基于可编程交换机的键值存储系统高效监控系统

设计文档

所在赛道：A

目录

一. 研究内容

1.1 研究背景

分布式键值存储[1][2][3]是如电商、社交网络等在线服务的关键基础设施，对于高性能在线服务至关重要。以电子商务为例，在“双 11”电商的限时抢购活动中，用户在读写存储服务器的商品内容时，均以键值对的形式对存储内容进行读写。当对商品进行浏览和购买时，每个商品可被视为多个键值对，键为商品的识别号，值包含商品的名称、描述、价格、库存等方面的信息。顾客下单时，客户端从存储服务器中获取商品库存信息，写入订单的内容。这些都需要大量应用到分布式键值存储系统，以满足严格的服务级别目标。

实践中，**键值存储系统可能会面临在高度倾斜和动态的工作负载下无法满足严格的 SLO[4][5]的问题**。例如，当一些数据被频繁访问，导致这些数据成为热点数据，而其他的数据被访问较少，从而造成数据倾斜[6][7]，这种情况下，处理热点数据的负载过高，会导致无法满足 SLO；比如如果一个电商平台的商品详细页上的热点数据可能是某些畅销商品的数据，那么访问这些畅销商品的请求可能会在某一键值存储节点上集中，从而导致该节点的负载过高，影响整个平台的性能。而在高度动态的工作负载下，键值存储系统需要不断的扩容以适应变化的负载，但是，扩容需要一定的时间和资源，因此可能会导致一定的延迟和服务不可用，从而导致无法满足 SLO。

同时，在实际应用当中，将要面对的常常并非是一种而是多种类型的工作负载，不同类型的负载会导致数据的访问模式、访问频率、

数据大小等方面的变化，因此需要针对不同类型的负载来进行相关的优化和调整。但是现阶段所应用的解决方案大多数都存在着很明显的局限性：一方面，应用这些方法可能无法同时感知可编程交换机上的多类型负载。另一方面，即使它们成功的感知了多类型负载，也可能会因为其所存储的巨大关键字空间迅速耗尽内存预算和控制平面带宽从而消耗大量的交换机资源而导致网络性能急剧下滑。

为了解决上文提到的现有系统无法对多类型负载进行有效的感知和监控以及在高度倾斜和动态的工作负载下无法满足 SLO 的问题，本项目将基于可编程交换机[8][9][10]部署 Lake 监控系统（**将在第二、第三部分进行具体阐述**）进行实践证明和应用设计，基于该系统来实现多类型负载感知，并为用户提供可视化的服务器负载优化选择，以此来为用户提供优化网络键值存储系统多类型负载感知的解决办法。同时，**Lake 监控系统也可以支持网内键值存储系统[2][3]，**这更加提高了系统的性能和效率。

1.2 研究内容

（1）利用可编程交换机优化分布键值存储过程中多类型负载感知低效的问题。Lake 使用可编程交换机来实现网络中的键值存储和流量调度功能。可编程交换机可以通过编程进行自定义的网络流量处理，因此 Lake 使用 P4 语言[11]编写了一组程序，以使交换机支持键值存储和多类型负载感知。这些程序被加载到可编程交换机中，从而使 Lake 实现了快速的键值存储和负载感知功能。此外，Lake 还使用 P4 控制平面来管理交换机，并与上层网络进行通信，以便进行负载感知

和流量调度。

（2）网内键值存储系统。Lake 使用网内键值存储系统作为其底层存储引擎，以实现高效的数据访问和管理。Lake 利用了可编程交换机上的硬件支持，实现了高速的、低延迟的键值存储服务。在网络中，Lake 将键值存储系统部署在可编程交换机上，通过数据包处理管道将存储请求和数据传输直接在网络中完成，避免了数据传输时的额外开销。此外，Lake 还利用了可编程交换机的多级缓存功能，将部分数据缓存到可编程交换机上，从而进一步降低了访问延迟。通过这种方式，Lake 能够实现高速、低延迟的键值存储服务，从而满足大规模数据处理的需求。

