分布式存储系统热点问题综述

高丹

摘 要 现代全球范围的互联网服务(如搜索、社交网络和电子商务)由跨越数百至数千台服务器的大规模分布式存储系统提供支持。分布式存储系统的任务是在巨大且不可预测的负载下提供快速、可预测的性能。像脸书的 memcached 部署这样的系统，存储数万亿个对象，并在每次用户交互中被访问数千次。为了实现一定的规模，这些系统分布在许多节点上；为了实现性能的可预测性，它们主要或完全将数据存储在内存中。这些系统面临的一个关键挑战是在热点问题严重即工作负载高度倾斜的情况下保持负载均衡，也就是解决热点问题。在如今的大数据时代，数据流行度通常是倾斜的，遵循幂律分布，也就是说热点问题普遍存在。热点问题通常会导致较大的服务延迟，甚至可能会使整个系统崩溃或者阻塞，这对用户来说是不能容忍的，因此在热点存在的情况下保持分布式存储系统的性能和可靠性十分重要。本文分析了热点问题产生的原因和解决热点问题的意义，并对解决热点问题的方法进行了归纳，从一致性哈希、数据迁移、数据副本、前端数据缓存、单节点热点问题这几个方面进行了介绍，主要针对每种方法介绍了几种经典技术并分析了该方法的优劣，最终对解决热点问题的技术进行了总结和展望。

关键词 热点；分布式存储系统；数据迁移；数据副本；前端缓存

Overview of Hot Issues in Distributed Storage System

Dan Gao

Abstract Modern global Internet services (such as search, social networking, and e-commerce) are supported by large-scale distributed storage systems spanning hundreds to thousands of servers. The task of distributed storage systems is to provide fast and predictable performance under huge and unpredictable loads. Systems like Facebook's memcached deployment store trillions of objects and are accessed thousands of times during each user interaction. In order to achieve scale, these systems are distributed on many nodes; in order to achieve predictability of performance, they mainly or completely store data in memory. A key challenge faced by these systems is to maintain load balance when the hotspot problem is serious, that is, when the workload is highly sloping, that is, to solve the hotspot problem. In today's big data era, data popularity is usually skewed and follows a power-law distribution, which means that hot issues are widespread. Hotspot issues usually cause large service delays, and may even crash or block the entire system. This is intolerable to users. Therefore, it is very important to maintain the performance and reliability of the distributed storage system in the presence of hotspots. This article analyzes the causes of hot issues and the significance of solving hot issues, and summarizes the methods to solve hot issues, from consistent hashing, data migration, data copy, front-end data caching, and single-node hot issues and mainly introduced several classic techniques for each method then analyzed the pros and cons of the method, and finally, summarized and prospected the techniques for solving hot issues.

Key words Hotspot; Distributed storage system; Data migration; Data replica; Front-end cache

1 引言

随着互联网的不断发展，全球网络化的不断普及，海量信息的不断聚合，数据量保持指数级增长。面对当前的海量数据，传统的单机文件存储系统并不能提供其所需要的存储能力和高性能读、写需求，同时，传统文件系统的扩展性不佳也使得当数据业务需求发生变化时服务能力不足。过去几十年里，关系型数据库系统因为其高可靠性和稳定性，在数据存储领域得到广泛使用。然而，随着大数据时代到来，数据的巨大规模以及要求快速响应的特性，传统的关系型数据库在处理方面显得力不从心。因为利用数据库中 sql 这样的查询语句，既要从磁盘读取数据，还要消耗时间解析 sql 语句，这样的代价十分大，所以分布式存储系统得到了广泛的使用。

如今基于云的在线服务，如搜索、电子商务和社交网络等应用的很大一部分对延迟十分敏感。然而，设计一个数据中心规模的系统来保证大多数用户请求的低延迟是众所周知的挑战。这种对延迟要求很高的系统通常由分布式存储系统提供支持，这种存储跨越数百台服务器，并通常使用一致的哈希算法提供快速定位和检索数据的方法。这种分布式存储系统通常使用横向扩展架构，通过这种架构，数据跨服务器池进行分区，每个服务器在内存中保存数据的一个区块，并负责针对该区块提供查询。在当今横向扩展环境中分布式存储系统的灵活性和可扩展性使其成为低延迟数据服务应用的最先进方法。然而，这种方法的一个主要痛点是，当数据量十分大或者用户请求十分庞大时，由于数据的流行度分布不均，通常会出现热点问题。

