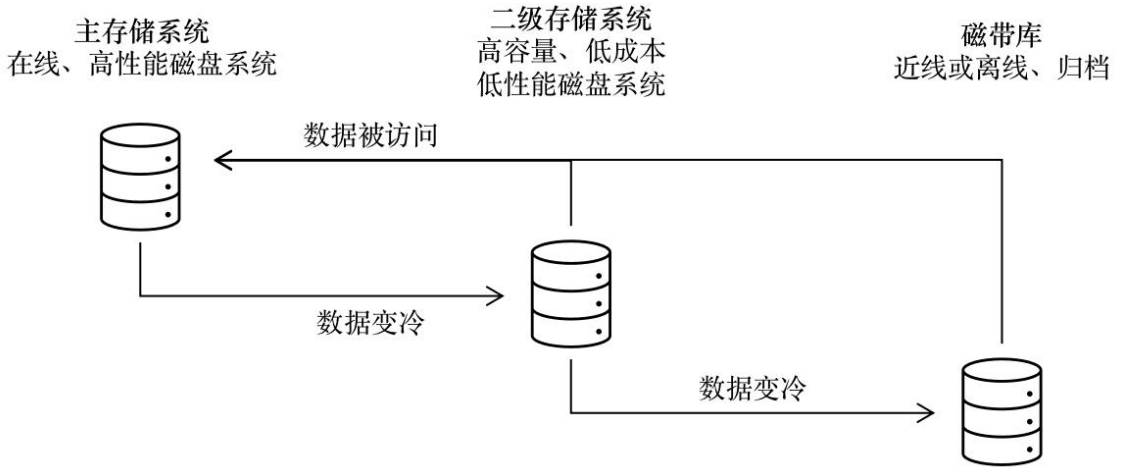

图 1-1 混合存储架构示意图

混合存储池是混合采用 SSD 和 SATA 磁盘作为存储，根据数据访问需要来存放数据，这样兼顾了整体的数据访问速度以及存储容量。ZFS[1]文件系统又称动态文件系统，是第一个 128 位文件系统，它最初是由 Sun 公司为 Solaris 10 操作系统开发的文件系统，其主要设计目标即包括了混合存储池的概念。ZFS 能够自动识别存储空间，以管理 SSD 和传统硬盘驱动器的混合构架，从而提高系统的整体性能。随后 Sun Storage 7000 系列的存储系统利用了 ZFS 的优点，以低功耗和低成本实现了高于传统存储系统的性能，为用户提供了一套简单实用、性能优异的解决方案。

除了在存储架构上进行优化，将更有价值的数据存储在高性能存储设备上，能够更进一步提高系统性能。数据价值的思想最早源于高速缓存的分级设计，为了提高高速缓存的命中率，高速缓存在进行数据替换时会采用 LFU、LRU 这样的策略，这些策略是利用了数据的局部性原理，一定程度上也反映了数据留在高速缓存内的价值。冷热数据的概念也源于此，所谓热数据即经常被访问的数据，而冷数据与之相反是不常被用到的数据。将冷热数据应用到混合存储池的存储架构上，基于数据的温度，将不同温度的数据存储到相应特性的数据库中，既可以实现数据的海量存储，也可以同时兼顾到数据的快速定位访问，提高存储空间的利用效率。

1.2 国内外研究现状

1.2.1 冷热数据判定策略

缓存技术在计算机的各个领域均有所应用，如操作系统中的高速缓存、网站

的静态缓存，它保证了系统的性能以及服务的稳定性。传统缓存替换策略根据所基于的数据特征不同而大致分为两种，如基于访问时间的 LRU、LRU-K、2Q、MQ 和 MRU 等，基于频率的 LFU、LRFU、ARC 和 MFU 等。而随着新型存储设备与云存储等新技术的出现，存储层次也随应用场景的不同逐渐变多样化，传统基于单一特征的单层缓存管理技术已难以满足需要，越来越多的研究开始从基于多维特征的缓存替换策略以及多层缓存结构出发。

除了传统基于缓存替换策略的冷热数据判定方法外，还有很多基于数据特征进行价值建模的冷热判定研究。2009 年，吕帅等采用 HSM（Hierarchical Storage Management）思想，提出了基于 FC-SAS 和 SATAII 的两级存储模型[3]，并设计了 FV 数据价值评定模型和迁移过程控制策略，保证了对数据价值精确估计，以及在两个级别的设备之间高效的数据迁移，此外为用户提供透明访问，同时尽量减小了对系统访问性能的影响。

2011 年，江菲等人提出基于多指标的数据价值模型以及迁移策略[4]，它将文件的属性分为静态因素和动态因素，静态因素包括文件的大小、用户数量和内容，动态因素包括文件最近一次的操作时间、文件的读写频率以及文件之间的关联度。文献基于这六个维度的特征建立线性的价值评估模型，并加以预测影响因子来动态调整迁移的阈值，以完成基于多维度文件价值的迁移策略。

2013 年，施光源等人提出一个基于扩展块的数据特征模型 EDM[5]，EDM 基于扩展的块级粒度构建分层的数据管理特征模型，该模型通过统计分析管理数据对象的特征并将冷热数据迁移到相应的层来自动做出数据管理决策，从而实现对存储资源的高效合理使用。

2014 年，黄冬梅等人针对混合云存储架构中的数据迁移问题[6]，提出了一种数据集的大小 S 成反比关系、与数据存储的时间长度 T 和访问频率 F 成正比关系的迁移函数，其中数据存储的时间长度 T 是该数据被访问时间间隔差值的倒数和，而访问频率 F 则是将数据各时间段访问频率求和得到。最后通过比较数据迁移函数与所设定数据敏感度阈值来判定数据是否需要迁移，提高了数据平均利用率，但敏感度阈值设定成为了另一个待研究的问题。

2015 年，郭刚等人在内存云分级存储架构下提出一种基于数据重要性的迁移模型（MMDS）[7]，通过数据的大小、访问时间特征以及用户访问总量等因素来计算数据的重要性；数据的潜在价值参考推荐系统中的相似用户思想以及 PageRank 算法中的重要性排名思想来进行综合评估，再结合数据重要性和数据潜在价值两方面来决定数据重要程度；最后根据数据的重要程度，设计了一种数据迁移机制，以实现对在线海量数据的有效存储和访问。

2020 年，张雷等人为提高边缘节点存储效率，减少与云服务器请求次数，首先通过比较相邻时间窗口内用户请求内容集合，来预测用户请求的流行度，然后结合时效性、文件大小优先性、流行度预测多方面考虑，提出了确定文件价值的方法[8]，并以此价值作为缓存替换的依据，有效降低了边缘网络请求的平均时延。

基于特定数据库的冷热数据也有不少的研究。2016 年，王海燕等人针对原生 HBase 数据压缩策略忽略了数据的冷热性，导致其在选择需要压缩的数据时存在片面性和不可靠性的问题，根据对数据文件的访问频率将 HBase 数据分为冷热数据，并为数据设定访问级别；然后再增加评估层，选取选择方法时综合了考虑相邻区和统计列的两个方面，最终确定了基于数据访问级别的压缩策略选择方法[9]。2017 年，冯超政等人研究原生 MongoDB 数据库，对其通过数据量来进行分片迁移的自动分片机制进行改造，基于数据的访问特性，采用朴素贝叶斯算法来进行冷热数据判定[10]，文献还定义热负载值为数据分片中热数据所占的比重，并以此作为确定数据迁移时机的依据，最后根据数据片之间的热负载差异建立新的数据迁移策略，以优化自动分片机制。

1.2.2 冷热数据存储策略

关于冷热数据存储策略的研究同样有很多的类型。冷热数据的存储策略都基于数据分层存储的思想，而分层存储通常都由两种或多种不同存储性能的存储介质构成，如上文中所提到的 SSD 配合 HDD 搭建混合存储架构，这样的架构在实际实现时也存在多种类别，以应对不用的应用场景需要。一种最常规的思想即 SSD 作为高性能存储设备，可把 SSD 作为整个存储系统的高速缓存来使用，缓存同样可分为只读缓存、直写缓存和回写缓存三种类型。

在缓存作为只读缓存的存储结构中，当新的写入请求到达并且其访问数据不在 SSD 中时，该数据需要记录到 HDD 中。若访问数据在 SSD 中可用，则需要更新 HDD 中的数据，并且 SSD 中的数据也可能被丢弃或更新。这样的存储结构中将写入请求直接传递给 HDD，因此减少了对 SSD 的写入操作次数，延长了 SSD 的使用寿命。同时，节省了写数据所使用的缓存空间，有更多空间可用于读取数据以提高读取性能。为了减少不必要的写操作，2015 年 Y. Liu 等人提出了一种基于降级计数器和所提出的控制度量来检查数据热度的方法[11]，计数器记录了从 DRAM 到 SSD 的降级次数，并用于预测将来对 SSD 的访问，将热数据块迁移到 SSD。2016 年，D. Zhao 等人设计了一种启发式文件放置算法[12]，通过考虑传入工作负载的 IO 模式来提高缓存性能，在用户级别实现了分布式缓存中间件，以检测和管理经常访问的数据。同年 D. Luo 等人提出了一种结合了 SSD 和 SMR 设备的

混合存储结构[13]，该结构利用 SSD 作为只读缓存来提高随机读取性能。

直写缓存和回写缓存的区别，与在传统高速缓存中两种缓存写入的主存的方式类似。在具有直写式高速缓存策略的存储结构中，数据除了写入高速缓存外，还需要同时写入磁盘中，并且仅当数据成功写入磁盘时才被认为是成功的。此策略主要用于 DRAM 缓存，以避免 DRAM 作为易失性存储器而在断电期间丢失数据，同时也提高了缓存中写入数据的读取性能。而具有回写策略的存储结构可以先将数据写入缓存，然后再将其刷新到磁盘，这在某些情况下可以显着提高写入性能。在以 SSD 为缓存的混合存储系统中，由于 SSD 是非易失性存储器，因此很少采用直写策略，即使在驱动器断电的情况下，也可以保证数据安全。为了提高混合存储系统的写入性能，2016 年 S. H. Baek 等人利用 SSD 的持久性设计支持读写请求的缓存[14]，在故障情况下也可保证缓存元数据的完整性和缓存数据的一致性。且使用回写策略，将新数据记录到 SSD 中并延迟刷新到 HDD。但同样可能会导致数据同步问题。文献提出若优化策略为定期写入数据刷新可能适用在部分场景下改善数据同步问题。

除了将 SSD 在存储结构中作为缓存使用，还有很多研究是基于 SSD 的分层存储结构。在此模式中，SSD 中的数据是长期存储的，而非类似缓存形式存在，且与在 HDD 中所存储的数据没有冗余部分。在分层存储结构中，数据热度将决定数据的分配和迁移。2009 年，X. Wu 等人基于数据属性和各个层中设备的状态，将冷热数据定位在不同层间[15]。冷数据保留在较慢的层中，而热数据可能存储到不同的层中，如热顺序访问数据保留在 HDD 中以节省 SSD 带宽，因为 HDD 具有与 SSD 相似的顺序读写性能。同时为了节省 SSD 的使用寿命并减少 SSD 中的 GC 进程数量，文献考虑系统性能和向 SSD 写入数据的大小之间的权衡，避免了数据在不同层间的频繁迁移。2011 年，F. Chen 等人基于数据访问局部性特征[16]，找出热点数据保存到 SSD 层以提高读取性能，同时将写数据分配到 SSD 层，提高写性能，但导致了 GC 进程数增加、SSD 寿命降低的问题。

在分层存储结构中，为避免迁移带来的 SSD 寿命减少，数据位置优化的频率非常低。因为当热点数据区域发生变化时，系统需要相当长的时间才能将热点数据移动到 SSD。而在缓存存储结构下，当数据热度发生变化时，可以在短时间内优化数据位置，但频繁的数据迁移同样会影响系统的整体性能。混合两种结构也因此成为了一个研究方向。2011 年，S. Bai 等人引入了闪存磁盘转换层(FDTL)来将数据分配到不同级别的设备[17]。数据分配是基于效用最大化和能耗最小化的政策。它结合了硬盘顺序访问速度和 SSD 随机访问速度的优点，提高了 IO 性能及吞吐量并降低了能耗。2013 年，S. Hayashi 等人在包含 SSD 和 HDD 的混合存储结构

中合并了分层和缓存方法[18]，通过定义分配比例来调整分层和缓存结构中的 SSD 盘的数量。其对于写密集型的工作负载系统性能，与不使用组合的系统相比有明显提高。

经过上述研究发现，关于冷热数据的研究不论是判定策略还是存储策略，在近几年仍有被讨论，而多数的研究都还是基于传统缓存替换策略的优化和延伸，基于数据特征的研究仍未有比较泛用的策略可以支撑。而面对即将到来的 5G 时代，在互联网上日增的数据量还将进一步扩大，研究基于冷热数据的存储方案仍有重要意义与价值。

1.3 论文主要研究内容

本文将从传统缓存替换策略的出发，分析了在传统缓存替换策略与冷热数据判定中存在的局限性问题，在此基础上研究设计了基于数据多维度特征的冷热数据判定模型，并以 Redis 与 HBase 所组成的混合存储架构为平台，对冷热数据存储方案进行了设计与实现，以求达到在高性能设备中存放高价值数据的目标。在实现冷热数据存储方案的过程中，主要展开了以下几个方面的研究：

1）深入研究了冷热数据判定策略，分析以往缓存替换策略的研究，发现其中存在的局限性，然后围绕冷热数据的特征，提出了一种综合数据多维度特征的冷热数据判定策略，以辅助冷热数据存储工作展开。

2）深入研究了冷热数据存储策略，基于常见分级存储数据库的数据迁移策略，提出在冷热数据库间进行数据迁移的方案，并对冷库与热库数据的存储进行适当的优化，进一步提高存储空间利用率和整体系统性能。

3）对实现的系统进行全面的功能和性能测试。论文最后对所实现的原型系统进行了全面的测试工作，将冷热数据判定策略与传统缓存替换策略进行了比对测试，同时将加入冷热数据识别的存储方案与一般存储方案进行了对比测试，验证了加入冷热数据识别后系统整体存储性能的变化。

1.4 论文组织结构

本文围绕冷热数据识别与存储策略展开，共包含六个章节，研究内容分别如下：

第一章为绪论部分，介绍了本论文的研究背景和意义，阐述了在混合存储池架构下采用冷热数据思想的价值，并对冷热数据识别相关的国内外发展现状进行了介绍，最后介绍了本文的核心研究内容及论文结构安排。

第二章对本文中所涉及的相关理论和技术进行综述，首先介绍了传统缓存替

换策略，包括 OPT、LFU、LRU 等算法，和与之相关优化算法，然后对其他基于多维度数据价值研究的冷热识别策略进行了分类综述，最后介绍了分级存储与数据迁移相关策略。

第三章介绍了本文冷热数据判定策略的研究与实现，从传统问题分析、理论基础、判定策略实现与判定策略应用几个方面对冷热数据判定模块进行了详细阐述。

第四章基于冷热数据的特征提出存储方案的设计目标，再结合冷热数据判定策略，介绍了冷热数据迁移模块的详细设计与实现。

第五章为系统的测试与分析，将本文冷热数据存储系统与不加入冷热数据分离的数据库系统分别进行对照测试。主要为功能上的基本功能测试、数据完整性测试、读写性能测试，对每项测试都会进行简要的分析。

第六章为总结与展望，对全文所完成的工作进行总结，并分析了工作中有待改进的地方以及不足之处，展望了未来的可研究方向。

第二章 相关理论与技术

本章节介绍了冷热数据存储机制在实现过程中所利用的理论基础以及实现方案，具体包括冷热数据判定策略所基于的两种缓存替换策略及其相关策略的介绍，以及冷热数据存储策略所基于的分层存储技术发展过程及最新技术介绍。

2.1 缓存替换策略

高速缓冲存储器是位于中央处理器与主存储器间的一级存储器，它相较于采用 DRAM 技术的主存储器，采用的是成本更为高昂但读写速度更快的 SRAM 技术[19]，同时其容量也比主存要小很多。

高速缓存的工作实现主要由高速存储器与联想存储器配合实现。高速存储器有着与主存相仿的行列式存储结构，其列数与主存相同，但行数要远少于主存的行数，它用于存储从主存中读取的数据。而联想存储器与高速存储器的存储结构一致，主存数据会寻找高速存储器中的同一列写入数据，此时联想存储器将会把该数据在主存中的行号在同一位置进行记录，以将主存与缓存的地址联系起来。当中央处理器需要读取主存数据时，就会将存取地址译码得到的列号并在联想存储器该列寻找行号，若找到则表示在高速存储器中存在该数据，则发生缓存命中，反之则需要在主存中继续读取数据。高速缓存工作原理如图 2-1 所示。

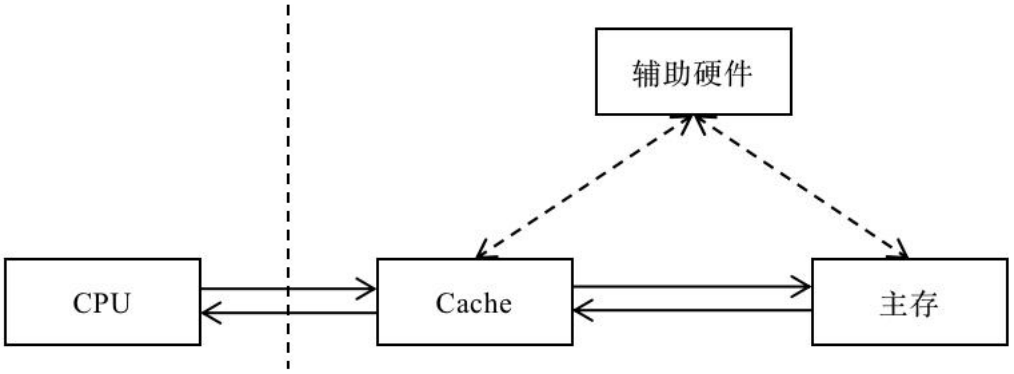

图 2-1 高速缓存工作原理示意图

当高速存储器存满数据时，需要考虑丢弃一些已存数据，以为需要访问的新数据留出空间，此时选择合适的数据进行换出将会对高速缓存的命中率有很大影响，这也直接影响了计算机系统的处理速度。为保证高速缓存命中率，缓存都是按照一定的算法进行替换。

2.1.1 LRU 算法

一种最常见的算法是最近最少使用算法（LRU）[20]，此算法以数据的最近使用时间为参考，考虑的是数据时间局部性原则，当缓存空间占满时，将最近最少使用的数据与新读取的数据进行置换。LRU 在具体实现时，一般都会为高速缓存存储单元保留时间信息，并根据此信息来追踪最近最少使用的高速缓存单元，且在每次使用高速缓存时，所有其他单元的寿命都会改变。LRU 原理示例如图 2-2 所示。