（3）对于分布式键值存储网内监控系统的可视化应用。利用 munin[12]插件和网页构建对 Lake 模型进行可视化应用。建立网页界面为用户提供可靠的加速方式和监控机制。

1.3 应用场景和价值

（1）应用场景

如 1.1 所述，本项目主要应用于电商、搜素引擎、社交网络等需要高性能在线服务的平台。Lake 正是为了解决如今的键值存储系统对于多类型负载感知所存在的局限性。

（2）价值

学术价值：1、解决了多类型负载感知问题：Lake 提出了一套完整的解决多类型负载感知的方案，包括感知、监控和优化等多个环节。这一方案的创新点在于能够有效地感知和处理多种不同类型的流量，

从而使得网络的性能和资源利用率都得到了显著提升。

2、推动了可编程交换机和键值存储系统的研究：Lake 的实现基于可编程交换机和键值存储系统，这两种技术都是当前网络领域的研究热点。Lake 的成功应用推动了这两种技术在实际网络中的应用和研究，同时也为其他研究者提供了一个很好的思路和方法。

3、为数据中心网络的设计和优化提供了借鉴：Lake 是一种新型的数据中心网络架构，其设计和优化思路都可以为其他数据中心网络的设计和优化提供借鉴。例如，Lake 中的负载感知和调度技术可以应用于其他数据中心网络中，以优化网络性能和资源利用率。

社会价值：Lake 框架可以帮助电商、搜索引擎等应用在提高系统性能的同时，具备更好的扩展性和可靠性。通过多类型负载感知系统，Lake 可以更好的针对不同类型的负载进行调度和资源分配，从而提高系统的吞吐量和响应速度。Lake 在电商应用中可以提高商品搜索和推荐的效率，使得用户能够更快速的找到自己需要的商品。在搜索引擎中，Lake 可以提高搜索结果的响应速度，从而提升用户体验。在实验中我们得到，Lake 相比于已存的系统，在吞吐量和平均响应时间两个指标上都有着至少 3 倍的提高。

二. 背景和动机

2.1 可编程交换机

可编程交换机是一种网络设备，它能够在数据包转发过程中执行用户定义的程序，以实现网络行为的配置和调整。可编程交换机现在已经被十分广泛的应用在各个方面，如网络流量控制、安全和远程监

测等。与以往的交换机相比，可编程交换机具有以下几个特点：

1.灵活性和可编程性：可编程交换机允许用户定义数据包处理逻辑和控制方法，以满足不同的应用需求，并且能够随时修改和更新。

2.高效性：可编程交换机的处理速度要比通用处理器快得多，因为其处理逻辑是集成在硬件中的。

3.可扩展性：可编程交换机可以与其他网络设备组合使用，以扩展其处理能力或添加其他功能。

2.2 键值存储中的偏斜工作负载

当将分布式键值存储扩展到数十亿客户端时，面临的主要挑战是在高度不均衡和动态的工作负载下满足严格的 SLO。具体来说，(1)高度不均衡时，受欢迎的项目接收的请求数量（每天数百万）远远多于其他项目（几乎没有），(2)对于动态分布，受欢迎项目的集合动态变化，很快变得不受欢迎<10min。受欢迎的项目可能会突然聚集到少数服务器上，导致服务器超载和性能下降。实际上键值存储系统的工作负载通常使用 Zipfian 分布进行建模，通常表现出高度不均衡的模式（例如，α>1 的 Zipf 分布[13]）。键值存储系统将项目分区并分发到多个服务器以确保 SLO。不均衡的工作负载主要包括不均衡的内容负载、网络负载和存储负载。这些不均衡的负载可能会超载少部分资源，这可能成为新的瓶颈并引入更长的尾部延迟。过度提供资源可以减少性能惩罚，且显著增加总体成本[3]。