在业务层面，热点问题很好理解，最典型的就是双十一零点秒杀，此时用户访问请求十分庞大，且某些活动力度大的商品访问量可能十分巨大，这个时候这些商品的数据流行度就会增大，也就是成为了热点。又比如社交网络，在社交网络中，与普通用户相比，极少数用户非常受欢迎，这两个不同的用户类别(热门和其他)导致了热度分布的高峰和长尾。在很多应用场景中都会出现类似的情况，数据对象通常具有严重倾斜的流行度，这意味着少量的热文件占数据访问的很大一部分，因此，包含热点数据的缓存服务器会变成热点，再加上网络负载不均衡，使得热点问题更加严重。据报道，在脸书集群中，负载最重的链路在 50%以上的时间内利用率比平均水平高 4.5 倍以上。常规观察到的热点以及网络负载不平衡会导致输入/输出性能显著下降，甚至会消除内存解决方案的性能优势。

分布式存储系统数据的流行度通常是倾斜的，遵循幂律分布。Chen 等人[1]分析了多种业务的数据访问分布，如图 1 所示，在日常情况下，针对 1%的数据，只有 50%的用户请求。但是在极端情况下，针对 1%的数据，有 90%的用户请求，即大量的数据访问只集中在少部分的热点数据中，若用离散幂率分布（Zipfian）刻画，其θ参数约为 1.22。这表明热点问题在互联网时代变得空前严重。

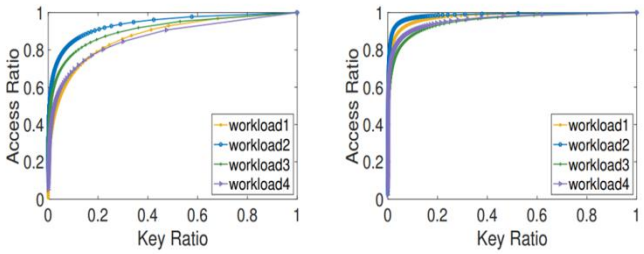

图 1 数据访问分布

这种现象背后有几个原因。首先，在线应用的活跃用户数量持续增长，实时事件(例如，在线促销、突发新闻)能够在短时间内吸引数十亿人访问几个数据，因此快速访问这些热点至关重要。以苹果新手机发售举例，手机的库存等信息只存在分布式存储系统的一个节点中，当新手机发售后，大量的果粉疯狂进行抢购下单，业务的访问量基本都聚集在这一个节点上，节点可能无法承载大量的热点访问，进而引发系统崩溃，严重影响用户体验。据报道，每 0.1 秒的加载延迟将让亚马逊损失了 1%的销售额，谷歌搜索结果每增加 0.5 秒的加载延迟，就会导致流量下降 20%。其次，由于此类应用程序下的基础架构越来越复杂，管道中某处的小错误(例如，由于软件错误或配置错误)有时会出人意料地导致对项目的重复访问(例如，无休止地读取和返回错误消息)。这种不可预测的热点可能使整个系统崩溃或阻塞。因此，如何解决热点问题并在热点存在的情况下保持分布式存储系统的性能和可靠性是非常重要的。

2 原理和优势

许多解决方案可以解决集群范围内的热点问题，例如一致性哈希、数据迁移、数据副本和前端数据缓存。此外，单节点热点问题也有一定的解决方案。例如，计算机体系结构利用分层存储布局(例如，磁盘、随机存取存储器、中央处理器高速缓存)在低延迟存储介质中缓存频繁访问的数据块。许多存储系统，例如 LevelDB 和 RocksDB，使用内存中的 KVSes 来管理热点数据。

一致性哈希就是通过一致性哈希算法将热点数据分片，使用一致性哈希算法分散到多个节点上，缓解系统的热点问题，然而一致性哈希算法无法应对热点动态变化的情况。

数据迁移就是通过存储系统中热点数据的物理迁移来解决热点问题。在这种方法中，首先要基于一定的统计数据确定热点数据的位置，也就是热点感知，而且这种热点感知机制一定要具有一定的稳定性，能够适应热点数据的快速变化。确定热点所在存储系统所在位置后，寻找系统中其他合适位置，将引起问题的全部或部分热点数据迁移到其他位置，分流访问密度。

数据副本就是将热点数据复制到多个节点上，分摊热点访问。这种方式的挑战在于如何感知热点以及保持数据副本的一致性，另外如果只是单纯复制数据那么无法应对写密集型工作负载，需要额外的机制解决这个问题。数据迁移和数据副本两种机制的开销相对来说都比较大，比如数据迁移、复制、查询路由、数据一致性等开销。

前端数据缓存通过设置客户端缓存存储热点数据，这种方法代价相对来说较小，极大地减轻了后端服务器的负担。如果用户请求中涉及到很多数据更新的请求，对客户端层面的缓存的数据需要维护，这个过程实现会变得很复杂。