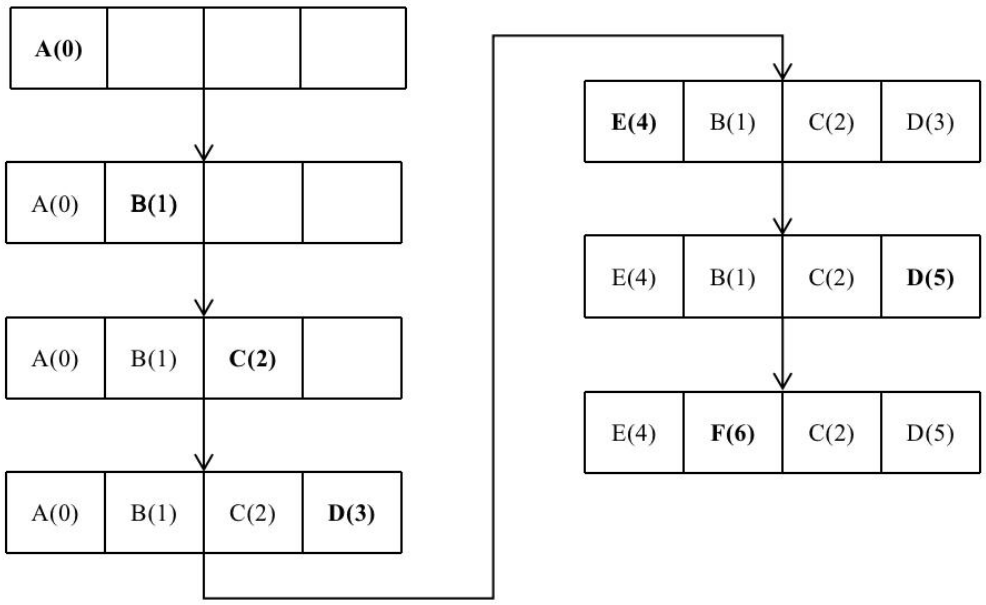

图 2-2 LRU 示例图

上图中高速缓存空间为 4 个存储单元，加粗数据即为当前从主存中读取的数据，每个数据括号内的数字代表了该数据进入高速缓存的时间位。示例中的数据读取顺序为 ABCDEDF，可见在读取数据 E 时，高速缓存是未命中的，此时按照时间位找到最低顺位的 A 将其换出；而读取数据 D 时，高速缓存中能够找到数据，则将 D 的时间位进行更新。

LRU 作为一种经典的缓存替换算法，它有着很多的变形。1993 年，E. J. O’Neil 等人提出的 LRU-K 算法[21]，将原始 LRU 算法的判断标准由“最近使用过 1 次”拓展为了“最近使用过 K 次”。相比与 LRU 算法，LRU-K 需要额外维护一个队列，用于记录数据的访问历史，仅当数据被访问达到 K 次时，数据才会被放入高速缓存当中。当发生缓存未命中时，算法会选择第 K 次访问时间距当前最久的数据进行替换。LRU-K 所维护的访问历史队列也有空间大小限制，它的作用相当于对进入高速缓存的数据进行了一次过滤选择，而历史队列中的数据淘汰原则

仍需按照一定策略进行，如 FIFO、LRU 等。LRU-K 的设计主要是为了解决缓存污染的问题，它具有 LRU 算法的优点，同时也避免了缓存数据的频繁置换以及循环访问的问题，而算法中 K 值的选择也对效果有一定影响，越大的 K 值会使得进入缓存的数据命中率越高，但同时访问历史队列的维护成本越高，实际应用中选择 K 值为 2 是综合各种因素的最优选择。LRU-K 原理示例如图 2-3 所示。

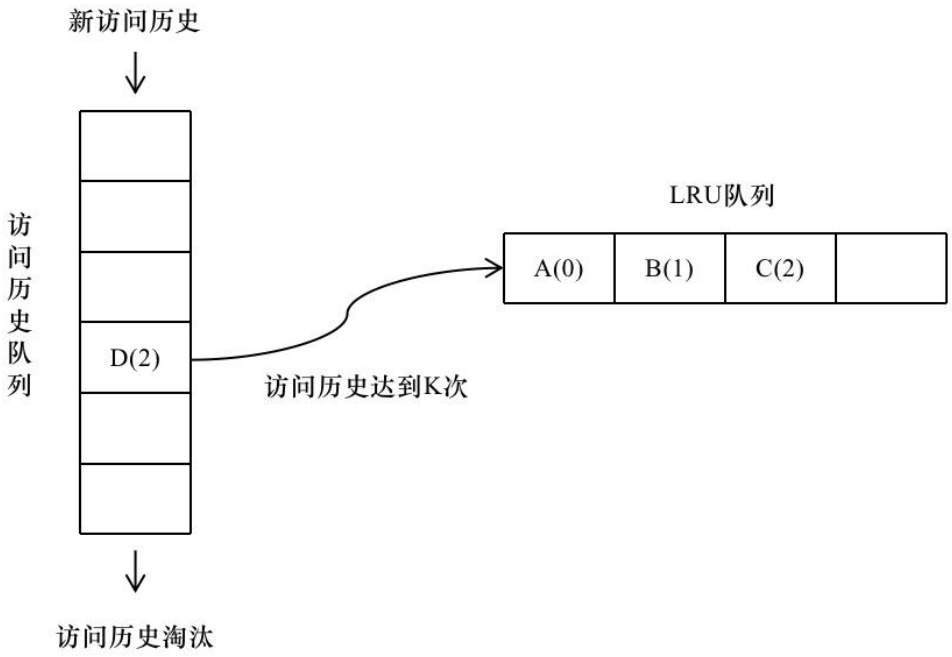

图 2-3 LRU-K 示例图

如上图所示，新数据不会直接进入到高速缓存当中，而需在访问历史队列中达到 K 次访问才可进入高速缓存的 LRU 队列里。1994 年 Theodore Johnson 等人提出的 2Q 算法[22]，就类似于 K 值为 2 的 LRU-K 算法，2Q 算法使用 FIFO 来维护访问历史队列，当数据被访问达到 2 次时再进入缓存中使用 LRU 算法进行替换。

上面提到的 LRU 算法变形都需要额外的队列进行维护，而针对特定的问题如循环访问问题，有一些更为简单的变形能够有效解决。Hong-Tai Chou 和 David J.DeWitt 在 1985 年提出的 MRU 算法[23]，与 LRU 相比更倾向于保留较旧数据，在发生替换时会优先丢弃最近使用的数据，此算法针对数据循环访问场景有着很好的表现，且也无需额外的空间来维护。MRU 原理示例如图 2-4 所示。

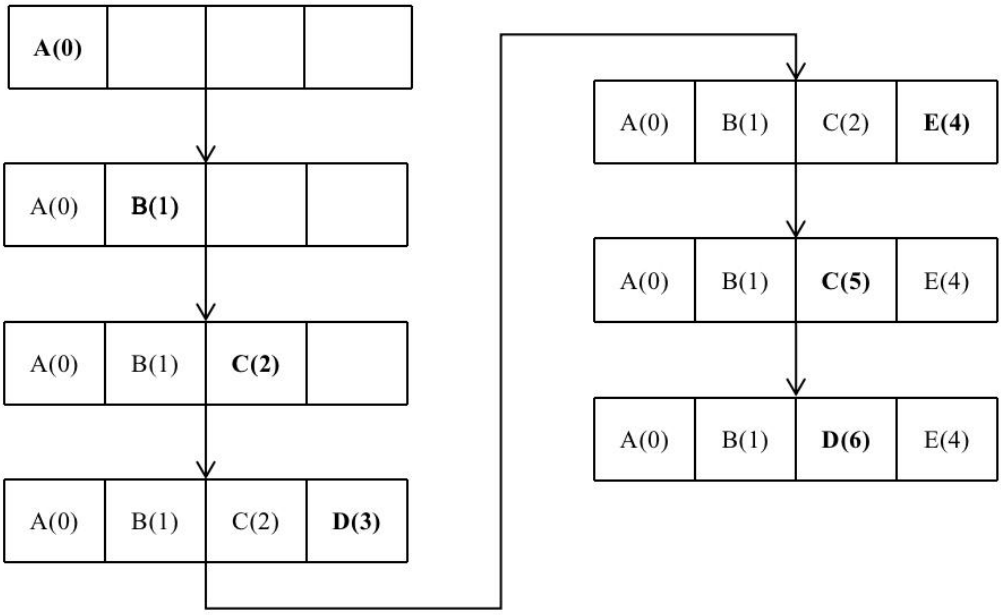

图 2-4 MRU 示例图

上图显示在发生缓存数据替换时，优先选择了最近的数据，这与 LRU 的设计正相反，但当访问序列为 ABCDDCBA 这样的循环序列时，则能发挥出较好的效果。

以上都是以最近一次或多次访问时间为标准来量化数据的时间局部性特征，2002 年 6 月，张晓东等人提出的 LIRS 算法为局部性的量化提供了新的思路，此算法也是针对 LRU 的优化。LIRS 使用两个参数来量化一个存储单元块[24]，一个是 R（recency）即最近被访问的时间，一个是 IRR（Inter-Reference Recency）表示同一块在两次访问中间访问其他不重复的块数量。LIRS 参数的示例如图 2-5 所示。

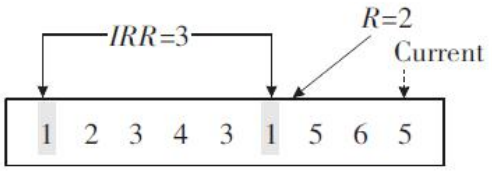

图 2-5 LIRS 参数示例图

在此基础上，LIRS 将数据块分为了两个部分，一个是 LIR 用于存放已经被访问两次的数据，一个是 HIR 用于存放当前仅被访问一次的数据，LIR 中的数据块通常会在高速缓存中常驻，而 HIR 则有高概率被换出。当 HIR 中任意块的 IRR 小于 LIR 中所有块的最大 R 值（Rmax）时，则该块将会被换到 LIR 中。LIRS 算法

会设置一个栈和一个 FIFO 队列，栈 S 负责 LIR 数据的淘汰，队列 Q 负责 HIR 数据的淘汰，栈 S 中除了存储 LIR 数据外，还会存储 HIR 中 R 值小于 Rmax 的数据。LIRS 算法过程如图 2-6 所示。

											Recency	IRR
A5									X		0	INF
A4		X					X				2	3
A3				X							4	INF
A2			X		X						3	1
A1	X					X		X			1	1
TIME	1	2	3	4	5	6	7	8	9	10		

图 2-6 LIRS 原理示例图

在上图中“X”表示在时间 t 访问一个块，所有块在第一次被访问时其 IRR 都默认为无限大（INF）。可以看到在时间 10 上，A1 块由于在时间 6、8 都被访问过，因此 A1 的 IRR 记录为 1，且时间 8 后距离现在仅访问过一个块，因此 A1 的 Recency 也为 1，其他数据块的 Recency 与 IRR 计算同理。那么在时间 10，LIRS 算法将分为两组，LIR 组包括{A1，A2}，HIR 组包括{A3，A4，A5}，LIR 是存储在高速缓存中的部分。若在时间 10 发生了对 A4 的访问，则会发生缓存未命中，此时根据 LIRS 算法，LIR 组将加入 A4，而 A2 不会被淘汰，位于队列 Q 底的 A3 将被淘汰。

LIRS 算法相较于 LRU，较好的解决了循环访问的问题，且在相对均匀的数据访问下有着更优的缓存命中率，但其缺陷在于实现较为复杂，且空间开销难以确定。还有其他很多类似算法在不同场景下有不错的效果。RR 算法在发生替换时随机选择一个候选数据，在必要时将其丢弃以腾出空间[25]，该算法不需要保留有关访问历史记录的任何信息而具有较好的性能，且能够进行高效的随机模拟[26]。SLRU 算法将缓存分为试用段和受保护段，利用缓存分级的思想对部分数据在被替换前再次给予被访问的机会[27]。PLRU 算法通过建立一棵二叉搜索树，通过树节点标识来决定需要访问的 LRU 元素[28]，避免了复杂的硬件设计。

2.1.2 LFU 算法

除了 LRU 之外，另一种常见的算法是最不经常使用算法（LFU）[29]，此算法以数据的最近使用频率为参考，当缓存空间占满时，将最不经常使用的数据与新读取的数据进行置换。LFU 在具体实现时，最简单方法是为加载到缓存中的每个

块添加一个引用计数器。每次引用到该块时，将其计数器增加一。当发生缓存未命中时，高速缓存将删除引用计数最低的块并插入新的块。最理想的 LFU 实现，应该是能保证每个数据都有一个计数器，但实际实现时只能保证缓存中有一个用于存储数据的计数器，一旦该数据被换出，相应的计数器也将被遗忘。LFU 原理示例如图 2-7 所示。

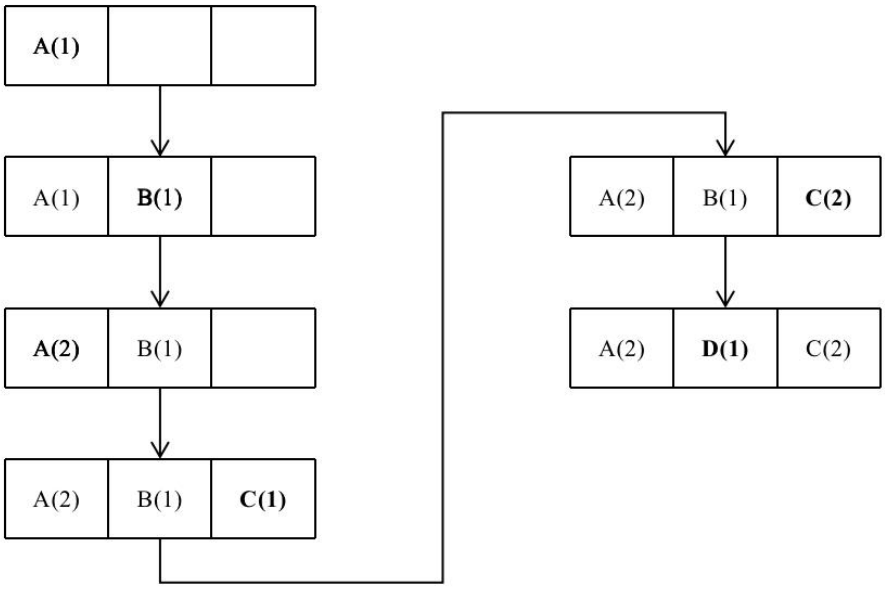

图 2-7 LFU 示例图

上图中高速缓存空间为 3 个存储单元，加粗数据即为当前从主存中读取的数据，每个数据括号内的数字代表了该数据在高速缓存中的访问频率。示例中的数据读取顺序为 ABACCD，当读取数据 D 时，发生高速缓存未命中，按照访问频率找到最低顺位的数据 B 将其换出，读取已经存在的数据时则将计数器进行累加。

LFU 相比于 LRU，对于数据循环访问以及局部随机访问的场景有着更高的效率，但其同样存在缺陷。考虑在内存中的一个数据，在短时间内被频繁访问，但长时间后不被再次访问。由于它的计数器值很大，因此即使在相当长的时间内不会再次使用它，它也会驻留在高速缓存中。这使得其他实际上可能更经常使用的数据很容易被清除，此外刚进入高速缓存的新数据也容易很快就会再次被删除，因为它们以低计数器开始。这也被称为 LFU 的缓存污染。

为应对 LFU 的缓存污染问题，Arlitt M 等人在 1990 年提出一种改进算法 LFU-Aging 算法[30]。此算法除了考虑数据访问频率外，还加上了访问时间因素，但它并非直接增加每个数据访问时间的记录，而是采用计算缓存中数据平均访问计数来约束极高频率数据的出现。此算法首先会设定一个平均访问计数阈值，每

当发生缓存命中时则会重新计算缓存中数据的平均访问计数，一旦该计数达到或超过此阈值，则采用减去固定的数值或者减去当前值一半的方式，缩减缓存中所有数据的访问计数。LFU-Aging 避免了短时间高频访问数据一直驻留缓存的问题，但其需要额外的计算开销来对算法进行维护。

除上述方法之外，Jin S 等人还从另一个思路尝试解决了 LFU 的缓存污染问题。他们提出了 LFUDA 算法[31]，为高速缓存增加了“年龄”的属性，且缓存中每个数据的计数不单由该数据被访问次数组成。在 LFUDA 算法中，每当一个数据进入高速缓存或者在缓存中被命中时，都需要重新计算计数器的值为该数据被访问次数与高速缓存年龄的总和；而高速缓存的年龄则在每当有数据被换出高速缓存时被设定为该数据的计数器，这样也保证了高速缓存的年龄一定小于等于缓存中所有数据的最小计数值。LFUDA 原理示意图如 2-8 所示。

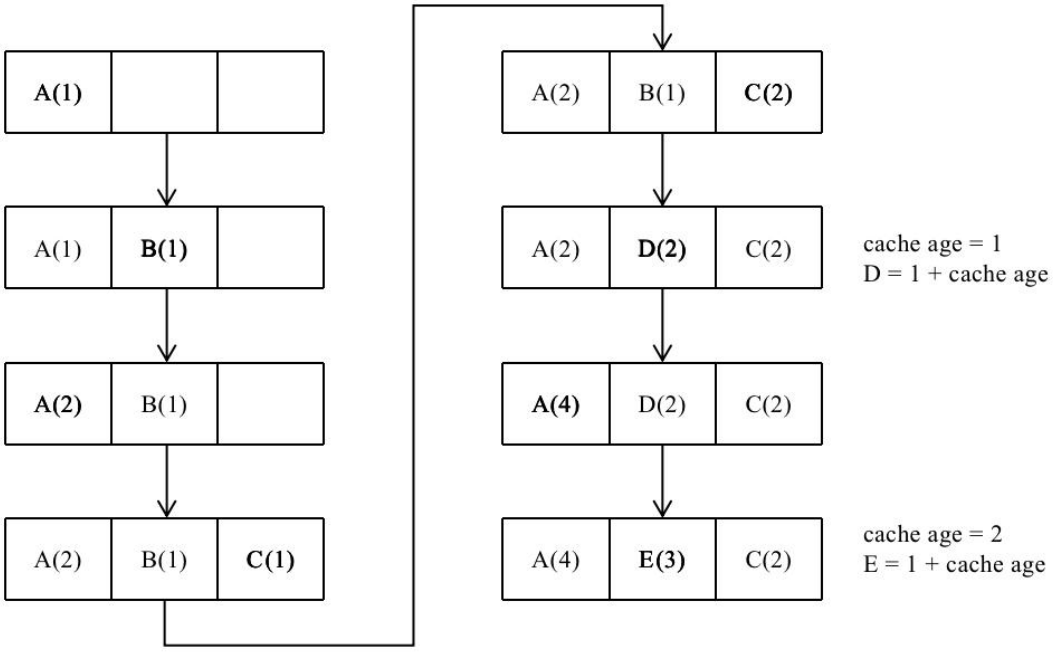

图 2-8 LFUDA 示例图

在上图中，未发生缓存未命中前，所有数据计数与 LFU 一样。在访问数据 D 时发生了数据替换，此时将缓存年龄设定为被替换出去的 B 的计数，而新加入缓存的 D 的计数记为 D 被访问次数与当前缓存年龄的和。缓存年龄的引入会将整体数据计数水平不断上抬，在 LFU 发生缓存污染时，LFUDA 将在一定时间后解决该问题，避免了污染数据的长期停留。

LFU 算法及相关算法均考虑在缓存环境中，其主要开销都来自于对数据的定位

和访问频率的记录。而在很多场景下对数据访问频率的精确计算是不必要的，一些方法采取概率估计的方式来降低存储与定位的开销。Count-min sketch 方法[32]，对于时间点 t 的 item，采用 d 个 hash 函数映射到 d 个 hash 表对应的位置并累加次数，查询某个 item 次数时，返回 d 个 hash 表对应位置的频数的最小值以代表该数据访问频数的估计。Count sketch[33]在 Count-min 的基础上引入一组取值范围为{1, -1}的 hash 函数 S1,...,Sd, 对于时间点 t 的 item, 采用 d 个 hash 函数映射到 d 个 hash 表对应的位置且对应位置计数器增加 Sd(it)。当查询某个 item 次数时，返回 d 个 hash 表对应位置的频数的中值，减小了 Count-min 频繁累加导致的误差。Conservative Update sketch [34]更改了数据更新策略，在更新时仅更新最小值，CU sketch 可以和不同的 sketch 相结合，但存在不支持删除操作的缺点。

2.2 数据分层存储技术

全球网络存储工业协会 SNIA 分层存储定义为“根据价格、性能或其他属性在物理上划分为多个不同类的存储，且数据会根据存取情况或其他考虑动态的在分级存储间移动”。如何在用户使用体验和厂家存储成本间找到平衡一直是被存储相关工作者讨论的话题，分层存储对此问题的解决办法就是将数据在正确的时间存储在正确的层级上，以满足性能和成本双方的考虑。

最早的分层存储将存储分为三层，第一层是由 FC/SAS 硬盘驱动器组成，第二层是由 ATA/SATA 驱动器组成，第三层是由磁带驱动器支撑。而随着新技术的发展出现了第零层，即由 SSD 驱动器构成，除了这些常规分层外，还出现了将存储缓存、公有云、服务器缓存作为一个分层的划分方式。如今的分层存储技术都是基于前几代的技术不断进化而来的，但也有本质上的不同。

2.2.1 HSM 分层存储管理

分层存储管理（HSM）是一种数据存储技术，其核心思想是在高成本和低成本存储介质之间自动的移动数据[35]。HSM 的产生背景是由于高速存储设备（例如固态驱动器）比慢速设备（例如硬盘驱动器，光盘和磁带驱动器）的成本要昂贵很多。虽然可以在高速设备上存取所有数据是理想的，但是对于许多组织而言其成本太高。HSM 系统的做法是将企业的大量数据存储在速度较慢的设备上，然后在需要时将数据复制到速度较快的磁盘驱动器上，本质上就是将快速的磁盘驱动器转换为较慢的大容量存储设备的缓存。此外 HSM 系统会监视数据的使用方式，基于此选择出更适合留在高速设备的数据，并在保证数据一致性的条件下对其他数据进行迁移。

HSM 的概念最早起源于上世纪 80 年代的数据设施分层存储管理 (DFHSM)，那个年代的联机硬盘价格过于昂贵，以至于没有能力去存储活跃数据。DFHSM 采用的办法是使用文件描述符配合迁移层级从磁带自动分段和检索数据，而文件将根据大小阈值以及创建时间被迁移至不同的层级中，这样的迁移方式对之后的数据迁移策略都有着启蒙式的影响。到了上世纪 90 年代，HSM 开始被广泛接受，且磁盘的成本也变得可以接受，此时 HSM 系统将符合条件的文件数据迁移到 HSM 服务器上，同时迁移文件也可直接在 HSM 服务器上备份获取，而文件迁移策略考虑的因素也加入了文件的大小、创建时间、类型、位置以及存储阈值和迁移计划。

2.2.2 ILM 信息生命周期管理

ILM 作为一种信息管理模型，其概念早在上世纪 60、70 年代就有被提出，而在互联网发展的背景下逐渐受到企业重视。ILM 的核心思想是监督信息从产生到回收的整个生命周期，以在信息的各个阶段优化信息的利用效率和成本，帮助企业以最低的成本获得最大的价值[36]。ILM 通常是针对业务信息流动来做出相应管理的，业务信息流动如图 2-9 所示。

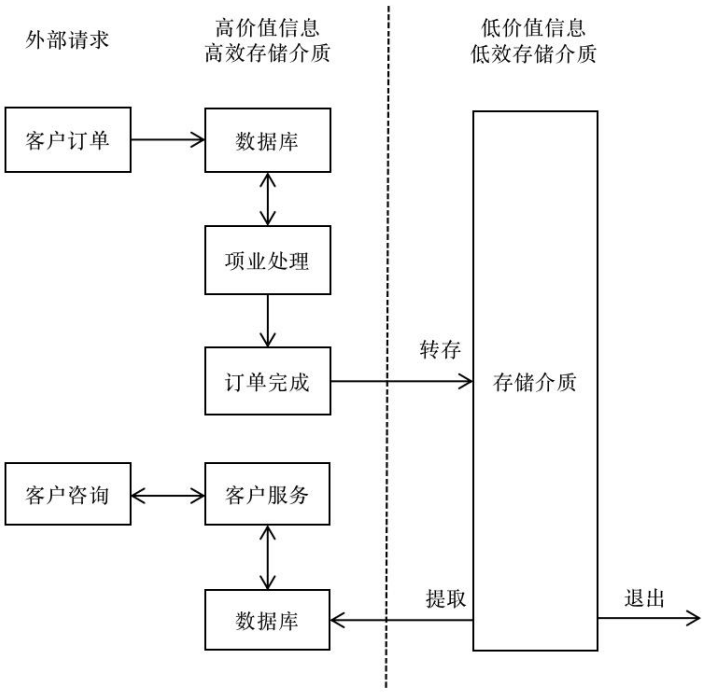

图 2-9 业务信息流动图

如图所示，一笔业务信息的诞生由客户下单开始，此时这份信息会被多条业务线同时需要和访问，此时信息拥有较高的价值。一旦订单完成了，这条业务信息

就不会被频繁访问，其价值也逐渐下降，此时便可将它转存到低成本的存储介质当中。如果该订单再次因后续客服问题被业务需要，业务信息价值再次被激活，则再将其读取到高效存储介质当中。等到订单超过了保修期之类的服务期限，则其信息价值会进一步下降，甚至就完成了该信息的生命周期而被销毁。整个信息生命周期过程，信息的使用与存储都会随其价值不断变化。

除了信息在生命周期中所表现的价值外，信息本身的内容和重要性也需被考虑[37]，如一些需要被审查的信息，即使在其业务生命周期中的价值已经降低，但是还有可能被反复访问，则这一类的信息也应存放在高效存储介质中以备使用。

公司可以使用 ILM 使信息管理与业务目标保持一致，根据信息在不同阶段所展现的价值来管理数据，并将其与分层存储系统结合起来以最大程度地利用信息。

2.2.3 其他分层存储技术

自动分层技术即依靠策略、数据访问模式来对数据进行迁移的分层存储方式，将被频繁访问的数据放在固态驱动器中以达到效率最优化[38]。类似的技术方式还有存储缓存分层，即通过识别热数据放入高速缓存的方式，来提高磁盘驱动器的整体的存储表现。这种类型的技术也是当今研究的热门，即通过识别出数据的冷热来将其存放在高效存储介质中，其中的关键技术就在于冷热识别算法，这也决定了整体的性能表现。

随着云存储技术的出现和发展，在传统的分层基础上还出现了云层存储[39]，企业或个人向各云服务提供商租用公有云或者搭建自己的私有云服务器，并用于数据存储。公有云上的云层存储通常都提供自动化监视和报告的管理功能，并可从本地集群自适应的向云存储迁移并分层。云层存储的结构图如图 2-10 所示 。

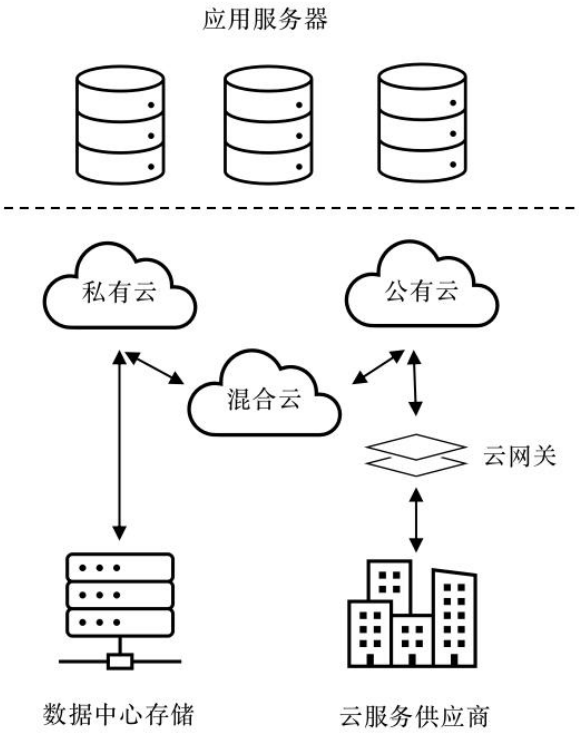

图 2-10 云层存储结构图

如上图所示，云网关被用于直接对数据块和文件进行访问，而云被用作自动备份层或手动归档层。云层存储的最终模式应该是出现混合云，它将使用通用数据管理协议将公共云和私有云结合在一起，使得数据在云上也按需分层存储，提升企业对云资源的利用效率。

2.3 本章小结

本章节介绍了本文存储机制研究和优化所基于的相关理论知识和技术。首先介绍了在操作系统中常见的缓存替换策略 LRU 与 LFU, 以及这两个策略衍生出的优化策略，这些优化思想为本文冷热数据判定策略提供了方向。接着介绍了分层存储技术的发展历史，回顾了 HSM 与 ILM 的发展过程，以指导后文冷热数据存储方案的设计。

第三章 冷热数据判定策略研究与实现

本章将分析传统内存置换算法以及其他数据温度判定方法存在的问题，围绕问题进行分析并提出本文所采取的冷热数据判定策略，并基于策略大致介绍整体冷热数据存储的框架结构，为后文存储方案详细设计进行铺垫。

3.1 冷热数据判定问题分析

关于冷热数据问题的研究，其方法可类比于操作系统的缓存替换策略，如OPT、LFU、LRU 等，这些算法都基于数据某一方面的特征作为依据来评估页面驻留在内存的价值。OPT 作为一种理想的缓存替换策略，它需要知晓每个数据下一次被访问的时间，并将时间间隔最大的数据进行替换，虽然这能最大化的利用内存，但未来时间数据的访问序列很难以进行预测，因而只可作为一种替换策略的评估标准。LFU 是基于数据历史访问频率的缓存替换策略，它认为历史访问频率最高的数据更有可能再次被访问，类似设计的算法还有 LRU，它是基于数据最后访问时间的缓存替换策略，即认为越近被访问到的数据其留在内存的价值越大。这两种算法也都只基于数据的一方面特征来设计策略，但实际数据的热度是由多方面去决定的，如数据的最后访问时间、近期访问频率等，都是重要的数据热度判断指标，在数据库中的数据访问特征更是需要综合考虑。

为在数据库中进行数据温度判定，本文对传统缓存替换策略进行了一定调整并弥补其所存在的不足，提出了一种结合数据多方面特征的冷热数据判定策略。具体解决思路为将数据访问时间与数据访问频率相结合，并加上数据间存在的联系，来共同构建数据温度判定模型。每个数据均根据访问时间、访问频率、数据关联性，为其标定一个实时温度，再基于这个温度将数据定为冷数据或者热数据。在最终预期的效果，本策略应保证能适应多种数据访问模式，较准确的对冷热数据进行区分识别以便于后续对数据进行相应存储方案设计，且不会带来过多的额外开销。下面将从冷热判定策略的准确性与高效性两方面进行详细需求分析，提出设计目标。

在准确性方面，本文需要提出能综合考虑数据访问时间、访问频率、关联性的多维度冷热判定策略，该策略直接决定了冷热数据的区分以及后续存储方案，也是最为核心的部分。它需要满足将数据的访问时间、访问频率、关联性这三个特征采用合理的方式进行量化，并整体构建数据温度模型，能够识别过滤出高频、较新的数据，且数据温度会随系统时间推进而动态变化，尽量避免较老但高频数

据长期判定为热数据、较新数据进入系统即判定为冷数据的现象。本文所提出的策略应对传统 LFU、LRU 策略难以应对的场景有一定的解决力，同时保证保有这些策略原有特征。

在高效性方面，本文需要获取数据的多维度特征进行综合评估，且保证每个数据的温度都可随时获取，以便判断冷热数据的程度，因此计算方面开销不宜过大。传统的缓存替换算法中，很多算法都需依赖特定的数据结构进行实现，但其所针对的是数据频繁换入换出的应用场景，具有很强的实时性，而在本文的数据库应用当中的冷热数据判定对实时性要求略低。因此，本文判定策略的设计目标应为在数据发生变化或需进行冷热数据分离时，能够高效的获取每条数据的温度，保证系统整体的正常运行。

3.2 冷热数据判定模块

在数据库中，每条数据均可包含一个时间戳字段，时间戳可用于标明数据插入数据库时的时间或是上一次访问的时间，本文中冷热数据判定策略正基于数据的时间戳进行模型建立。下面将会先总体介绍冷热数据存储的系统框架，以便于理解冷热数据判定模块在整个系统中所处位置，然后具体介绍冷热数据判定模型建立的理论依据以及其具体应用方式。

3.2.1 系统总体框架设计

本文所建立的是基于数据温度的存储方案，最终目标是对存储于本系统中的数据进行冷热识别，并根据其温度分别存入相应的数据库当中，以提高空间利用效率和计算效率。因此在方案中会存在冷库与热库两个概念，热库用于存放温度判定为热的数据，冷库则相反，冷库与热库的数据库相关内容将在下一章进行讨论。冷库与热库之间的数据会互相进行置换，而其置换的时机与数据选择则由冷热数据迁移模块独立负责，此模块也将在下一章进行具体介绍。冷热数据判定模块则建立在两个数据库之外，进入系统的数据优先落在热库当中，并由冷热数据判定模块为其计算一个初始温度，有查询引起的数据访问也将通过冷热数据判定模块，对被访问数据的温度进行调整。系统的整体架构图如图 3-1 所示：

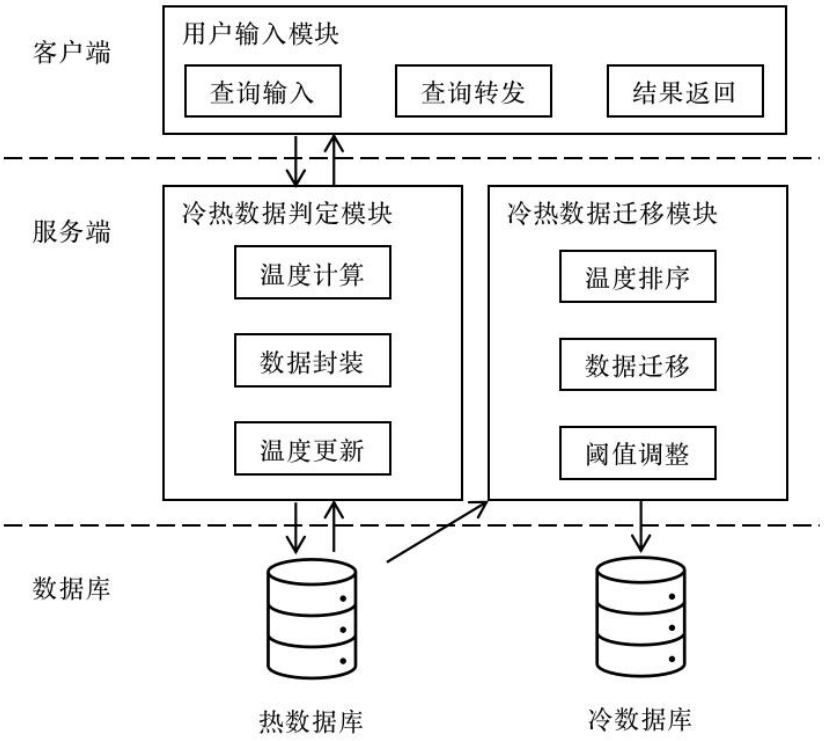

图 3-1 冷热数据存储系统架构图

如上图所示，整个系统由客户端、服务端与数据库三部分组成，其中用户输入模块作为系统的客户端，主要负责对用户所输入的查询转发至服务端的模块，并将查询结果返回至用户。服务端分为冷热数据判定模块与冷热数据迁移模块，来实现对冷热数据的识别与存储，冷热数据判定模块会根据客户端输入的查询为数据封装温度相关信息并存入热数据库中，其主要功能包括数据温度计算、数据封装以及数据温度更新；冷热数据迁移模块则根据热数据库存储的温度信息，对冷热数据进行划分并完成迁移存储，主要包括温度排序、数据迁移和阈值调整，关于迁移模块的详细设计将在下一章具体阐述。

在本文中，系统对于冷库与热库的原生数据库并不会进行修改，而是在其之外独立增加冷热判定模块和冷热调度模块两个模块来支持冷热数据的分离存储。本章重点在于介绍冷热判定模块，如上文所说，数据的插入和修改操作均会导致数据温度的变化，而数据温度的计算必须依赖于冷热判定模块。冷热数据判定模块的主要功能是根据数据包含的时间戳、关联数据以及原有温度信息，对数据温度进行计算，并将这些信息更新并重新封装至原始数据当中，其他的数据插入与修改操作还是保持热库原生的操作方式，不会修改数据原有的信息。对于数据的查询操作，本模块也会对查询命中的数据进行温度更新，但不影响查询结果。对于数据的几种基本数据库操作，冷热数据判定模块的交互逻辑如图 3-2 所示：

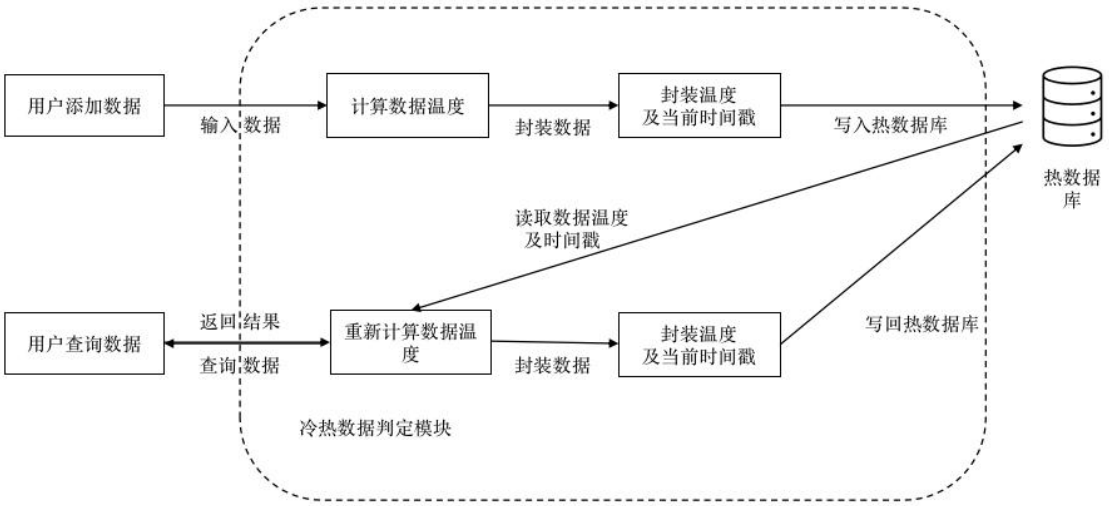

图 3-2 冷热数据判定模块的交互逻辑框图

3.2.2 冷热数据判定策略设计

上一节介绍了冷热数据存储系统的整体框架以及冷热数据判定模块在整个系统中所起到的功能，本节将介绍冷热数据判定策略的设计。

关于冷热判定模型的建立，其最主要的依据是数据访问的时间特征，因此需建立对时间变化敏感的模型。牛顿冷却定律是用于描述一个高温物体在低温环境中，向周围传递热量并逐渐冷却的规律，它表明了物体温度的变化速度与环境的温差成正比关系[40]，即物体温度高于环境越多，其温度下降速度越快。该定律的公式表示如下：

$$T'(t) = -\alpha(T(t) - H) \qquad \text{公式 (3-1)}$$

上式中 T 表示物体温度，α表示物体温度变化速度，H 表示环境温度，将 T 对 t 进行求导得到的即为温度变化速度。若对该公式进一步求解，可以得到如下推导结果：

$$T(t) = T_0 e^{-\alpha(t-t_0)} + H(1 - e^{-\alpha(t-t_0)}) \qquad \text{公式 (3-2)}$$