分层存储采用分层机制，将热点数据缓存在低延迟存储介质中，比如 DRAM 和 NVM 混合存储系统[2]，通常将热点数据存放在 DRAM 中。

另外还有许多其他解决方法，比如设计基于热点感知的数据结构[1]、基于数据热度预测模型[3]通过负载聚类指导请求分配到低负载节点等方法。

3 研究进展

3.1 集群范围内热点问题

3.1.1 一致性哈希

一致性哈希算法解决热点问题就是使用一致性哈希算法将数据分片，分布到多个节点上，分摊热点访问。

一致性哈希算法在 1997 年由麻省理工学院的 Karger 等人[4]在解决分布式 Cache 中提出的，设计目标是为了解决热点(Hot spot)问题，初衷和 CARP 十分类似。一致性哈希修正了 CARP 使用的简单哈希算法带来的问题，使得 DHT 可以在 P2P 环境中真正得到应用。

算法具体流程如下：

（1）求出服务器节点的哈希值， 将其配置到 $0\sim2^{32}$ 的圆上；

（2）用同样的方法求出存储数据的键的哈希值并映射到圆上；

（3）从数据映射到的位置开始顺时针查找，将数据保存到找到的第一台服务器上；

（4）如果超过 2^{32} 仍然找不到服务器，就保存到第一台服务器上。

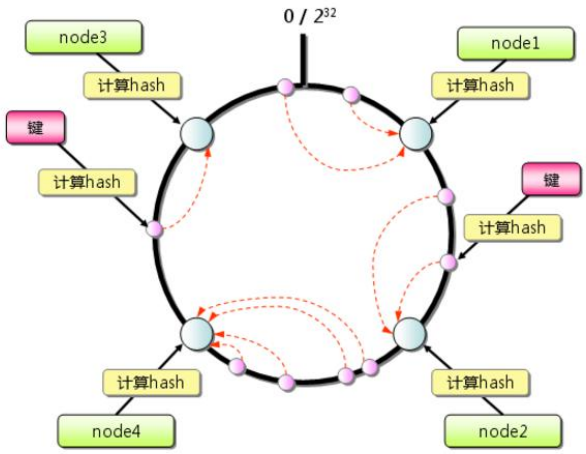

图 2 一致性哈希算法

当有节点出现故障离线时，按照算法的映射方法，受影响的仅仅为环上故障节点开始逆时针方向至下一个节点之间区间的数据对象，而这些对象本身就是映射到故障节点之上的。当有节点增加时，比如，在节点 A 和 B 之间重新添加一个节点 H，受影响的也仅仅是节点 H 逆时针遍历直到 B 之间的数据对象，将这些重新映射到 H 上即可，因此，当有节点出现变动时，不会使得整个存储空间上的数据都进行重新映射，解决了简单哈希算法增删节点，重新映射所有数据带来的效率低下的问题。

后来的研究者在此基础上，提出了虚拟节点的概念来提高分布式存储的负载均衡效[5]，从而解决

分布式存储系统的异构问题，即根据物理服务器性能的不同将其划分为不同个数的虚拟节点，而数据存储位置通过散列计算映射到不同的虚节点。

一致性哈希算法及其改进算法把数据的读写请求均匀分散式的分配，在一定程度上考虑了请求分配的均衡性和异构节点的问题，但与传统的静态负载均衡算法类似，其请求的分配未充分考虑节点负载的动态变化。同时数据定位算法不变，只是多了一步虚拟节点到实际节点的映射，例如定位到“Node A#1”、“Node A#2”、“Node A#3”三个虚拟节点的数据均定位到 Node A 上。这样就解决了服务节点少时数据倾斜的问题。在实际应用中，通常将虚拟节点数设置为 32 甚至更大，因此即使很少的服务节点也能做到相对均匀的数据分布。

3.1.2 数据迁移

数据迁移即通过在存储系统中数据的物理迁移来解决热点问题，将热点数据从一个节点迁移到其他负载较低的节点，缓解该节点的热点问题。

Rebecca Taft 等人[6]在设计的分布式在线事务处理数据库管理系统 E-Store 中对热点问题进行了处理。他们引入了一种细粒度即元组级分区。在一起的工作中只能迁移大数据块，如果大量请求只访问几个热点数据，那么当热点数据被散列到相同的块或被分配到相同的范围，这一种分块策略就会失效。因此他们引入了一种结合细粒度热快和粗粒度冷块的两层方法，通过提取块的热元组来动态改变块，热元组被认为是独立的块。数据迁移具体流程如图 3 所示。