如求解结果所示，牛顿冷却定律反映了物体温度随时间推进是一个不断衰减的过程，这也可作为随时间衰减模型的基本理论框架。本文中数据的冷热程度正是一个随时间衰减的过程，最新访问的数据应具有较高的温度，而随时间推进其温度应逐渐降低，因此可考虑将牛顿冷却定律推导公式作为基础来建立模型。

相比于真实物理环境中的物体，数据库里的数据仅有时间这一特征，而环境温度信息仅可通过计算全部数据的平均温度加以代替，从实际数据温度的影响因

素来考虑，这样的代替是没有意义的。因此本文提出的冷热判定模型中，数据的温度仅与其最近访问时间与访问频率有关。基于上述考虑，将公式 3.2 做适应性的变形，并加上访问带来的温度上升变量 W，可以得到如下公式：

$$T(t_n) = T(t_{n-1})e^{-\alpha(t_n-t_{n-1})} + W(t_n) \qquad \text{公式 (3-3)}$$

其中，$T(t_n)$表示t_n时刻数据的温度，α为预先设定的温度衰减系数，$W(t_n)$为一个离散函数，当t_n时刻数据被访问，则 $W(t_n)$为估计温度上升数值，反之为 0。

上述数据温度为数据自身的访问所带来的温度变化，除此之外，数据与数据之间也存在着关联性，即数据在访问规律上呈现出的特征，尤其在数据循环访问的模式下，数据有着明显的关联关系。例如存在“ABCDDCBA”这样的访问序列，当“ABCD”已经被访问时，系统若能识别并记录下各数据间的联系，在下次同样数据再次被访问时，将与之关联的数据温度也进行一定预热。当再遇到“DCBA”的后续访问时，如“D”被访问时，对上次访问中“D”的前序数据“C”进行预热，则可以保证关联数据都存在一定的热度，该访问序列即可保证均在热库中被高效访问。对于每个数据记录其关联数据，仅记录上一次访问的前序数据，当数据再次被访问时，若其前序数据改变，则再影响原关联数据后，其关联数据也会进行更新。数据温度关联更新过程如图 3-3 所示。

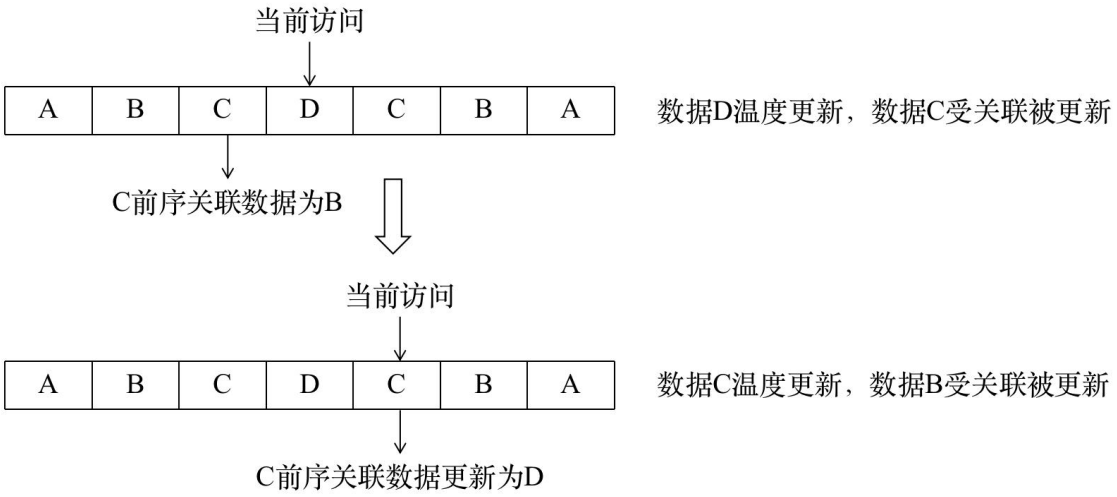

图 3-3 数据温度关联更新示意图

这样的数据关联性温度的影响，就类似于物体间的温度传递，两个相邻近的物体中有一个物体的温度上升，则热量也会向周围物体进行传递。同样基于温度变化公式 3.2，假设数据 A 被访问升温后的温度为T_A，而与数据 A 相关联的数据 B 的温度为T_B，此时数据 A 相当于环境温度，而数据 B 则是随环境温度变化的对象。假设数据间传递仅经过一个时间单位，那么可以得到如下公式：

$$T_B' = T_A + (T_B - T_A)e^{-\alpha}$$ 公式（3-4）

其中，T_B则代表数据 B 受数据 A 升温影响之后数据温度。由于这样的温度传递带来的变化较小，而发生二次传递所导致的数据温度变化会更加微弱，考虑节省计算成本，在实际应用时仅考虑被访问数据直接关联数据的温度变化。且本次数据温度的变化并不等同于数据被访问，因此所记录的数据上次被访问时间并不会随之更新，而是在下次被访问时直接将当前数据温度当做上次访问后的数据温度进行计算。特殊情况下，当数据的前序访问数据为自身时，数据不会对自身的温度再次累加，且会将原先的关联数据改为自身，这样做是为了避免当一个数据被连续重复访问时，其关联数据温度反复提升的问题，而实际此时数据的访问模式已经发生了改变。

3.2.3 冷热数据判定策略实现

上一节介绍了冷热数据判定策略中数据温度的计算方式，数据温度由数据自身的访问更新与关联数据的传递更新两方面来决定。数据温度计算的具体流程如图 3-4 所示。

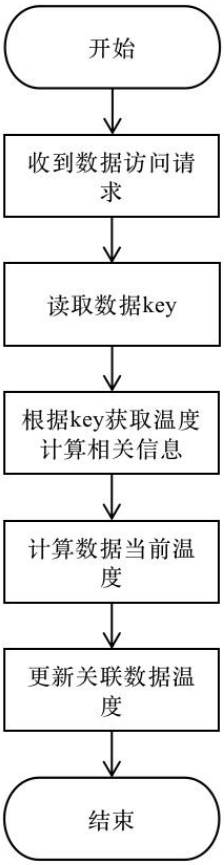

图 3-4 数据温度计算流程图

如上图所示，数据是以 hashmap 的结构进行存放的，其具体结构在下一章将进行介绍。当系统受到访问请求时会在热数据库中读取到具体数据，接着将上次存放的数据相关信息解析得到用于后续计算。流程中包含两个关键函数 calTemp 和 updateTemp，这两个函数分别实现了公式 3-3 与公式 3-4，用于数据本身的温度更新以及关联数据的温度更新。在实现过程中涉及到一个指数运算的过程，为了加快指数运算的速度，采用快速幂的算法加速了计算过程，降低了整个流程的开销。

3.2.4 冷热数据判定策略分析

基于前两节的描述，数据温度的计算方式已经介绍完毕，本节将基于这些计算公式，从理论上分析在传统常见的几种数据访问模式下，本文所提出的冷热数据判定策略与传统 LRU、LFU 等算法间的效果比较，以及针对传统算法所面临问题是否可以得到解决。

前文提到 LRU 算法对于频繁访问的数据识别效果较差，比如在“AAAABCD”，数据 B、C、D 的优先级均要高于数据 A，而实际上数据 A 在整个访问序列中占据了大部分的访问量，在未来仍存在较高可能被访问。为了便于计算，假设公式 3.3 中的温度上升数值 W 为 1，物体温度变化速度的比例系数α为 0.05。数据温度计算示例如图 3-5 所示。

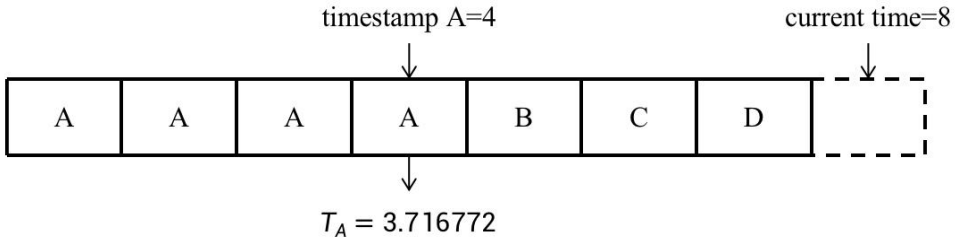

图 3-5 数据温度计算示意图

如上图所示，数据 A 保存了最后次访问后的时间戳与温度信息。根据公式 3.3 对数据 D 访问结束时刻的数据温度进行计算，可以得到T_A为 3.199055，其他数据温度类比与数据 A 的方法计算可得T_B为 0.860708，T_C为 0.904837，T_D为 0.951229。可以发现此时数据 A 仍保持着最高的温度，并不会被最近访问的数据 D 所超过，且在数据 B、C、D 间，仍保持着越近访问的数据温度越高的规律，符合 LRU 算法的预期效果。

而对于 LFU 算法，其所面临的主要问题是历史高频访问数据会长期保持高温难以被置换，即不存在有效的降温策略。同样以上述计算方式，在访问序列“AAAABCD”中，数据 A 在该序列访问结束时刻仍保持的最高的温度，若将访

问序列继续延长，经过计算，数据 A 的温度会在 D 之后的第 24 个访问时间单位后降温至 1 一下，即在 D 之后访问的第 24 个数据温度将会超过 A，A 将不再具有最高优先级，且若在此时刻之前有其他数据被多次访问，优先级都有可能超过 A。由于本文策略对温度的计算会随着时间衰减，有效避免了历史高频访问数据的污染问题，且由于指数函数的特征，衰减速度会随时间推移不断增加，也符合数据被访问的概率分布，但也保持了 LFU 算法对数据访问频率的考虑。数据温度下降示意图如图 3-6 所示。

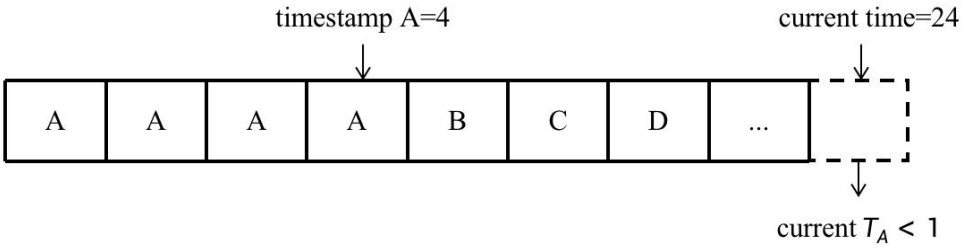

图 3-6 数据温度下降示意图

以上均是针对顺序非循环访问下的策略分析，对于循环访问模式下的数据温度则需加入数据关联性因素进行计算。假设访问序列为“ABCDEDCBA”，当时数据 D 被二次访问时，其上一次所关联的数据 C 会得到一次升温，这可能导致后续温度计算时顺序发生变化。关联数据温度更新如图 3-7 所示。

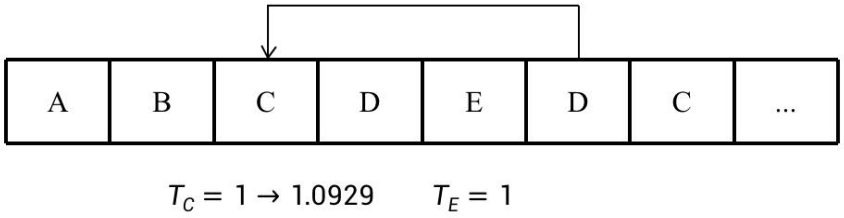

图 3-7 关联数据温度更新示意图

如上图所示，根据公式 3.4 计算，此时数据 D 的温度为 1.904837，数据 C 原先记录的温度为 1，则受影响后温度变为 1.0929。若计算此时所有数据温度，则可得到数据 E 温度为 0.904837，而数据 C 温度为 0.940667，可以发现由数据 D 被访问所带动的数据 C 的温度已经超过了更近时刻访问的数据 E 的温度，若按优先级进行排序，则数据 C 将更有可能被识别为热数据，即在循环访问的模式下，本策略可以对数据访问模式具备一定适应性。

综上三种数据访问场景下的理论分析，本文冷热数据判定策略在保持了 LRU 与 LFU 的原有识别能力外，还解决了循环访问模式下的策略失效问题，具备较优

的识别稳定性。

3.2.5 冷热数据判定策略应用

在前几小节介绍了基于牛顿冷却定律推导公式在数据库中加以变形的冷热判定模型，本小节将介绍该模型如何在数据库的几种基本操作中进行应用，以及该模型如何进一步应用于冷热数据的判定和比较。数据库的基本操作包括增删查改，这几种操作都会带来数据自身或者周围数据的温度变化，下面对他们分别进行介绍。

当发生插入操作时，全新的数据加入到数据库中，应赋予其环境温度值作为初始值。在上文的讨论里说到，环境温度在数据库中没有现实意义，因此在本文中，直接将公式 3.3 中的温度上升变量 W 作为数据插入的初始温度。

当某数据被修改或者访问时，本文将其视为同一类型的数据温度变化，会将其原先温度根据访问间隔时间进行冷却后，再进行一次升温得到最新的温度。参考公式 3.3，假设上次访问时间为t_{n-1}，当前时间为t_n，数据记录温度为 $T(t_{n-1})$，冷却系数为，则经过$t_n - t_{n-1}$时间后数据冷却为 $T(t_{n-1})e^{-\alpha(t_n-t_{n-1})}$，再加上温度上升变量 W，即可得到最新的温度。

当某数据被删除时，其自身温度并不需要再被考虑，删除操作带来的仅是其关联数据温度的变化。被删除的数据，其温度相当于变为 0，参考公式 3.4，假设关联数据上次温度为 T，则受被删除数据影响后的温度为 $Te^{-\alpha}$，若该数据本身温度已经为 0，则不再进行计算。

基于上述的描述，系统即可在任意时间得到某一数据的具体温度，数据间的温度比较也可直接基于此数值，温度较大的数据为较热数据，即更有可能被访问的数据，反之则为较冷数据。基于此温度如何进行冷热数据的划分，将在下一章存储方案中具体介绍。

3.3 本章小结

本章对全文所要实现的冷热数据存储架构进行了简要介绍，随后详细介绍了冷热数据判定模块在整个系统中所发挥的作用及其核心判定策略的设计与实现。

在设计目标方面，首先分析了传统缓存替换策略常用算法的优劣，找出其可能存在的不足，针对这些问题并加以本文应用场景的分析，明确了本文冷热数据判定策略所需满足的准确性与高效性要求。

随之在介绍完整存储系统架构的基础上，划分出了本文重点实现的冷热数据判定与冷热数据迁移两个模块。在第二节中详细介绍了冷热数据判定策略所基于

的理论基础，以及在应用场景下的数据温度模型建立，最后将模型的具体的应用方式进行了介绍，为下一章对冷热数据存储方案的设计与实现进行铺垫。

第四章 冷热数据存储策略研究与实现

在第三章中对本文的冷热数据存储方案总体框架设计和功能模块划分进行了阐述，并详细介绍了冷热数据判定策略。本章将先分析冷热数据的特征，提出存储方案的设计目标，根据设计目标对冷热数据库进行选型。然后基于上一章描述的判定策略，对冷热数据迁移模块进行详细设计与实现。

4.1 冷热数据存储策略分析

对冷热数据进行判定识别的最终目的，即是根据数据温度的不同，将其存放在不同的存储介质上，以提高优质存储空间的利用率。本节将先对冷热数据的特征进行分析，再根据其特征提出相应的存储需求，以求相比混合存储，能够达到优化目标。

4.1.1 冷热数据存储特征

在很多实际应用场景下，像是论坛、电商、微博等，其数据访问的特征都呈现冷热分布，只有小部分的数据会被高频的访问，而大部分数据只有少量的访问量。为了高速响应用户的访问请求，服务商需要购置大量的存储设备来存放这些数据，且需保证数据可以快速获取。存储介质的选型直接决定了数据读取的速度，像是基于闪存或者 DRAM 的固态硬盘相比于一般的机械硬盘，其随机读写性能就高出很多，但同时其价格也会更加昂贵。服务商若全部购置高性能的存储设备，自然能保证更加优质的服务提供，但成本也会难以估量，且其存放的数据，可能仅有小部分数据被高频访问，浪费了大部分的存储性能。

因此，针对冷热数据相应的特征，应选择适当的数据存储介质，以优化存储性能并降低存储成本。对于热数据，它往往数据量较小，但需要被高频的访问，在存储时需优先考虑其读写性能；而对于冷数据，它具有庞大的数据量，可是往往对读写性能要求略低，在存储时更应考虑存储量级的优化以及数据的可获取性。

4.1.2 冷热数据存储需求分析

对于冷热数据不同的存储特征，本文从存储基本需求、存储空间需求和读取性能需求三个方面对其进行分析：

（1）存储基本需求方面，不论采用何种存储介质存放数据，所有数据的数据完整性都需要有保证，即需满足数据的精确性与可靠性，在存储系统正常运行的

状态下，不允许出现数据丢失、数据错误等问题。

（2）存储空间需求方面，对于热冷数据的存储都应在保证满足读取性能的前提下，尽量的优化存储结构，以在有限的存储空间下，存下更多的数据。其中，由于冷数据的读取性能要求低于热数据，在冷数据存储时更应注意结构优化，保证读写可靠性的前提下，尽量缩小存储空间开销。

（3）读取性能需求方面，热数据为频繁访问数据，因此需保证热数据能够高效的读取，而冷数据需关注其读写可靠性，保证不会出现数据读取错误、写入失败等问题。

4.2 冷热数据库设计

上一节分析了冷热数据的存储需求，为满足其对应的需求，选择合适的存储介质与数据库直接决定了系统整体的成本与存储性能。在工业界，绝大多数互联网企业面对的都是 PB 级及以上的数据量，且总体数据量还在不断的增长，日增都可高到数百 TB 甚至更高。

面对这样庞大的存储需求，业界通常会将数据按业务线进行划分，且将数据分在线上数据库与线下数据库存储。这样的做法带来了多方面的优势，一是减小业务量级，使得每条业务线所需数据更加明确，不用每条业务的读写均落到总数据库上，提高每条业务线的运行效率；二是提高了数据容灾性，企业的全量数据由于其数量庞大而难以备份，将数据根据业务拆分成多个线上数据库后，可以更加轻量的为每条业务线进行数据备份管理，即使出现了事故导致线上数据库的失效，也可由备份数据库继续工作，保证业务的正常运行。