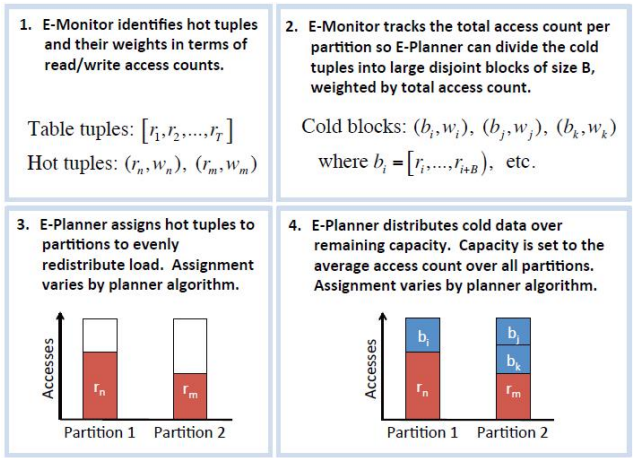

图 3 E-Store 数据迁移流程

在数据迁移时主要有两个控件，一个是 E-Monitor 监视器，一个是 E-Planner 执行器。监视器监视数据块和元组的状态，识别热元组并根据元组的读写次数来计算权重；跟踪每个分区的访问总次数，将冷元组划分为大小为 B 的大型块，按总访问次数计算权重。执行器在整个集群中一次性将热数据分配给分区，以均匀负载，然后将冷数据块放入每个节点的剩余容量中。

Cheng 等人[7]在设计内存对象缓存时实现了一个自适应负载平衡器以经济高效地缓解热点问题。考虑到不同技术的特点，该负载平衡器分为三个阶段，第一个阶段利用数据复制技术，第二阶段实现服务器-本地缓存的数据迁移，第三阶段实现缓存协调迁移。阶段状态转换图如图 4 所示。

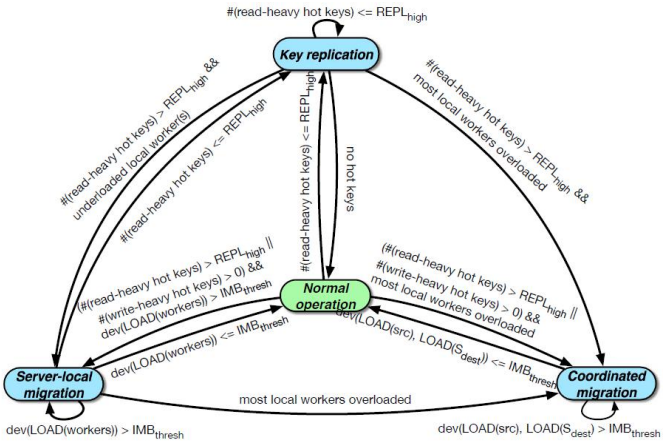

图 4 负载均衡器阶段状态转换图

第一阶段的数据复制提供了一种细粒度的机制减轻由少数热点数据引起的负载不平衡，不需要昂贵的跨服务器数据移动。但是这种方法不能缓解写密集型工作负载中基于写热点的热点问题。因此，如果复制的热点数据数量超过 REPLhigh 或者观察到系统中出现由于写密集型工作负载导致的热点问题，此时多阶段负载平衡器就会触发其他阶段。

第二阶段的服务器-本地缓存数据迁移缓解由跨越一个或多个缓存的大量热点数据组成的热点问题，此时需要在不同的 worker 之间重新分配缓存。在这一阶段，当本地阶段之间存在高负载不平衡，即存在空虚或者负载较轻的本地 worker，可以通过交换或者迁移过载 worker 的缓存来容纳更多的负载。这一阶段采用征数线性规划算法技术最优迁移方案，保证将过载 worker 的负载降低到可以接受的范围同时不会使其他 worker 过载。

当缓存服务器过载或者缺少可以处理缓存迁移的本地节点是说明阶段二无法找到一个可用的热点问题解决方案，这个时候第三阶段就被触发。第三阶段将缓存从一个缓存服务器迁移到集群中

其他低负载的服务器中。过载的 worker 通知集中式协调器触发缓存服务器之间的负载平衡。协调器定期从所有集群 worker 获取统计信息，包括关于请求到达率(负载)、存储的数据量(内存消耗)和读/写比率的缓存级信息。阶段三中的算法利用这些统计信息来选择负载最低的目标服务器，以将缓存从源过载 worker 重新分配给目标的 worker。

3.1.3 数据副本

数据副本解决热点问题是通过有选择性地将热点数据复制到多个节点上，分摊热点访问。然而，现有的数据副本方法面临着两个挑战，首先是客户端必须能够识别复制地对象及其位置，这可以通过使用集中式目录或者将目录复制到客户端来完成。第二个挑战是数据副本的一致性，很多系统只复制只读对象或者要求用户管理复制的不一致性。

针对以上挑战，Lin 等人[8]于 2020 年在存储系统 Pegasus 中使用一致性目录利用数据副本来解决热点问题。Pegasus 是一种机架级存储系统设计，由多个存储服务器通过单个机架顶部交换机（TopR）连接而成。在交换机 TopR 中实现网络一致性目录，跟踪被复制的数据及其位置，利用交换机的请求流量中心视图，以负载感知的方式将请求转发到副本。与以前的方法不同，Pegasus 的一致性目录还允许它在每次写操作时动态地重新平衡副本集，从而加速读和写密集型工作负载，同时保持很强的一致性。

在 TopR 中实现一致性目录，需要在交换机数据平面中实现一致性目录的所有数据结构和功能逻辑，将赋值的数据及其副本集存储在交换机的内存中，基于自定义的数据报报头进行匹配和转发，并为一致性协议应用目录更新规则。由于交换机的资源有限因此交换机只能支持有限个复制的数据，根据研究，只需要保证 O(nlogn)个最热的数据被复制就可以实现负载均衡。此外一致性目录只存储小的元数据。

在设计了一致性目录之后需要一致性协议来支撑。他们设计了一个基于版本的非阻塞一致性协议。交换机为每个写请求分配一个单调递增的版本号，并将其插入数据报报头。服务器将版本号存储在每个数据旁边，并在读写回复中附上版本号。交换机还在一致性目录中存储每个复制数据的完整版本号。当接收到读或写回复时，交换机将恢复中的版本与目录中的完整版本进行比较，并将副本集重置为仅包含源服务器。随后，后续的读取请求被转发到有新值的服务器，当多个服务器具有数据的最新值时，回复中的版本号等于最终版本。在这种情况下，将源服务器添加到副本集，以便后续的读取请求可以分布在最新的副本中。

负载感知调度用于交换机转发请求。文中实现了一个加权循环策略：存储服务器定期将其系统负载传输到控制器，控制器根据负载信息为每台服务器分配权重并将其存储在交换机上。交换机将请求转发给副本集中与其权重成比例的服务器。这中机制也可以用于写请求，交换机可以在每次写入时为数据选择新的复制副本集，它可以将写请求转发到一个或多个服务器，一致性目录确保数据的一致性。由于交换机可以频繁移动数据，因此它能对读和写请求都使用负载感知调度，这是 Pegasus 提高读写密集型工作负载性能的关键。

3.1.4 前端数据缓存

前端数据缓存就是通过预先标记热点，设置客户端层面的缓存，减小键值存储系统的负载压力。相对于数据迁移来说，设置前端缓存的成本更低。数据迁移需要将数据从一个后端节点移动到另一个后端节点，这在网络带宽、输入/输出、一致性及索引更新等方面代价都很高，而缓存维护成本更低，可以很容易地存储查询或请求地结果，为将来地请求提供服务，而无需重新计算或检索。

Fan 等人[9]在 2011 年通过研究表明，一个小型、快速的前端缓存可以直接为热点数据提供服务，而无需查询后端节点，从而提供有效的动态负载平衡，使后端的负载更加均匀。他们通过使用使用 FAWN-KV 键值存储系统，85 个低功耗后端节点的测试平台集群和一个高性能的具有几兆高速缓存的先进服务器作为前端节点，通过改变前端节点和后端节点的规格以及工作负载来观察系统的性能，最终得出缓存只需要存储 O(nlogn)个最热的数据，就可以保证良好的负载平衡，其中 n 是后端节点的总数(与数据的数量无关)。

然而，使用前端缓存进行负载平衡时，传统的缓存体系结构(如旁路式 Memcached 和路径贯穿式小型缓存)存在一个主要缺点，在这些体系结构中，客户端必须首先将所有读取请求发送到缓存。当命中率较低时，这种方法会带来很高的开销，比如当缓存较小且主要用于负载平衡时，情况就会这样。一些旁路式系统让客户端负责处理高速缓存未命中的情况[，这进一步增加了系统开销和尾部延迟。另一种路径贯穿式缓存将缓存放在前端的负载平

衡器（Load Balancer）这种设计容易受到前端崩溃的影响。

基于以上思想，Li 等人[10]在 2016 年提出了一种新的集群级键值存储体系结构，可以在多种多样且快速变化的工作负载下保持高效率。如图 5c 所示，SwitchKV 使用混合的服务器类，其中专门配置高性能节点充当快速、小型缓存，资源受限的后端节点对数据进行哈希分区之后将数据缓存到这些高性能节点中。

SwitchKV 设计的核心是一种高效的基于内容的路由机制，该机制使用软件定义网络(SDN)技术以最小的开销为请求提供服务。客户端将数据编码到数据报的报头中，并将请求发送到 OpenFlow 交换机。这些交换机维护所有缓存项目的转发规则，包括为每个缓存的数据创建精确匹配规则，为每个后端创建通配符规则，根据嵌入的数据将请求以线路速率直接路由到缓存或后端节点。

SwitchKV 通过将高速缓存移出数据路径，并利用优化后的交换机硬件来匹配并以线路速率将数据转发到正确的节点，从而实现高性能低延迟。所有响应都在一次往返中返回，对于不在缓存中的数据的大量查询没有开销。SwitchKV 可以通过添加更多的缓存节点和交换机来横向扩展，并且对缓存崩溃具有一定的适应力。