对于这样的线上线下数据库划分，线上数据库由于需要强实时性来支持与业务服务端的交互，往往采用 SSD 作为存储器。SSD 具有较好的随机读写性能，正适用于作为搭建线上数据库的存储设备。而线上数据库的选型可根据实际存放的数据特征，来选择合适的数据库进行存储，如常见的 MySQL、LevelDB 等均被用作线上数据库来支持业务。

线下数据库负责的是整个企业或者某一部门的全量数据存放，如用户的所有历史信息、行为等，由于其数据量的庞大，并不会与业务直接相关，往往是需要在历史数据的基础上做一定的数据分析或者构建一些过滤器，再将这些结果存回线上数据库来支持业务运转。因此在数据读取方面，线下数据库的读取性能要求并不高，仅需满足企业内部所需的数据分析或信息提取即可，它通常是使用相对成本较低的 HDD 来支持海量数据的存放，并在其基础上搭建如 HBase 等支持大规模数据读写的分布式数据库，来保证一定的存储性能。企业常见数据交互图如图 4-1

所示 。

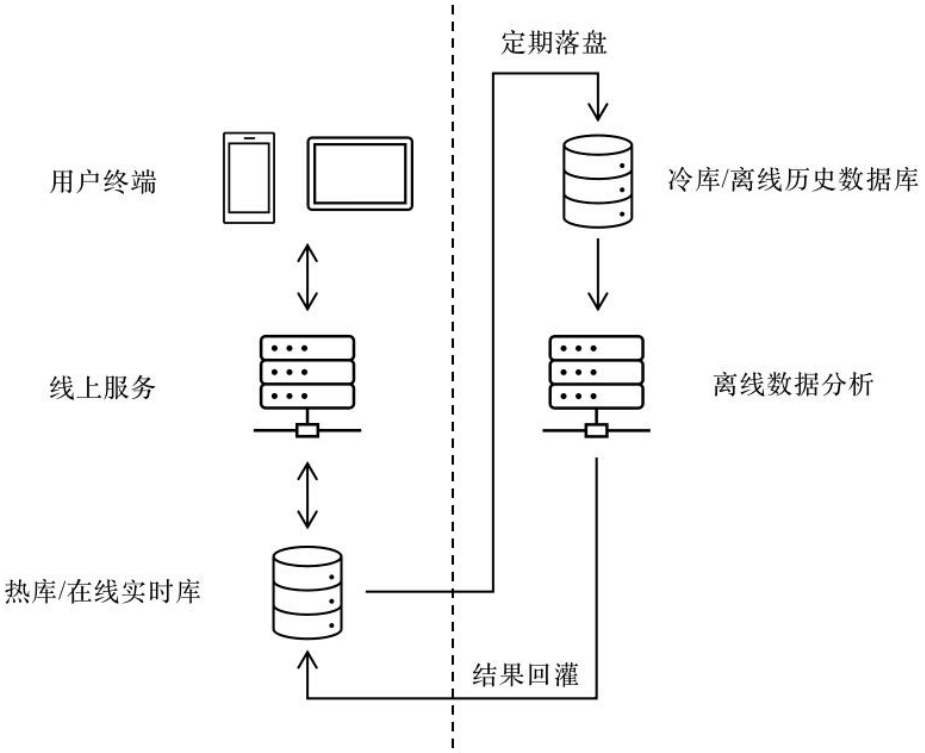

图 4-1 企业数据交互示意图

如上图所示，与用户直接交互的是线上服务器，线上服务器承载了各业务线的在线流量，相关的数据均在读写性能较优的线上数据库进行存取，保证了业务的实时性需求，给用户更加及时的信息反馈与优质的用户体验。线上数据库应存储空间限制，并不能够存储全部的用户全量历史数据，因此线上数据库的数据会按一定的时间周期向离线数据库进行刷盘，也为新的在线流量留出空间。离线数据库由于读写性能弱于线上数据库，数据刷盘的过程也需要一定的时间进行。为了同时进行数据分析以及线上同步，通常也会做多副本处理，读副本用于做数据分析，而写副本用于承载线上数据，多个副本间在线上刷盘周期空隙间进行切换交替使用。

在本文中，冷热数据库的需求与业界的线上线下数据库需求相仿，完全可以参照业界数据库的选型方式来进行实现，而本文冷热数据库的不同之处，在于热库数据的区分方式比线上更加精确，所基于的存储介质速度更快，冷热数据库间的数据迁移更加频繁，但同时存在热库空间限制更小的问题。业界线上数据库所存储的数据一般以时间为依据，存放一天或几天内的全量数据，这样的划分方式导致驻留在线上数据库的数据也不完全为热数据，且数据量也相对更大。本文的热数据库将根据上一章的冷热判定结果，来做更加精确的筛选，也可根据需要来控

制热数据库中的数据量，以求得对热数据更高的读写性能。

基于上述分析，热数据库的选择更应侧重于少量热数据的读写性能，而存储数据量足以适应数据访问模式即可。内存作为性能仅次于高速缓存的存储介质，其空间大小完全足以存放存在局部性访问特征的数据，因此本文采用 Redis 基于内存的数据库作为热数据库，保证热数据最高的读写性能。冷数据库在选择上主要需考虑海量数据的存储能力以及良好的扩展性，HBase 作为分布式数据库，对海量数据存储有较好的支持性，因此本文采用 HBase 数据库作为冷数据库，为大量的冷数据提供存储。

4.2.1 热数据库设计

Redis 是一个高性能的 key-value 非关系型存储系统，它基于 C 语言进行开发并提供开源免费使用。Redis 支持传统键值数据以及 String，list，set，zset，hash 和其他数据结构，以支持更复杂的存储需求，并基于底层数据结构根据存储对象编码对数据进行了优化存储，以进一步压缩数据存储空间。Redis 数据库的高性能主要是依托内存的高速读写效率，相比于依赖于磁盘存储的传统数据库，它不存在 I/O 上的开销，且在基于单线程的 Redis 实例中，不存在上下文切换以及多线程间通信所导致的开销。使用 Redis 作为热数据库，将热数据都存储在内存中进行访问，能够最大程度的提升其读写性能。

对于 Redis 存储所采用的数据结构，因为除需要存储原始键值对外，还需要对数据访问时间戳、数据温度和前序访问数据三个属性，因此考虑采用 hash 的数据结构来存储。其具体存储结构如图 4-2 所示：

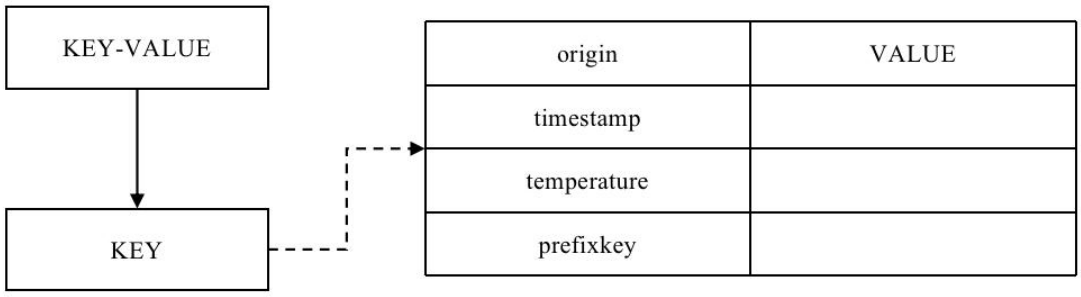

图 4-2 Redis 中 hash 结构存储示意图

其中 origin 与 VALUE 即为原始的 KEY-VALUE 数据；timestamp 为数据上次被访问时间戳，时间戳的记录由模块以系统时间为基准进行记录；temperature 为数据上次访问时更新的温度，即温度是与时间戳相匹配的，以便于在下次数据访问时基于两者计算数据最新的温度；prefixkey 为当前数据访问的前一个数据的 key

值，用于表征数据间的关联，可通过 prefixkey 还原出最近的数据访问序列。

4.2.2 冷数据库设计

HBase 是一个高性能、易伸缩的分布式列式数据库，HBase 可将大量单一存储能力较弱的存储设备有效组织管理起来，搭建高可用性存储集群。传统数据库存在单机面对大数据量难以存储、备份机制不完善、数据访问速度随数据量增长而下降等方面的问题。针对这些弊端，HBase 支持线性扩展，通过扩展存储节点的方式突破了单机存储容量受限的瓶颈，从而支撑海量数据的存储；HBase 依托 HDFS 进行数据存储，其将数据备份在多台节点上，保证了分布式环境下的数据安全性；此外 HBase 虽然存储在磁盘上，但通过 zookeeper 协调查找数据，保证了较高的访问速度。

将 HBase 作为冷数据库，能够支持大量的数据存放。相对于热数据库，其表中所存储的信息仅包含了数据的原始信息，即键值对，而无需存放时间戳和数据温度信息，具体原因在下一节将展开介绍。但 HBase 原生即支持基于时间戳的多版本存放，在发生冷热数据迁移时亦可提供多版本信息供于查阅。HBase 具体表结构见图 4-3 所示：

RowKey	TimeStamp	CF1	CF2	CF3
KEY	timestamp1	VALUE1	-	-
	timestamp2	VALUE2	-	-
	timestamp3	VALUE3	-	-

图 4-3 HBase 数据存储示意图

4.3 冷热数据迁移模块

上一节针对冷热数据的特征，对比了工业界的线上线下数据库组织架构，进行了数据库选型。回顾第三章提到的整个系统的存储框架，目前已经完成了冷热库选型，冷热数据判定模块也可以将每条数据标定一个数据温度，下一步数据将如何选择存放冷库或者热库也直接决定了存储性能的发挥，本节将介绍冷热数据迁移模块的设计与实现方式。

4.3.1 冷热数据迁移策略

在传统数据库中，尤其是基于 LSM 树结构的数据库，都存在数据从内存向磁盘迁移的过程，此过程或被成为持久化过程。像是 LevelDB 数据库在内存中的 memtable 也会根据所设定的存储大小阈值，达到阈值之后向磁盘进行落盘，转换成 sstable。Redis 作为内存数据库，具有 RDB 机制，可将内存数据定期保存到磁盘，以防止由于断电和其他事件而导致内存数据丢失。这些数据库的共同特点，就是他们都采用了分级分层的存储思想，这样数据迁移是不可或缺的一步工作。同样在本文的冷热数据存储系统中，冷热库其实相当于是整个系统的两层，因此也需要合适的迁移策略，确保热库始终有足够的空间来存放新插入的热数据。

在分级存储中，现在普遍采用两种数据迁移方法是基于存储空间的高低水位法和基于数据访问率的缓存替换迁移算法[6]。在高水位和低水位方法中，要能够迁移数据的先决条件是磁盘上具有足够的空间。为了在适当的时间启动迁移过程，系统需要实时监视磁盘空间饱和度。此方法在进行数据迁移时仅考虑了存储设备的剩余空间，而疏忽了数据的访问特征。而缓存替换迁移算法采用一系列的方法移除无价值的数据，其优点是考虑数据的访问特点，有利于存储的总体优化使用，但是会存在额外的磁盘开销，且很依赖于特定的数据结构进行实现。

本文在第三章的冷热数据判定策略中，已经将数据的访问特点进行建模，对每条数据均生成了数据温度以表征数据访问特征。在此基础上，本文冷热数据迁移模块拟采用高低水位法对热数据库进行饱和度监控，同时兼顾了数据的特征以及存储设备的存储状态，弥补了单独采用高低水位法进行数据迁移的缺点，让热库一直保持高效且可用的状态。高低水位法监测示意图如图 4-4 所示：

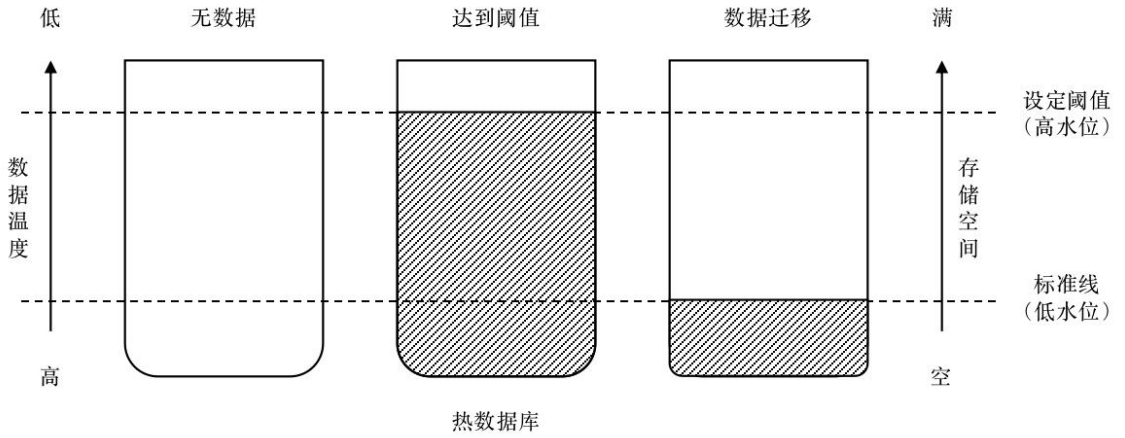

图 4-4 高低水位法示意图

如上图所示，在热数据库存储数据中存在两个由系统设定的阈值，分别标示

了热数据库理论允许存放的最大数据量以及热数据库在经过第一次数据迁移后所保持的最低数据量。由于热数据库是易失性的，每次系统的重启都会导致热数据库中的数据丢失，因此在整个存储系统运行初始状态下，热数据库都是从空状态开始记录数据。随着数据访问量的上升，热数据库空间使用率会同步上升，而热数据库状态由迁移模块实时监控，一旦使用率达到最大数据量阈值则会发起迁移流程，将数据量控制下降到最低数据量，再继续累积数据不断重复。

以上为理想情况下的数据迁移策略，但是考虑到热数据库为内存数据库，即发生如断电等事故时，若数据仅在热数据库中存储，则该部分数据会丢失难以恢复。为应对内存数据丢失问题，除了在高水位阈值发生数据迁移外，还需要结合Redis 数据库原有的 RDB 与 AOF 持久化机制保证数据在断电情况下的恢复。Redis 数据库的 RDB 机制，原理上是将某时刻的所有数据以快照的形式保存在磁盘上，并生成二进制文件。原生 Redis 提供了三种 RDB 触发机制：save、bgsave 以及自动化。save 需要客户端下达命令，在执行期间 Redis 服务器处于阻塞状态，当内存中数据量过大时，会导致客户端操作延迟过高；自动化的方式则需要通过配置文件来进行，以 m 秒内存在 n 修改为触发机制，但由于冷热数据场景下数据访问模式无法确定，此方式并不是最合适的方案。bgsave 的方式与 save 类似，但是它是通过创建子进程的方式异步执行快照操作，因此阻塞仅发生在创建子进程的时间，而创建快照的同时客户端可以正常接受请求。在 Redis 中执行 bgsave 过程如图 4-5 所示。

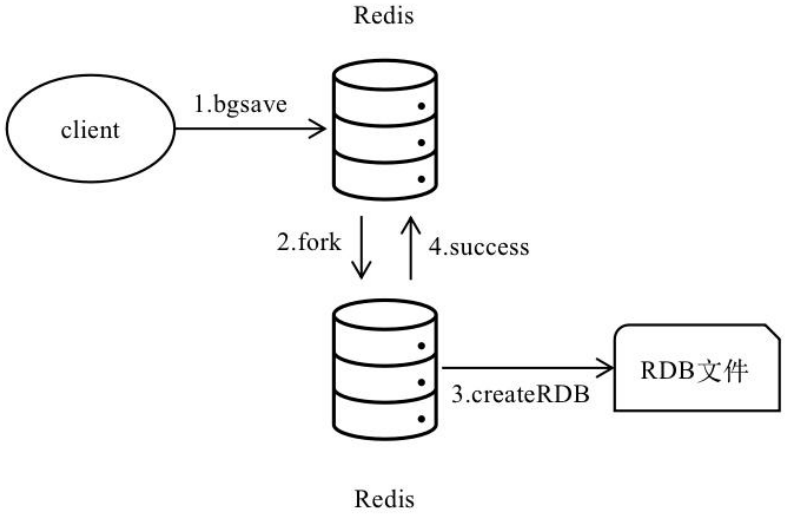

图 4-5 Redis 执行 bgsave 流程图

通过 bgsave 获得 RDB 文件后，即可在断电后将数据库恢复到记录时的状态，但在异步创建 RDB 的过程中，服务器所接受的数据修改请求无法避免丢失。Redis

数据库的 AOF 机制，是以类似日志追加的形式来对操作进行记录，这种方法具有更强实时性和数据完整性保障，但相应的持久化存储开销以及数据库的写 QPS 会受影响。在冷热数据分离存储的场景下，由于热数据库会被迁移至冷数据库，热数据库中数据量始终不会过大，因此采用 RDB 的方式来保证数据不被丢失更加符合要求，执行 bgsave 的时机也与热数据库达到迁移阈值相对应，保证了未发生迁移的热数据的数据安全性与完整性。

热数据库所需的两个阈值可以直接以固定值进行设定，考虑巴莱多定律将系统的初始阈值分别设定在热数据库容量的 20%与 80%，这样足以保证热数据库的稳定运行，但这种方式欠缺对数据访问模式的考虑，比如在热数据访问均匀分布的情况下，若设定的存储阈值过低，则会导致一部分仍保持较热的数据被迁移到冷数据库中，进而导致此部分数据可能会快速回温至热数据库里，发生频繁的数据迁移。为应对不同场景的需要，阈值应根据数据访问模式而动态调整。本文为动态调整阈值范围，在每次执行数据迁移时，会对热数据库中数据温度进行记录，并对这些数据进行均匀采样，以识别被迁移部分数据温度的分布情况。理想情况下的数据温度采样应服从正态分布，如图 4-6 所示。

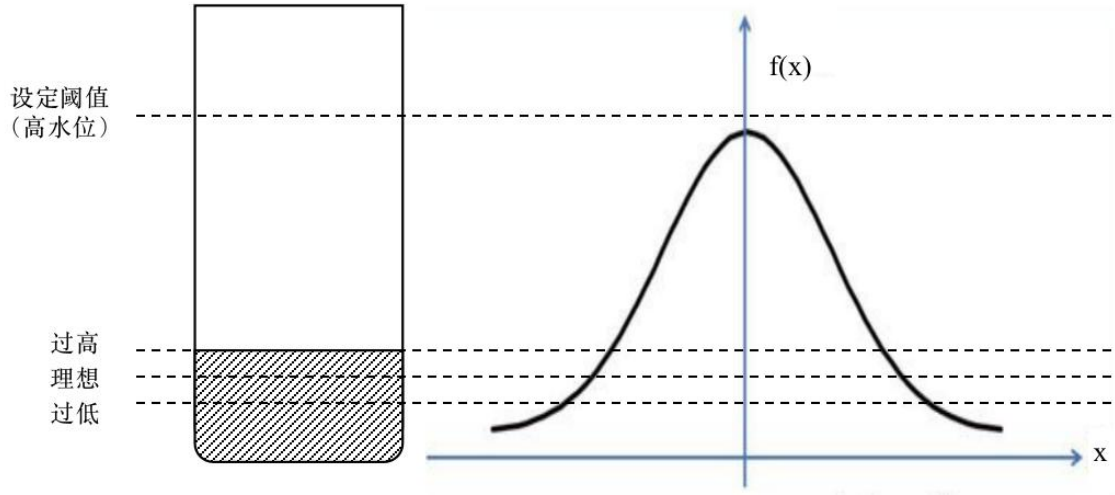

图 4-6 数据温度分布图

如上图所示，理想的标准线阈值应处于数据分布的 2σ与 3σ之间的水平，保证大多数的访问都落在标准线阈值一下的数据上，而若所设定阈值所对应的数据温度在分布中水平过高或者过低都是不理想的阈值划分。通过采样的数据温度即可对当前阈值在数据中所处位置进行识别，比如图中过高的阈值线，其周围采样得到的数据温度间跨度应该较大，这表明当前阈值内包含的数据已存在部分的冷数据，此时应将标准线的阈值适当调低，将更多的数据迁移到冷数据库中。相对的

若阈值线周围采样得到的数据温度间跨度很小，说明在阈值周围的数据访问频率较为均匀，此时应将标准线阈值进行提高，以适应当前的访问模式，避免数据的频繁迁移。

4.3.2 冷热数据迁移实现

上一小节描述了冷热数据的迁移策略，本节将具体介绍策略实现方案。冷热数据迁移模块是独立于冷热数据库的模块，它负责监控热数据的饱和度以将超过阈值的冷数据迁移至冷数据库中进行存储。在其监控的同时，热数据库会保持接受新插入的数据，并为每条数据记录其数据温度和最近访问时间戳。当热库数据量到达设定阈值（高水位）后，本模块会将热数据库中数据以数据温度按从小到大进行排序，并把数据量控制在标准线上（低水位），多出的数据将会被模块迁移至冷数据库中存放。本模块工作流程图如图 4-7 所示：

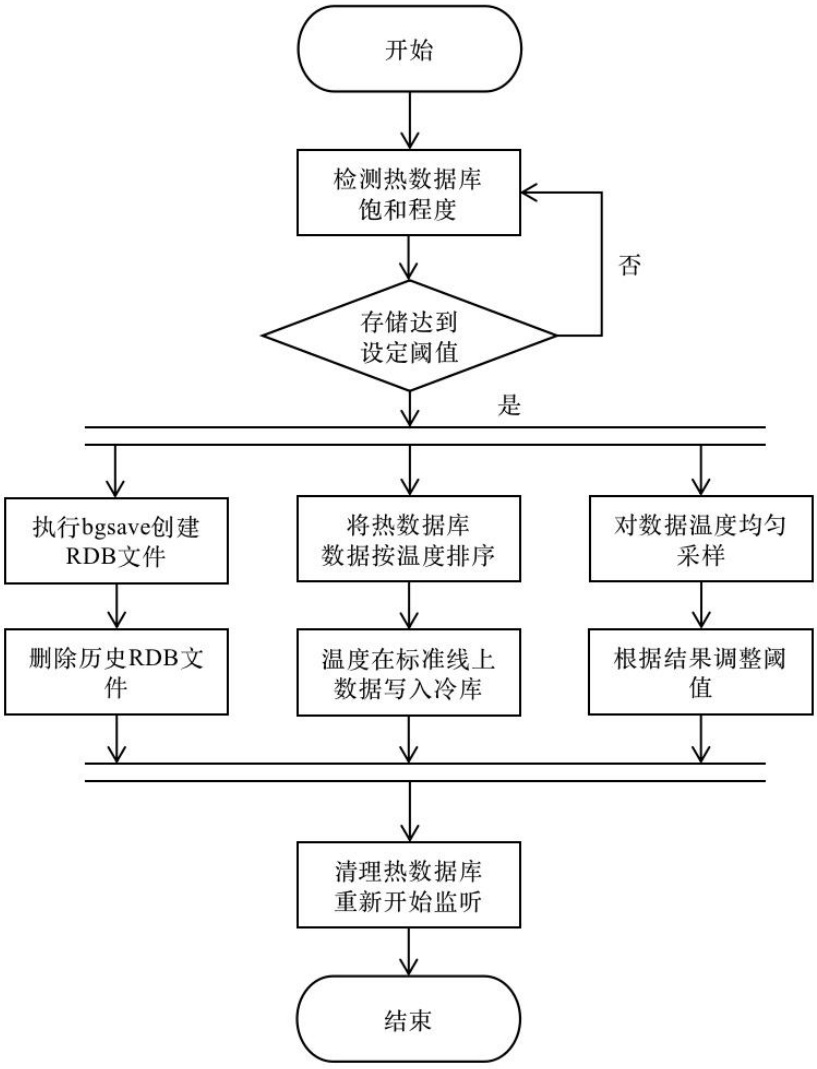

图 4-7 冷热数据迁移流程图

本模块在发生迁移时首先会将数据按温度进行排序，随后的流程主要分三个过程进行，分别是在冷库中写入数据、在热库中删除数据和分析温度分布调整阈值。这三个过程是采用的是多线程的方式实现，互相之间不存在逻辑顺序，以加快整体数据迁移流程。被迁移的数据会调用 HBase API 批量写入冷数据库中，若数据原先在冷库中已经存在，则会以多版本的形式进行保存。热数据库删除数据工作与对数据库建立快照持久化是一个耦合的过程，当热库清理完毕后系统会立即调用 bgsave 命令来创建 RDB 文件，创建完毕后会将历史 RDB 文件丢弃，至此数据的迁移工作已经完成，热数据库也可以开始进行新的数据写入操作。

阈值调整过程在实现时仅依赖于迁移时所计算的数据温度，在数据按温度排序后，此过程会在所有数据中按排序均匀抽取 100 个温度样本，并计算温度样本间的间隔。比如系统原先设定的标准线阈值为 20%，当热数据饱和度达到 80%时触发了数据迁移，则会取排名在 0.8%、1.6%、...、79.2%、80%的数据温度，然后计算采样数据的温度差，由于标准线为 20%，则对比 20%-19.2%与 20.8%-20%的温度差与其他温度差之间的差异。若 20%-19.2%与 20.8%-20%的温差相比在 20%以下的温差明显过大，则将标准线阈值降低至 16%，若过小则将标准线阈值提高至 24%。

完成数据迁移后，存入到冷数据库中的数据将不再拥有温度属性，仅保有数据原有的信息。当冷数据再次被访问的时候，该数据会再次进入热数据库中并重新标定温度，但此时冷数据库中保存的数据将不会被删除，即此份数据会同时保持双副本状态，但冷数据库中存放的是该数据的老版本。在下次数据变冷迁移至冷库时，新数据将覆盖老数据副本，完成数据的更新。数据回温的示意图如 4-8 图所示:

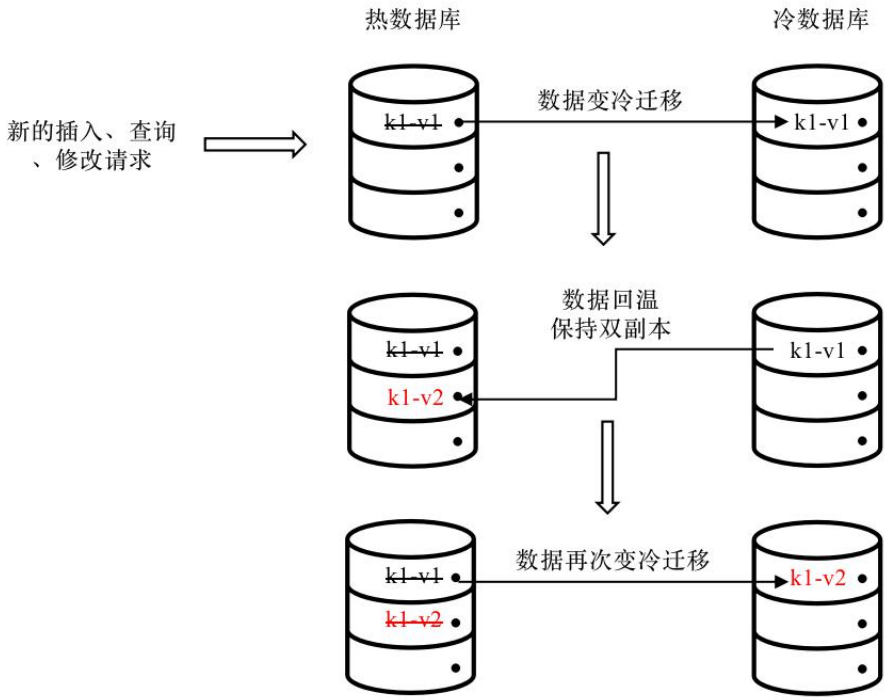

图 4-8 数据回温过程示意图

除了由程序触发的数据迁移外，本系统还实现了手动触发迁移的过程，该过程会将热数据库中的所有数据完全进行迁移，即清空热数据库存储空间。手动迁移是为了应对两种情况而设计，一方面是当数据库或业务终止时，仍会有部分数据驻留在热数据库中，需要将这部分数据同步到冷数据库里，以保证数据的持久化存储；另一方面是当热数据库长期未接受访问时，所存放数据温度均已降到最低，不再具有访问参考价值，此时手动的将这部分数据刷入冷数据库中，能够加快应对新的数据访问模式，避免内存污染。

4.3.3 冷数据压缩存储

HBase 为适应海量数据存储的需求，其提供了对数据的多种压缩支持，如常见压缩算法的 GZIP、LZO、Snappy。不同压缩算法间，在压缩率、编码速度、解码速度等方面均有所差异，而压缩算法的选择取决于所存储数据的应用场景。Google 曾对 HBase 中的几种常用算法进行过测试，测试结果如表 4-1 所示。

表 4-1 HBase 压缩算法测试结果

Algorithm	remaining	Encoding	Decoding
GZIP	13.4%	21MB/s	118MB/s
LZO	20.5%	135MB/s	410MB/s
Zippy/Snappy	22.2%	172MB/s	409MB/s

如表中所示，GZIP 算法拥有最高的压缩率，但是相对的它的编解码速度要远低于另外两种算法，另外两种压缩算法则相对压缩率稍低，却有更高的压缩与解压速度。从上测试结果可以分析得出，当数据被压缩后不常用的情况下，采用 GZIP 的压缩策略，以取得空间方面的最大利用是最为理想的，编码与解码只需保证在需要数据时可以保证数据被完整的恢复出来即可。而当数据仍有很高可能性被再次利用时，则应选择后两种压缩算法，在空间开销与解压时间开销中取得平衡。

对于本文中数据的存储，由于已经在热数据库中进行了冷热数据识别，最终存放在冷数据库中的数据，一定是访问率较低的数据。结合上文提到的数据压缩算法，显然压缩率最高的 GZIP 算法较为适合本文的应用方式。但考虑到数据回温的可能性，当数据刚从热数据库中被迁移至冷数据库中，最有可能因数据迁移策略中阈值的设置问题，存在部分数据很快会被再次访问。因此在冷数据库中，对于刚进入冷库的这部分数据采取 Snappy 压缩策略，能够对数据迁移策略阈值过低所导致的问题进行一定的弥补。冷数据的数据块压缩转移如图 4-9 所示。

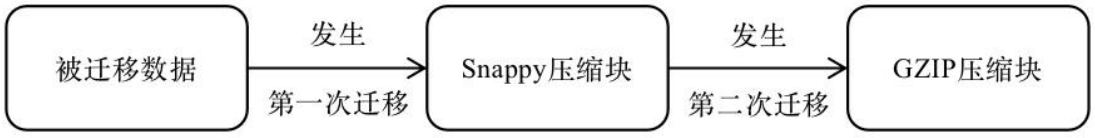

图 4-9 冷数据块压缩策略转移图

如上图所示，当数据从数据迁移模块中被转移到冷数据库时，优先采用 Snappy 的压缩方法，保证数据在回温时不会产生过大的开销。当发生下一次数据迁移时，原先保存的 Snappy 压缩块由于其温度进一步冷却，将转换为 GZIP 压缩块存储。

4.4 本章小结

本章对冷热数据的存储方案进行了整体性的介绍，首先从冷热数据的特点出发，提出了其存储上的需求，再针对具体需求进行了冷热数据库设计。随后通过分析分级数据库的迁移方案，提出本文所采用的冷热数据迁移策略，并详细介绍

了冷热数据迁移模块的设计与实现。

在存储需求方面，冷数据更偏向大量存储，而热数据对读写性能需求有更高要求。本章以此为目标，通过类比工业界划分线上线下数据库的存储结构，设计了本文存储方案所基于的冷热数据库。

在冷热数据迁移策略上，首先分析了常用数据库中的数据持久化方案，以及分级存储中常用的数据迁移方案，分析这些方案的优劣。最后结合上一章所提出的冷热数据判定策略，设计了冷热数据迁移模块的结构流程以及实现方式。

第五章 系统测试与分析

前两章分别介绍冷热数据的判定策略与迁移策略的方案设计与实现。本章将通过实验测试的方式，来检验冷热数据判定策略的准确性以及判定策略的时间空间开销，并与传统缓存替换策略的效果及开销进行对比分析。然后本文将以电子商务基准测试集，来测试真实条件下冷热数据迁移效率以及热数据库的访问命中率。最后将对实验结果进行分析，找出实现中存在的不足以及导致问题的原因。

5.1 实验环境与数据集

本文实验分成两个部分进行，对冷热数据判定策略的实验主要内容是与传统缓存替换策略进行比对实验，该部分实验所基于的环境是在单机环境下在内存的固定大小存储空间里模拟缓存，采用 JAVA 对多种策略进行实现。对冷热数据迁移策略的实验主要内容是对热数据库的命中率以及数据存储空间利用率进行实验测试，该部分实验所基于的环境是单机部署的 Redis 数据库以及集群部署的 HBase 数据库构成，其中集群包含一个主节点与两个从节点。实验具体硬件配置如表 5-1 所示。

表 5-1 实验硬件配置表

硬件设备	配置信息
Redis 主机	操作系统：macOS 11.2.2 处理器：2.9GHz Intel Core i5 内存：8GB LPDDR3
HBase 主节点	操作系统：Ubuntu 16.04 处理器：3.2GHz Intel Core i5 内存：4GB DDR3 磁盘：400GB HDD
HBase 从节点	操作系统：Ubuntu 16.04 处理器：3.2GHz Intel Core i5 内存：4GB DDR3 磁盘：400GB HDD

实验软件环境如表 5-2 所示。

表 5-2 实验软件环境表

软件名称	版本信息
Redis	6.0.8
Hadoop	2.8.5
HBase	2.0.0

对冷热数据判定策略实验所采用的数据是模拟多种数据访问模式的数据集，考虑到数据存在的几种基本访问模式，包括循环访问模式、扫描访问模式、局部访问模式。本文以这几种访问模式模拟缓存中的数据访问，并以此计算缓存命中率以及时间开销。

对冷热数据迁移策略实验是参照 redis-benchmark 程序来编写与测试的，redis-benchmark 模拟了 N 个客户端同时发送 M 个总查询命令到 redis 服务器的过程。本文参照此思路实现了对加入冷热数据迁移策略后 redis 的基准测试，并且采用真实电商场景的工作负载数据集来进行测试。

5.2 实验过程与结果分析

5.2.1 冷热数据判定实验

冷热数据判定策略实验是在程序中模拟缓存访问与替换的过程，为便于测试对比，下文称所提出冷热数据判定策略为 LTU 算法。为对真实缓存进行仿真，本文在实现冷热数据判定策略以及多种缓存替换策略时均对所使用空间大小进行了限定。参考实验设备的真实缓存空间大小如表 5-3 所示。

表 5-3 缓存空间对照表

缓存级别	空间大小
L1 缓存	256 KB
L2 缓存	1.0 MB
L3 缓存	6.0 MB

本文将以 L1 缓存为基础测试大小，优先测试在此大小限制下各策略的缓存命中率。在此基础上，以 256KB 为底数的指数倍数扩大缓存大小限制，测试直到缓存空间足够存储全部测试数据的缓存命中率变化。

在实现冷热数据判定策略时，数据是由三元组构成{data，temperature，timestamp}，其中 timestamp 需要记录系统时钟采用 int 类型存放，而 temperature 采用 float 类型存放，data 在实验中用两个字符来代替。在数据结构实现的过程中，

所采用的结构是 JAVA 的 map 数据结构，以数据的 data 作为 key，而 temperature 和 timestamp 组成的二元组作为 value 保存。数据 temperature 的计算方式参照上文中的公式进行实现，当 map 达到存储空间上限时，则需要根据 temperature 对数据进行排序，并丢弃出 temperature 最低的数据元素。

对比实验的 LFU、LRU 算法同样用 JAVA 参照真实缓存替换实现方案进行实现 ，LRU 是基于双向链表结构实现，LFU 则类似于本文提出的冷热数据判定策略，需要额外空间存储数据的访问命中次数。

对于每种访问模式的测试，本文均选择 5000 个数据量为标准，保证数据量在各算法所限定的存储空间内无法完全存放；数据访问次数选择 1000000 次，以保证每种策略的缓存命中率在重复实验下基本稳定，能够表征算法的准确性。缓存命中率是以未发生数据替换的访问次数与总数据访问次数比值来计算，此命中率亦可用于推算在真实缓存中的命中率以及平均访问时延。

对于循环访问模式，其数据的访问特征是被访问的序列可能按访问顺序再次被访问，例如在循环处理某些文本内容时，部分长内容会被反复访问。该部分数据示例如图 5-1 所示。

```
MacBook-Pro-3:Downloads macbookpro$ cat loop\ sequence.dat
b6 i6 r0 l2 d2 b0 n0 w2 h3 c8 b6 i6 r0 l2 d2 b0 h3 b2 d1 a6 i1 y1 l9 n0 r6 p8 m2
 e5 v3 n0 x0 a5 v5 d0 l9 v4 d8 i8 p2 c3 q4 o0 p7 f1 p0 j2 h0 q2 g7 g6 z0 u4 d5 s
3 z0 d1 r2 r7 g8 j6 t8 l6 p2 d8 f0 l8 z0 l6 g8 a5 x1 m1 y9 i5 x7 f9 d1 i3 f6 m7
g1 p8 f4 k7 e6 k9 f3 g6 w1 p9 p5 o6 n8 m6 m1 n5 f3 r6 n8 g3 m2 o0 s1 u7 o3 h7 xi
```