评估和分析表明，与传统系统相比，SwitchKV 能够更有效地处理现代云应用的流量特性。

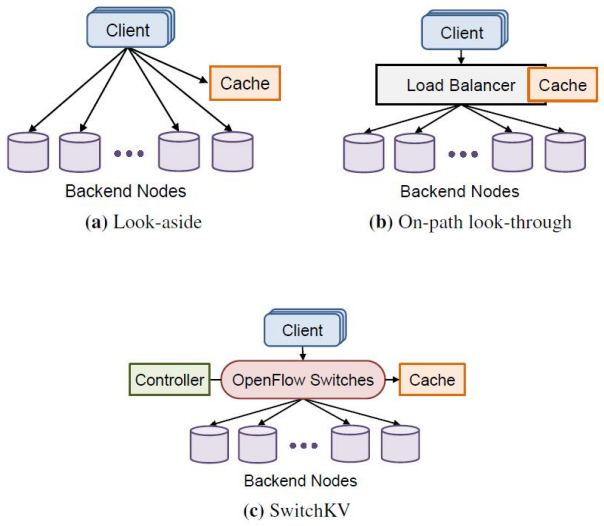

图 5 三种前端缓存架构

小型缓存解决方案无法扩展到多个集群。每个群集使用一个缓存节点只能提供群集内负载平衡，而不能提供群集间负载平衡。对于跨多个集群的大规模存储系统，集群之间的负载会不平衡。但是，增加另一个缓存节点是不够的，因为缓存机制要求缓存处理对 O(nlogn)个热点数据的所有查询。

因此，它需要另一个具有多个缓存节点的缓存层来实现集群间负载平衡。挑战在于缓存分配。仅仅将热点数据复制到所有缓存节点会导致缓存一致性的高开销。另一方面，简单地在缓存节点之间划分热点数据会导致缓存节点之间的负载不平衡。在高度偏斜的工作负载下，系统吞吐量仍将被一个缓存节点瓶颈限制。因此，关键是仔细地划分和复制热点对象，以避免缓存节点之间的负载不平衡，并减少缓存一致性的开销。

Liu 等人[11]于 2019 年提出了一种新的分布式缓存机制 DistCache，为大规模存储系统提供了可靠的负载平衡。DistCache 支持“一个大缓存”结构，即一组快速缓存节点充当单个超快速缓存。DistCache 通过多层缓存拓扑和查询路由共同设计缓存分配。关键思想是使用独立的哈希函数在不同层的缓存节点之间划分热点数据，并应用 power-of-two-choices 负载均衡机制[12]实现自适应路由查询。

使用独立的哈希函数进行缓存分区可以确保如果缓存节点在一层中过载，那么该节点中的热点数据集将以高概率被分配给另一层中的多个缓存节点。这种思想由对利用展开图和网络流的严格分析支撑，即证明了存在一种在不同层之间拆分查询的解决方案，这样在任何层中都不会有缓存节点过载。此外，由于热点数据只在每一层复制一次，因此缓存一致性的开销最小。

使用 power-of-two-choices 进行查询路由能够高效、分布式、在线地在各层之间进行分割查询。查询基于缓存负载以分布式方式路由到缓存节点，无需中央协调，也不知道分割查询的最佳方案。文中利用排队论证明它是渐近最优的。

DistCache 是一种通用缓存机制，可应用于许多存储系统，例如，基于固态硬盘的存储(如 SwitchKV)的内存缓存和基于交换机的缓存，用于像 NetCache[13]这样的内存存储。文中提供了一个具体的系统设计来横向扩展网络缓存，以展示远程缓存的强大功能。设计了控制层和数据层来为新兴的基于交换机的缓存实现 DistCache。控制器的可扩展性很高，因为它不在关键路径上，只负责计算缓存分区，不参与处理查询。每个缓存交换机都有一个本地代理来管理自己分区的热对象。

数据平面设计利用了可编程交换机的能力，并创新性地使用了超越传统网络监控的网络内遥测技术来实现应用级功能——通过在数据报报头中piggybacking 来分散高速缓存交换机的负载，以增强 power-of-two-choices，应用两阶段更新协议来确保缓存一致性。

3.2 单节点热点问题

内存键值存储被广泛用于缓存热数据，以解决磁盘存储或分布式系统中的热点问题。然而，内存KVS 内部的热点问题通常被忽略。

针对这个问题 Chen 等人[1]提出了 HotRing，一个热点感知的内存中 KVS，它利用了一个哈希索引，该索引针对一小部分数据(即热点数据)的大规模并发访问进行了优化。最初的想法是使查找项目所需的内存访问与其热度相关，也就是说，越热的数据读取越快。在其中有两点挑战：热点转移问题——热点是不断变化的。用有序环结构替换哈希索引中的冲突链，这样当热点转移时，头指针可以直接指向热点项目，而不会影响正确性。此外，使用一种轻量级机制来检测热点转移。大规模并发——热点本来就是由大规模并发请求访问的，需要为它们维持高并发性。采用了一种无锁设计，并对其进行了扩展，以支持 HotRing 所需的所有操作，包括热点移动检测，头部指针移动和重新哈希。

HotRing 在传统链式哈希索引基础上，实现了有序环式哈希索引设计。如图 6 所示，将冲突链首尾连接形式冲突环，保证头指针指向任何一个项目都可以遍历环上所有数据。然后，HotRing 通过无锁机制移动头指针，动态指向热度较高的项目（或根据算法计算出的最优项目位置），使得访问热点数据可以更快的返回。

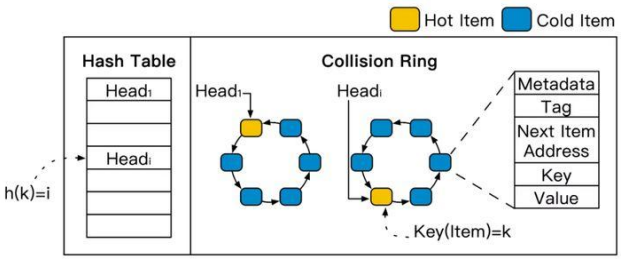

图 6 hotring 结构

1. 有序环设计

实现环式哈希索引后，第一个问题是要保证查询的正确性。若为无序环，当一个 read miss 操作遍历冲突环时，它需要一个标志来判断遍历何时终止，否则会形式死循环。但是在环上，所有数据都会动态变化(更新或删除)，头指针同样也会动态移动，没有标志可以作为遍历的终止判断。

利用 key 排序可以解决这个问题，若目标 key 介于连续两个 item 的 key 之间，说明为 read miss 操作，即可终止返回。由于实际系统中，数据 key 的大小通常为 10~100B，比较会带来巨大的开销。哈希结构利用 tag 来减少 key 的比较开销。tag 是哈希值的一部分，每个 key 计算的哈希值，前 k 位用来哈希表的定位，后 n-k 位作为冲突链中进一步区分 key 的标志。为了减小排序开销，构建字典序：order = (tag, key)。先根据 tag 进行排序，tag 相同再根据 key 进行排序。

2. 热点转移识别

在有序环散列索引中，查找过程可以很容易地确定是否有命中或未命中。剩下的问题是当热点移动时，如何识别热点并调整头指针。为了获得良好的性能，必须考虑两个指标，即识别精度和反应延迟。热点识别的准确性由已识别热点的比例来衡量。反应延迟是新热点出现的时间和成功检测到它的时间之间的时间跨度。考虑到这两个指标，首先引入随机移动策略，以极低的反应延迟识别热点。论文提出了一种统计采样策略，该策略以相对较高的反应延迟提供了高得多的识别精度。

随机移动策略：每 R 次访问，移动头指针指向第 R 次访问的 item。若已经指向该 item，则头指针不移动。该策略的优势是不需要额外的元数据开销且不需要采样过程，响应速度极快。实现简单开销小.但是当环中存在多个热点项目时，只能处理一个热点项目，可能会产生头节点的频繁移动，导致该策略会非常低效（工作负载倾斜较小，选择的周期 R 较小）。

采样分析策略：每 R 次访问，尝试启动对应冲突环的采样，统计 item 的访问频率。若第 R 次访问的item 已经是头指针指向的item，则不启动采样。这种方法适用于环中存在多个热点项目的情景，选出的 hot item 更可靠但是实现相对复杂，采样统计开销比较大。

采样所需的元数据结构如图 7 所示，分别在头指针处设置 Total Counter，记录该环的访问总次数，每个 item 设置 Counter 记录该 item 的访问次数。因为内存指针需要分配 64bits，但实际系统地址索引只使用其中的 48bits。使用剩余 16bits 设置标志位(例如 Total Counter、Counter 等)，保证不会增加额外的元数据开销。该策略的优势是，通过采样分析，