图 5-1 循环访问模式数据示例图

上图为循环访问测试数据在小数据集上所生成测试数据，如图中表明所示，存在部分序列数据会被循环访问。此模式下的实验结果如图 5-2 所示。

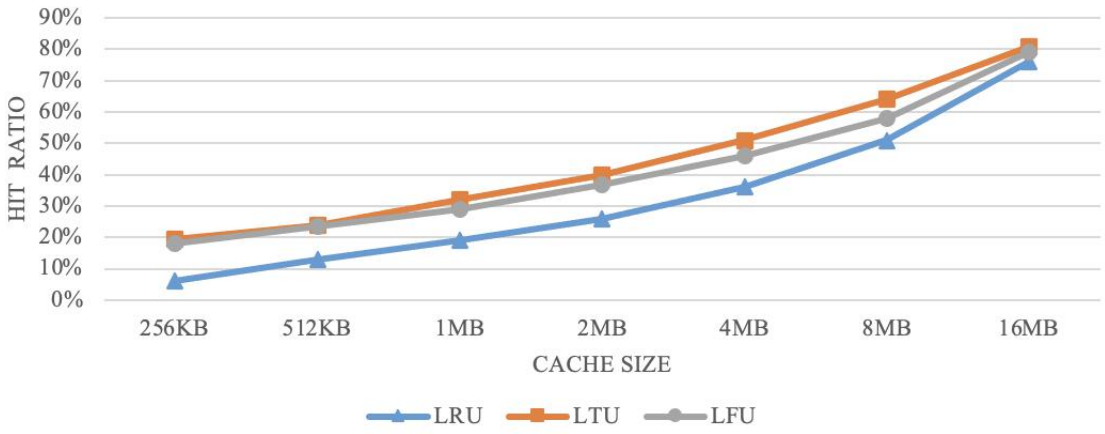

图 5-2 顺序访问模式测试结果图

如上图所示，在此数据访问模式下，LTU 算法与 LFU 算法具有类似的缓存命中率，且 LTU 要略高于 LFU 的结果。而 LRU 的算法效果则相对一般，分析其原因是在此模式下，最近被访问的数据可能具有较低的优先级，而相对历史访问数据更有可能被再次访问到，LRU 每次将最久未访问的数据抛弃恰与本场景相反，因而效果不佳。

对于扫描访问模式，它的数据访问与循环访问模式相反，更类似于对大型数据集的反复扫描，在测试数据中会出现具有回文特征的访问序列。该部分数据示例如图 5-3 所示。

```
MacBook-Pro-3:Downloads macbookpro$ cat repeat\ sequence.dat
n6 g1 t6 w8 u1 d8 e6 e4 a9 g2 o8 g2 a9 e4 e6 d8 u1 w8 t6 g1 n6 o0 k2 u4 w9 j9 l9
 d9 i1 u5 d0 z0 q2 z0 d0 u5 i1 d9 l9 j9 w9 u4 k2 o0 g2 a9 e4 e6 d8 u1 w8 t6 g1 n
6 y1 p1 s7 r3 r9 t4 r9 r3 s7 p1 y1 z0 d0 u5 i1 d9 l9 j9 w9 u4 k2 o0 g2 a9 e4 e6
d8 u1 w8 t6 g1 n6 r8 m5 a4 o8 c2 m1 i6 m1 c2 o8 a4 m5 r8 r9 r3 s7 p1 y1 z0 d0 u5
 i1 d9 l9 j9 w9 u4 k2 o0 g2 a9 e4 e6 d8 u1 w8 t6 g1 n6 v8 e9 f8 p8 t7 p4 r7 p4 t
7 p8 f8 e9 v8 m1 c2 o8 a4 m5 r8 r9 r3 s7 p1 y1 z0 d0 u5 i1 d9 l9 j9 w9 u4 k2 o0
```

图 5-3 扫描访问模式数据示例图

上图所示同样为扫描访问模式在小数据集上生成的测试数据，可以看到其中包括了大量长度不一的回文序列数据。扫描访问模式下的实验结果如图 5-4 所示。

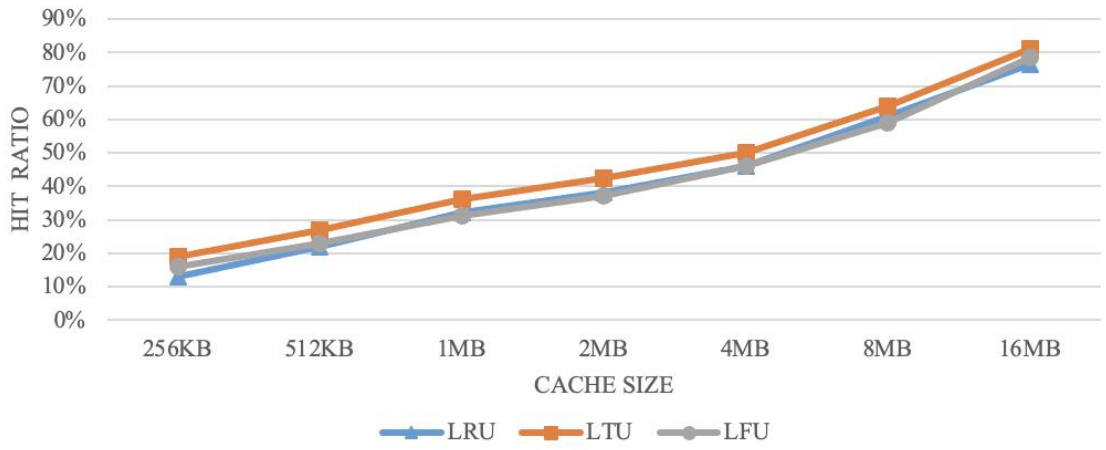

图 5-4 扫描访问模式测试结果图

如图所示在扫描数据访问模式下，纵向比较可以发现 LRU 算法的命中率有明显提升，而 LFU 算法有着与循环访问模式下类似的缓存命中率，本文 LTU 算法的效果也相较于循环访问模式有所提升。分析两种模式下效果差异的原因，推测在本策略中考虑到了数据间的关联性，每个数据的访问都会带来前序数据的升温，这种关联性对于回文形的数据访问具有良好的适用性。

局部访问模式是对真实文件访问模式的一种模拟。考虑到数据局部性原则，在数据访问时往往呈现出与统计学中概率分布类似的访问特征，如类似正态分布的访问局部性。在模拟此类数据访问时，测试数据采用服从幂定律概率分布的生成方式。对于局部访问模式，实验结果如图 5-5 所示。

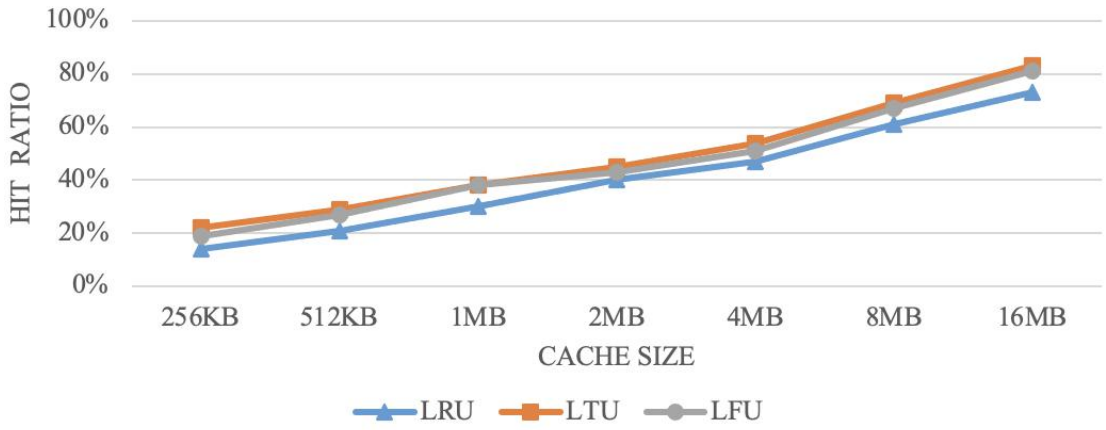

图 5-5 局部访问模式测试结果图

如上图所示，对于局部访问模式的实验结果，三种策略均取得了较前两种模式下更好的效果。综合三种数据访问模式来看，本文所提出的 LTU 算法在命中率上要高于 LFU 与 LRU 两种算法，效果上与 LFU 算法的表现更为接近。冷热数据判定策略针对三种场景都具有良好的适应性，且对于模拟真实文件访问的访问序列具有最优的命中率，说明本策略对数据访问特征具有良好的表征能力。

时间开销方面，本文以缓存命中率最优的局部访问模式下的数据进行策略间的比对测试。测试结果是在不同缓存空间大小下，总数据访问次数与总访问时间的比值。实验结果如图 5-6 所示。

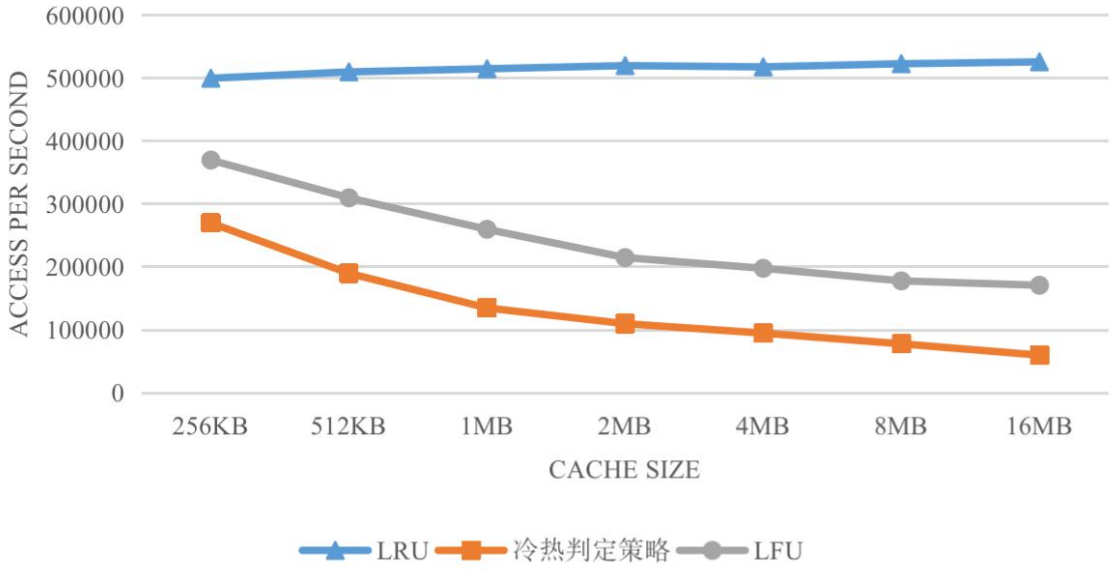

图 5-6 算法性能测试结果图

从上图中可以看出，实现上相对简单的 LRU 算法具有最低的时间开销，其每秒处理数据访问量可达 500000 以上，且随着缓存空间增大，其处理效率仍保持上升趋势。而 LFU 算法与本文提出的冷热判定策略性能要差于 LRU 算法，缓存空间增大也导致了处理效率的下降。从算法实现角度分析，在 LFU 与冷热判定策略的实现过程中，涉及到数据替换的时候都会导致一次对缓存数据的全量排序操作。若遇到数据频繁的换入换出，则会导致反复的排序而带来过多的开销。冷热判定策略相较于 LFU 算法，每当数据被访问时都会涉及到一次指数运算，而 LFU 仅需一次加法运算，且冷热判定策略在进行排序时需对所有数据的温度再次进行运算，因而时间效率上要低于 LFU 算法。

由缓存命中率与算法效率两方面的测试来看，本文提出的冷热判定策略具有良好的缓存命中率，但是在时间开销上有明显缺陷。时间开销上的问题可能由实现方式导致，在真实实现冷热判定策略作为缓存替换策略时，为避免频繁换入换出带来的额外开销，可考虑将策略结合多级缓存空间进行应用，在每级缓存里再采用冷热判定策略计算数据温度并将温度最低的数据放入更高的缓存当中。类似于本文中提出的数据迁移策略，可在从最高级缓存中抛弃数据时，批量化的抛弃一部分数据为缓存留出空间，以将每次排序的结果最大化利用起来。

5.2.2 冷热数据迁移实验

对冷热数据迁移部分的实验，将结合冷热数据判定模块与冷热数据迁移模块共同完成。首先将基于冷热数据判定模块进行基本功能测试，以保证后续测试的

正确性。随后将基于两个模块共同完成redis-benchmark进行性能测试，与原生Redis数据库的性能进行比对。

冷热数据判定模块在本系统中相当于服务端，原先用户需通过Redis客户端来连接Redis服务器，然后在客户端上发起对数据库的相应操作。本文的冷热数据判定模块实现了对Redis服务端的连接，并对系统所需的基本Redis命令进行了相应的实现。由于本文所需数据结构仅涉及到了Redis的哈希结构，因此在命令方面对其进行了简化操作，便于用户输入和使用。冷热判定模块命令与Redis客户端命令对照表如表5-4所示。

表5-4 Redis命令对照表

冷热判定模块命令	Redis客户端命令	功能
ADD KEY VALUE	HMSET key field value [field value]	创建一个键为key的哈希表对象
DEL KEY	DEL key	删除指定key的对象
	HDEL key field1 [field2]	删除一个或多个哈希表字段
GET KEY	HGETALL key	获取在哈希表中指定key的所有字段和值
	HGET key field	获取存储在哈希表中指定字段的值
SET KEY VALUE	HSET key field value	将哈希表key中的字段field的值设为value

由于对用户来说，其插入数据库的数据仍为普通的KV对，而冷热判定模块会对该数据进行拆分和再封装，在原先数据基础上加入timestamp、temperature、prefixkey三个键值对，再存入Redis当中。下面是对数据的增删查改几种基本操作的功能测试，测试结果如图5-7所示。

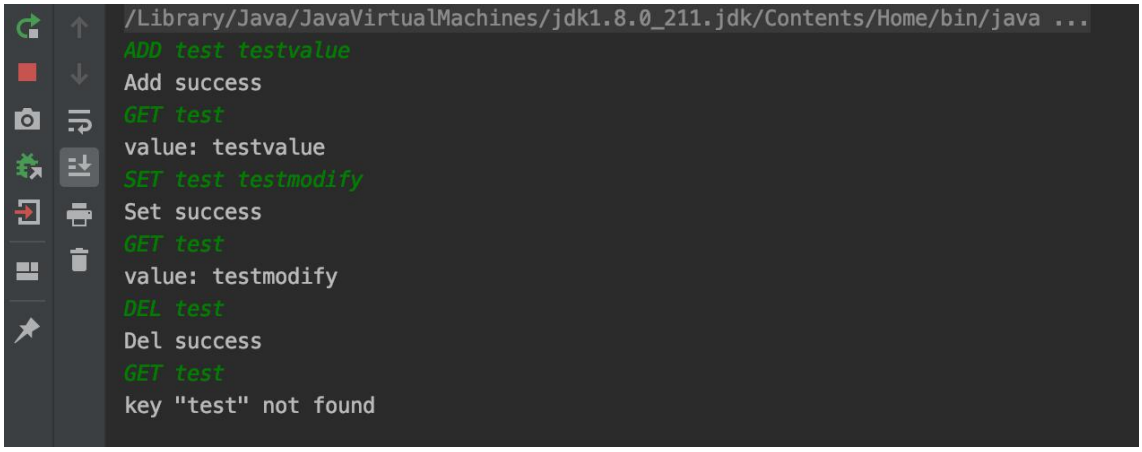

图 5-7 迁移模块基本功能测试结果图

如上图所示，连接冷热数据判定模块后，测试首先插入了键为 test 值为 testvalue 的键值对，模块返回结果：插入成功；然后利用 GET 命令尝试查询键为 test 的数据，得到了返回结果：成功获取到数据 testvalue。下一步测试了 SET 命令，修改了键 test 所对应的值，然后再通过 GET 命令验证修改操作是否成功执行，测试得到返回结果：成功获取到了修改后的值 testmodify。最后测试了 DEL 命令，将键为 test 的数据从数据库中删除，同样用 GET 尝试获取数据，得到了返回结果：键为 test 的数据不存在。上述测试操作证明了 ADD、SET、GET、DEL 四个基本命令的功能正确性，满足了后续测试的基础需求。同时，测试采用 Redis 原生客户端对插入数据进行了查询，结果如图 5-8 所示。

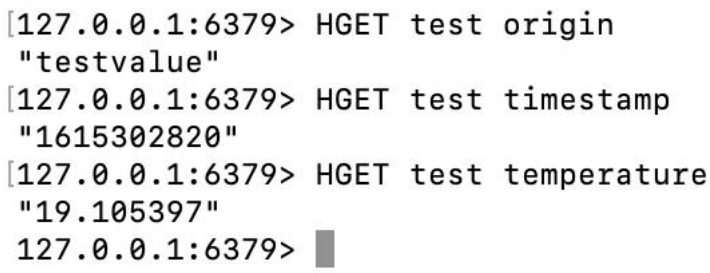

图 5-8 迁移模块基本功能验证测试结果图

上图表明在 Redis 中，刚刚插入的数据可以被成功获取到，并且冷热数据判定模块所需的数据也被成功封装和存储。

在确保基本功能正确后，下面对模块进行性能测试。原生的 redis-benchmark 提供了对 Redis 数据库全方位的基准测试，包括了 SET、GET 以及集合、列表的 PUSH、POP 等多种操作的性能测试。其基础测试结果如图 5-9 所示。

```
        MacBook-Pro-3:~ macbookpro$ redis-benchmark -c 50 -n 10000 -q
PING_INLINE: 52910.05 requests per second
PING_BULK: 57471.27 requests per second
SET: 51282.05 requests per second
GET: 59880.24 requests per second
INCR: 57803.47 requests per second
LPUSH: 51282.05 requests per second
RPUSH: 53191.49 requests per second
LPOP: 56497.18 requests per second
RPOP: 51546.39 requests per second
SADD: 54054.05 requests per second
HSET: 52356.02 requests per second
SPOP: 60240.96 requests per second
ZADD: 50000.00 requests per second
ZPOPMIN: 58479.53 requests per second
LPUSH (needed to benchmark LRANGE): 52356.02 requests per second
LRANGE_100 (first 100 elements): 17857.14 requests per second
LRANGE_300 (first 300 elements): 8123.48 requests per second
LRANGE_500 (first 450 elements): 5694.76 requests per second
LRANGE_600 (first 600 elements): 4282.66 requests per second
MSET (10 keys): 43103.45 requests per second
```

图 5-9 redis-benchmark 原生测试结果图

如上图所示，测试参数指定 50 个并发连接数和 10000 个请求数，即代表有 50 个 Redis 客户端同时发送 10000 个总查询的请求，通过计算每种操作的平均响应时间得到了相应的测试结果。通过查看 redis-benchmark 官方文档，可以得到其测试过程中所使用的方案也十分简单，测试过程都是针对单个键值进行测试的，而面对不同数据访问模式的工作负载模式，无法在该测试中表现出来，且其测试数据量不会导致内存空间不够用的结果，因此该测试指标均为内存命中的环境下的结果，不具真实应用场景下的可比性。

为建立合适的测试方案，本系统基于 redis-benchmark 的框架，简化了其中的测试目标，仅对上文中本系统提供的 GET 和 SET 命令进行测试，且所采用的数据集是天池实验室所提供的“淘宝用户行为数据集”，该数据集记录了 2017 年 11 月末一周的共计上亿条淘宝用户行为数据，其涉及约百万位不同用户的 4 种行为（点击、购买、加购、喜欢）以及四百多万件不同商品。数据集用户行为信息如表 5-5 所示。

表 5-5 淘宝用户行为数据表

列名称	说明
用户 ID	整数类型，序列化后的用户 ID
商品 ID	整数类型，序列化后的商品 ID
商品类目 ID	整数类型，序列化后的商品所属类目 ID
行为类型	字符串，枚举类型，包括('pv', 'buy', 'cart', 'fav')
时间戳	行为发生的时间戳

为将数据用于测试，本测试对数据进行了清洗，首先将 4 种用户行为简化为了对数据的查询和修改操作，将'pv'和'fav'定义为 GET 操作，将'cart'和'buy'定义为 SET 操作。最终数据保留了一条用户行为记录中的商品 ID、行为类型以及时间戳信息，由于商品具体信息在数据中并为提供，因此本测试为商品加入了固定长度的商品信息。测试过程将按照用户行为时间戳对数据进行查询操作，当数据查询失败时则会触发商品信息插入操作，测试过程中冷热数据迁移模块将在后台运行并监控 Redis 数据库状态，并触发执行数据迁移操作。最终测试结果以测试运行总时间与每种类型查询条数比值，估算各类操作性能。

本测试将与原生 Redis 数据库性能采用相同的测试方式进行对比测试，并且对原生 Redis 开启 RDB 机制保证数据的持久化。为对比加入冷热数据迁移前后的性能的差异，测试过程中记录了总记录数 25%、50%、75%和 100%时的系统耗时。测试结果如图 5-10、5-11 所示。

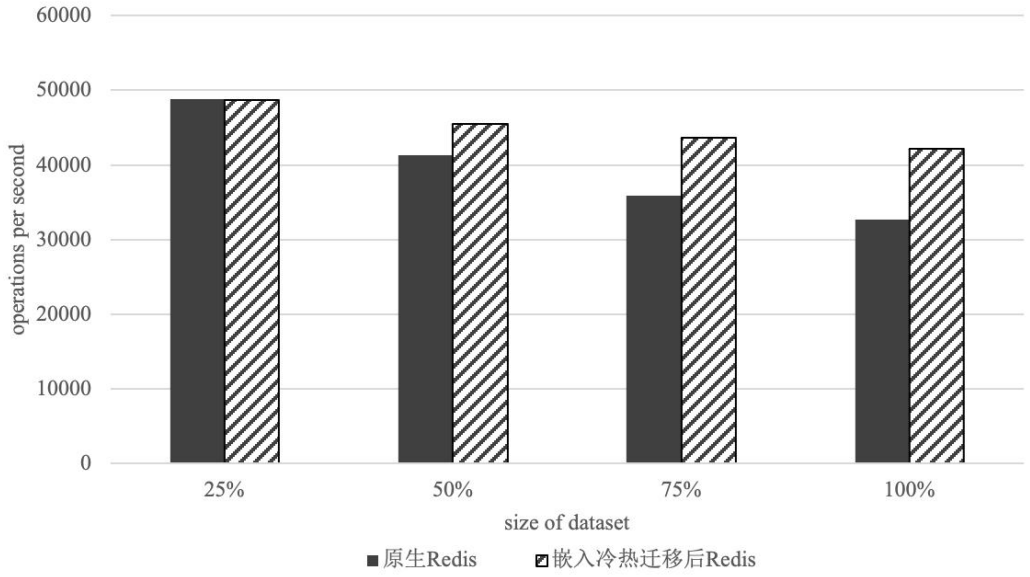

图 5-10 GET 操作性能测试结果图

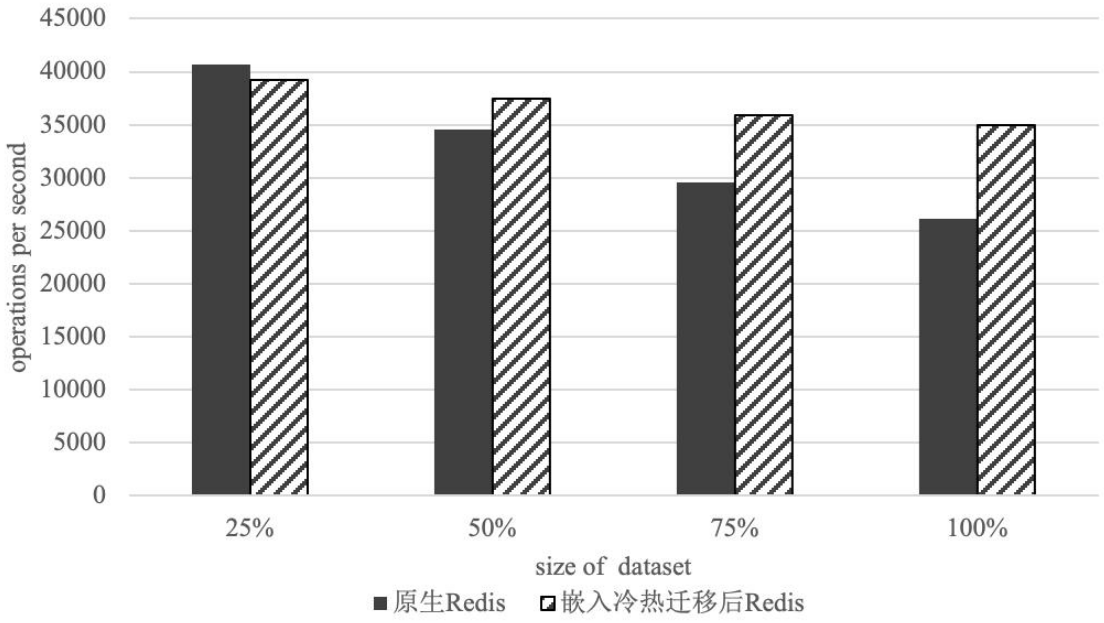

图 5-11 SET 操作性能测试结果图

如上图所示，总体上加入冷热数据迁移策略后的 Redis 要比原生 Redis 具有更优秀的性能。具体来说，在原生和加入冷热数据迁移的 Redis 中，SET 命令的每秒执行操作数要低于 GET 操作；两者的性能在测试进行到数据集 25%数据量时差异并不明显，原生 Redis 性能会稍优于加入后的性能，而当测试达到 50%及以后的进度时，加入冷热数据迁移策略的 Redis 性能开始超过原生 Redis，且差距随进度不断拉大。分析实验结果在 50%进度开始有分化的原因，可能是在进度为 25%时，数据在内存中基本可以存储，无需将数据持久化到磁盘中，而测试进展至 50%甚至之前，数据开始出现部分迁移至磁盘的情况。在原生 Redis 中，针对内存存储不足的情况 Redis 会按 RDB 机制自动执行 save 操作对数据进行持久化，此机制可能导致数据被多次持久化以及内存磁盘间数据冗余，进而导致数据插入操作数量上升，影响其他操作的性能。本文提出的冷热数据迁移策略，按温度对数据选择性的持久化，按数据访问模式对内存数据及时进行迁移，平衡了各种操作间的工作负载，达到降低整体操作时延的效果，本实验结果也表明了本文策略的有效性。

5.3 本章小结

本章基于前两章的描述，对本文所提出的冷热数据判定策略以及冷热数据迁移策略分别进行了实验测试。

在冷热数据判定策略测试方面，本文以限制内存空间的方式模拟缓存环境，并通过模拟多种数据模式来测试基于冷热数据判定策略的替换策略，并与传统缓

存替换策略间进行了缓存命中率与算法性能两个方面的对比。测试结果表明本文所提出的策略在缓存命中率上有一定优势。

在冷热数据迁移策略测试方面，本文首先测试了作为系统服务端的冷热数据判定模块其基本功能，包括数据的增删查改四种基本操作，保证了后续测试的正确性。随后基于 redis-benchmark 基准测试方案，将天池实验室的“淘宝用户行为数据集”进行适应性修改，完成了对加入冷热数据迁移策略后的 Redis 与原生 Redis 间的性能测试比对。结果表明加入本文所提出迁移策略后，Redis 相比原生数据库方案在 GET 与 SET 操作上均有了一定的提升。

第六章 总结与展望

6.1 全文总结

随着数据量的不断提升以及硬件设备的不断发展，数据库在存储数据时应加入更多方面的考虑，以求得更高的存储性能以及查询速率。传统数据库通常都还是基于单一存储介质的存储方案，导致一些高性能设备用于存放大量非常用数据，从而造成其性能上的浪费。为解决存储设备与数据间的匹配问题，分层存储的思想在上世纪就被提出，但在考虑数据特征时仍存在一定的局限性，且无法很好的适应数据访问模式的变化。针对这些问题，本文围绕冷热数据的判定策略与存储策略进行了分析研究，提出一种具有较好识别性和适应性的冷热数据存储机制。本文的主要工作总结如下：

1) 描述了基于不同存储介质的混合存储架构以及其所面临的关键问题，介绍了冷热数据识别策略和冷热数据存储策略相关的国内外发展现状。在此基础上分析发现当前冷热数据存储机制仍面临的数据特征考虑局限、数据访问模式难以识别的问题。

2) 详细阐述了冷热数据存储机制相关的基础理论知识和技术，包括操作系统中常见的缓存替换策略 LRU 与 LFU 及其优化策略，并回顾了分层存储技术的发展历史，包括 HSM 与 ILM 的出现与发展，为后文存储策略设计进行指导。

3) 设计了一种基于数据多维度特征的冷热数据判定策略，以及配合冷热数据判定策略的数据存储策略。基于数据的访问时间、访问频率、数据关联性三个方面的特征，构建了数据温度模型，以对数据热度进行量化。然后提出了基于数据温度的数据迁移策略，对数据访问模式进行识别并自适应性的调整数据迁移策略。

4) 测试了本文提出的冷热数据判定策略与冷热数据存储策略的性能，将冷热数据判定策略与传统缓存替换策略 LFU、LRU 进行了命中率与性能两方面的对比，并完成了嵌入冷热数据迁移策略的 Redis 数据库基准测试。对测试结果进行了分析，提出可能存在的问题以及改进方向。

6.2 课题展望

本文实现的冷热数据判定策略，通过结合数据多维特征，达到了冷热数据识别精确度上的优化，但如实验结果所示在算法性能方面仍存在提升空间，在算法实现上可考虑基于优先队列等对数据排序支持性更好的数据结构来存放数据，同

时将冷数据批量分级化的进行抛弃，以降低频繁替换时排序所带来的性能下降。在存储策略方面，目前已实现对数据访问模式的识别，并会根据情况来调整迁移阈值，但调整的幅度现在相对固定，此部分若考虑加入机器识别的算法，对历史访问模式进行识别，将阈值调整至最合适的位置可对整体冷热数据迁移效果带来进一步的提升。

致 谢

时光飞逝，在电子科技大学历时七年的学习生涯已经接近尾声。回想硕士三年学习的经历，有太多的感慨难以言表，付出了很多的同时也深感收获满满，最大的收获即是一路走来助我成长的人。

首先要衷心感激我的研究生导师侯孟书教授三年以来对我的指导。从研零保研到实验室首次与侯老师交谈，就感受到了老师的学识渊博、平易近人，老师对学术与工作的严谨以及待人的谦和都给我留下了深刻的印象。学术上侯老师带领我入门了分布式存储领域，教导了我要追根溯源的去研究问题，这些思维方式都在科研生活的方方面面影响着我。本论文的撰写过程中，侯老师从选题到设计都给予了耐心的指导，帮助我不断的修改和完善论文。在生活上，侯老师言传身教的告诉我们要成为一个有温度的人，他的生活哲学鼓励我积极面对生活中遇到的各种挫折，侯老师的谆谆教诲我将一生受用。

然后我要感谢实验室的其他老师们，与老师们的学术交流与讨论启发我在科研中不断思考，在参与科研项目中的精益求精教会了我在未来工作中追求卓越的重要性。此外还要感谢实验室的师兄、师姐、师弟、师妹以及同届同学们，在学习生活上互帮互助同甘共苦。与实验室伙伴们共同学习成长的时光，是我在研究生阶段收获最珍贵的礼物。

最后我要感谢我的家人们为我提供了良好的学习环境，让我能够安心完成自己的学业，勇敢的面对生活中遇到的各种问题。感谢我的女友在生活中的陪伴，在我最艰难时对我的不离不弃。我会将这三年里所学的一切牢记于心，继续以积极的态度乐观生活不断前进。

向评阅本文的所有老师致以真挚的感谢与祝福。

参考文献

[1] Sylvain Moreau, Florent Ripaud, Fethi Saidi , et al. Zfs: The last word in file systems[J]. Materials Transactions, 2006, 47(4):1115-1120.

[2] 王峰,王伟,刘洋.一种基于固态盘和硬盘的混合存储架构[J].河南师范大学学报(自然科学版),2013,41(04):153-157.

[3] 吕帅,刘光明,徐凯,等.海量信息分级存储数据迁移策略研究[J].计算机工程与科学,2009,31(S1):163-167.

[4] 江菲,汤小春,张晓,等.基于价值评估的数据迁移策略研究[J].电子设计工程,2011,19(07):11-13.

[5] 施光源,王恩东,张宇.基于块级的分级存储数据特征模型及其应用研究[J].计算机研究与发展,2013,50(S1):322-331.

[6] 黄冬梅,杜艳玲,贺琪.混合云存储中海洋大数据迁移算法的研究[J].计算机研究与发展,2014,51(01):199-205.

[7] 郭刚,于炯,鲁亮,等.内存云分级存储架构下的数据迁移模型[J].计算机应用,2015,35(12):3392-3397.

[8] 张雷,李琳,陈鸿龙,Daniel Bovensiepen.一种面向工业边缘计算应用的缓存替换算法[J/OL].计算机研究与发展:1-11[2021-03-10].http://kns.cnki.net/kcms/detail/11.1777.TP.20210225.1045.002.html.

[9] 王海艳,伏彩航.基于 HBase 数据分类的压缩策略选择方法[J].通信学报,2016,37(04):12-22.

[10] 冯超政,蒋溢,何军,等.基于冷热数据的 MongoDB 自动分片机制[J].计算机工程,2017,43(03):7-10+17.

[11] Liu Yi,Ge Xiongzi,Huang Xiaoxia, et al. MOLAR: A Cost-Efficient, High-Performance SSD-Based Hybrid Storage Cache[J]. The Computer Journal,2015,58(9).

[12] Zhao D, Qiao K, Raicu I. Towards cost-effective and high-performance caching middleware for distributed systems[J]. International Journal of Big Data Intelligence, 2016, 3(2): 92-110.

[13] Luo D, Wan J, Zhu Y, et al. Design and implementation of a hybrid shingled write disk system[J]. IEEE Transactions on Parallel and Distributed Systems, 2015, 27(4): 1017-1029.

[14] Baek S H, Park K W. A fully persistent and consistent read/write cache using flash-based general SSDs for desktop workloads[J]. Information Systems, 2016, 58: 24-42.

[15] Wu X, Reddy A L N. Managing storage space in a flash and disk hybrid storage system[C]//2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems. IEEE, 2009: 1-4.

[16] Chen F, Koufaty D A, Zhang X. Hystor: Making the best use of solid state drives in high performance storage systems[C]//Proceedings of the international conference on Supercomputing. 2011: 22-32.

[17] Bai S, Yin J, Tan G, et al. FDTL: a unified flash memory and hard disk translation layer[J]. IEEE Transactions on Consumer Electronics, 2011, 57(4): 1719-1727.

[18] SHINICHI HAYASHI,NORIHISA KOMODA. Evaluation of Volume Tiering Method and SSD Cache Method in Tiered Storage System[J]. Electronics and Communications in Japan,2015,98(7).

[19] Huang C C, Nagarajan V. ATCache: Reducing DRAM cache latency via a small SRAM tag cache[C]//Proceedings of the 23rd international conference on Parallel architectures and compilation. 2014: 51-60.

[20] Dan A, Towsley D. An approximate analysis of the LRU and FIFO buffer replacement schemes[C]//Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems. 1990: 143-152.

[21] O'neil E J, O'neil P E, Weikum G. The LRU-K page replacement algorithm for database disk buffering[J]. Acm Sigmod Record, 1993, 22(2): 297-306.

[22] Johnson T, Shasha D. 2Q: a low overhead high performance bu er management replacement algorithm[C]//Proceedings of the 20th International Conference on Very Large Data Bases. 1994: 439-450.

[23] Chou H T, DeWitt D J. An evaluation of buffer management strategies for relational database systems[J]. Algorithmica, 1986, 1(1-4): 311-336.

[24] Jiang S, Zhang X. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance[J]. ACM SIGMETRICS Performance Evaluation Review, 2002, 30(1): 31-42.

[25] Guo F, Solihin Y. An analytical model for cache replacement policy performance[C]//Proceedings of the joint international conference on Measurement and modeling of computer systems. 2006: 228-239.

[26] Zhou S. An efficient simulation algorithm for cache of random replacement policy[C]//IFIP International Conference on Network and Parallel Computing. Springer, Berlin, Heidelberg, 2010: 144-154.

[27] Karedla R, Love J S, Wherry B G. Caching strategies to improve disk system performance[J]. Computer, 1994, 27(3): 38-46.

[28] Ghasemzadeh H, Mazrouee S, Kakoee M R. Modified pseudo LRU replacement algorithm[C]//13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'06). IEEE, 2006: 6 pp.-376.

[29] Robinson J T, Devarakonda M V. Data cache management using frequency-based replacement[C]//Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems. 1990: 134-142.

[30] Arlitt M, Friedrich R, Jin T. Performance evaluation of web proxy cache replacement policies[C]//International Conference on Modelling Techniques and Tools for Computer Performance Evaluation. Springer, Berlin, Heidelberg, 1998: 193-206.

[31] Jin S, Bestavros A. Popularity-aware greedy dual-size web proxy caching algorithms[C]//Proceedings 20th IEEE International Conference on Distributed Computing Systems. IEEE, 2000: 254-261.

[32] Cormode G, Muthukrishnan S. An improved data stream summary: the count-min sketch and its applications[J]. Journal of Algorithms, 2005, 55(1): 58-75.

[33] Charikar M, Chen K, Farach-Colton M. Finding frequent items in data streams[C]//International Colloquium on Automata, Languages, and Programming. Springer, Berlin, Heidelberg, 2002: 693-703.

[34] Goyal A, Daumé III H, Cormode G. Sketch algorithms for estimating point queries in nlp[C]//Proceedings of the 2012 joint conference on empirical methods in natural language processing and computational natural language learning. 2012: 1093-1103.

[35] Sienknecht T F, Friedrich R J, Martinka J J, et al. The implications of distributed data in a commercial environment on the design of hierarchical storage management[J]. Performance Evaluation, 1994, 20(1-3): 3-25.

[36] Reiner D, Press G, Lenaghan M, et al. Information lifecycle management: the EMC perspective[C]//Proceedings. 20th International Conference on Data Engineering. IEEE, 2004: 804-807.

[37] Chen Y. Information valuation for information lifecycle management[C]//Second International Conference on Autonomic Computing (ICAC'05). IEEE, 2005: 135-146.

[38] Oe K, Iwata S, Honda T, et al. On-The-Fly Automated Storage Tiering (OTF-AST)[C]//Proc. of The Third Asian Conference on Information Systems – Special Session on Information Storage (ACIS-IS 2014), Nha Trang, Viet Nam. 2014: 537-544.

[39] Cheng Y, Iqbal M S, Gupta A, et al. Cast: Tiering storage for data analytics in the cloud[C]//Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing. 2015: 45-56.

[40] O’ Sullivan C T. Newton’ s law of cooling—A critical assessment[J]. American Journal of Physics, 1990, 58(10): 956-960.

攻读硕士学位期间取得的成果