可以计算选出最优的头指针位置，稳态时性能表现更优。

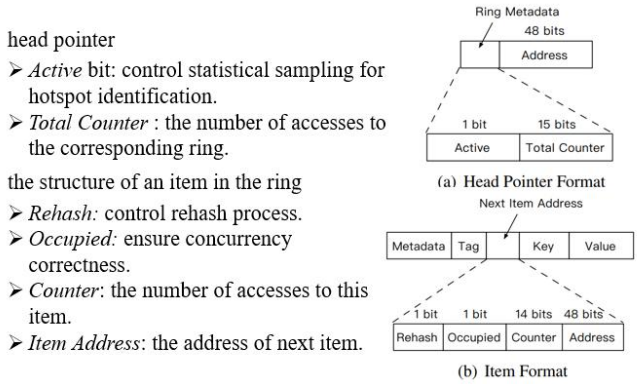

图 7 hotring 元数据结构

采样流程如下：

（1）打开 head.active

（2）后续的请求的访问记录会被记录到头指针的 total_count 和对应 item 的的 next.count，采样个数是 R

（3）采样结束后，将头指针的采样标记恢复过来

（4）采样完毕，得到每个 item 的访问次数 nk，环的总访问次数为 N，计算每项收益，选收益最小的为新的头节点，收益公式如下：

$$W_t = \sum_{i=1}^{k} \frac{n_i}{N} * [(i-1) mod\ k]$$

即计算所有项到该项的距离乘以该项的访问频率，求得每一项的期望值，选择最小的期望值作为新的头节点，最终就是访问频率越小的离该节点越远

在单线程和多线程情况下，对几种数据结构的性能进行了测试，Hotring 在大量读操作的情况下，可以实现一个很高的性能；链或者环中数据项的个数即便很多，Hotring 也可以保持一个很好的性能；Hotring 在数据访问呈现严重倾斜的情况下，也能保持非常好的性能。

5 总结及展望

热点问题在分布式存储系统中普遍存在，如何解决热点问题使得系统在高度倾斜的工作负载下保持一定的性能已经成为了分布式存储系统的一大挑战。针对解决热点问题的方法，作了以下总结：

（1）在集群范围内解决热点问题得到了很大的发展，主要的技术有一致性哈希、数据迁移、数据副本、前端数据缓存。推测可能是由于如今的系统以横向扩展为主，所以大多数的工作都是针对集群内节点之间的问题而忽略了节点本身的性能提升。

（2）一致性哈希实现虽然很简单但是由于其本身的限制只能针对静态负载问题，对于随时转移的热点问题不是很适用。

（3）数据迁移和数据副本实现开销比较大，需要支持迁移、复制、路由查询、一致性维持等开销。

关于单节点热点问题的解决方案比较少，未来可以更多地关注节点本身解决热点问题地能力，比如研究可以用于解决热点问题地索引结构，新型非易失性存储器技术和更快的网络堆栈等。

参 考 文 献

[1] Chen J, Chen L, Wang S, et al. Hotring: A hotspot-aware in-memory key-value store[C]//18th {USENIX} Conference on File and Storage Technologies ({FAST} 20). 2020: 239-252.

[2] Jin H, Li Z, Liu H, et al. Hotspot-aware hybrid memory management for in-memory key-value stores[J]. IEEE Transactions on Parallel and Distributed Systems, 2019, 31(4): 779-792.

[3] 李浩. 面向航天遥测数据的分布式存储系统负载均衡控制研究[D].重庆大学,2019.

[4] Panigrahy R. Relieving hot spots on the world wide web[D]. Massachusetts Institute of Technology, 1997.

[5] Li J, Nie Y, Zhou S. A Dynamic Load Balancing Algorithm Based on Consistent Hash[C]//2018 2nd IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC). IEEE, 2018: 2387-2391.

[6] Taft R, Mansour E, Serafini M, et al. E-store: Fine-grained elastic partitioning for distributed transaction processing systems[J]. Proceedings of the VLDB Endowment, 2014, 8(3): 245-256.

[7] Cheng Y, Gupta A, Butt A R. An in-memory object caching framework with adaptive load balancing[C]//Proceedings of the Tenth European Conference on Computer Systems. 2015: 1-16.

[8] Li J, Nelson J, Michael E, et al. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories[C]//14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20). 2020: 387-406.

[9] Fan B, Lim H, Andersen D G, et al. Small cache, big effect: Provable load balancing for randomly partitioned cluster

services[C]//Proceedings of the 2nd ACM Symposium on Cloud Computing. 2011: 1-12.

[10] Li X, Sethi R, Kaminsky M, et al. Be fast, cheap and in control with switchkv[C]//13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16). 2016: 31-44.

[11] Liu Z, Bai Z, Liu Z, et al. Distcache: Provable load balancing for large-scale storage systems with distributed caching[C]//17th {USENIX} Conference on File and Storage Technologies ({FAST} 19). 2019: 143-157.

[12] Mitzenmacher M. The power of two choices in randomized load balancing[J]. IEEE Transactions on Parallel and Distributed Systems, 2001, 12(10): 1094-1104.

[13] Jin X, Li X, Zhang H, et al. Netcache: Balancing key-value stores with fast in-network caching[C]//Proceedings of the 26th Symposium on Operating Systems Principles. 2017: 121-136.