2.3 网内键值存储

网内键值存储系统利用新兴的可编程交换机来路由、缓存和复制流行的数据项。通常，这些系统由于交换机可见性（即请求聚合点）而能够感知多类型负载，并由于高性能的交换机 ASICs（即>10Tbps[14]）而能够实时平衡负载。这些系统可以为键值存储提供高吞吐量、低延迟和可扩展性。接下来，本项目将详细阐述每个典型的工作流程，分为两个阶段：负载感知和负载均衡。

2.3.1 网内缓存。NetCache[1]将 ToR（顶层交换机）作为负载平衡缓存（中等缓存命中率<50%），吸收热点项，并使存储服务器上的负载更加均匀。在负载感知阶段，它维护一个 CMsketch 来检测热门未缓存的键和一个 Bloom 过滤器来在数据平面消除复制的键。在每个时间窗口结束时，将统计信息报告给控制平面，在那里控制器确定缓存集。在负载平衡阶段，数据平面中的缓存集会被新的缓存集更新。缓存的键直接回复读取请求到客户端，而写请求被转发到服务器以确保一致性。

2.3.2 基于内容的路由。SwitchKV[2]将存储节点分为慢存储节点和内存缓存节点，并维护每个键的地址，以便可以快速路由请求到存储节点。在负载感知阶段，它为每个后端存储节点保持最近访问键的本地频率计数器。对于频率大于阈值的键，SwitchKV 定期将最近的热键和突然变得非常热的键报告给缓存节点。在负载平衡阶段，内存缓存节点添加或驱逐选定的键，并直接将请求路由到相应的存储节点。

2.3.3 网内复制是指 Pegasus[3]通过将热点数据复制到多个存

储服务器来分配负载。它在保证数据一致性和协调性的同时，重新平衡数据副本。交换机维护一个网内一致性目录，跟踪和管理数据副本的位置。在负载感知阶段，它对未复制键的请求进行采样，并将其转发到交换机 CPU，在该程序中计算访问频率。每个服务器收集负载统计信息并报告给控制器，在那里，两个独立的寄存器数组跟踪每个副本键的读/写计数。在负载平衡阶段，控制器基于负载统计数据选择副本。

三. 设计思路和理论分析

3.1 总体设计思路

Lake 模型主要由以下几个模块组成，并且每一模块所解决的问题如下。

1、数据分发模块：负责将数据分发到各个计算节点，并根据数据的特点进行负载均衡。该模块可以根据数据的大小、类型、访问频率等信息进行动态调整，以达到更好的负载均衡效果。这解决了在高度倾斜和动态的工作负载下无法满足严格的 SLO 的问题。

2、负载感知调度模块：根据系统的负载情况，动态调整计算节点的任务分配，以提高整个系统的处理能力。该模块可以根据不同类型的任务进行调度，比如将高计算密集型任务分配给计算能力更强的节点，将高带宽需求的任务分配给网络带宽更宽裕的节点等。这一模块解决了之前存在的对于多类型负载感知无法感知或效率低下的问题。该模块能够根据负载的类型和需求，合

理地分配计算资源，并确保任务能够在所需的时间内完成。同时，该模块还可以监控系统的负载状况，及时地调整任务的调度，从而避免系统出现瓶颈，提高系统的性能和可靠性。

3、网内键值存储模块：在网络层上实现键值存储系统，可以提供高效的数据访问和管理。该模块通过网络层直接访问数据，避免了数据传输过程中的开销，从而提高系统的处理速度。

4、资源管理模块：负责管理整个系统的资源分配，包括计算资源、存储资源、网络带宽等。该模块可以根据系统的负载情况和用户需求，动态调整资源的分配情况，以实现更好的资源利用率。这有助于提高 Lake 的吞吐量和降低延迟。

5、系统监控模块：对系统的各个组件进行实时监控，以检测系统中的异常情况并进行处理。该模块可以监控系统的运行状态、负载情况、资源利用率等指标，以实现系统的自我调节和优化。这有助于从整体上对于 Lake 的执行进行优化。

3.2 Lake 模型建立

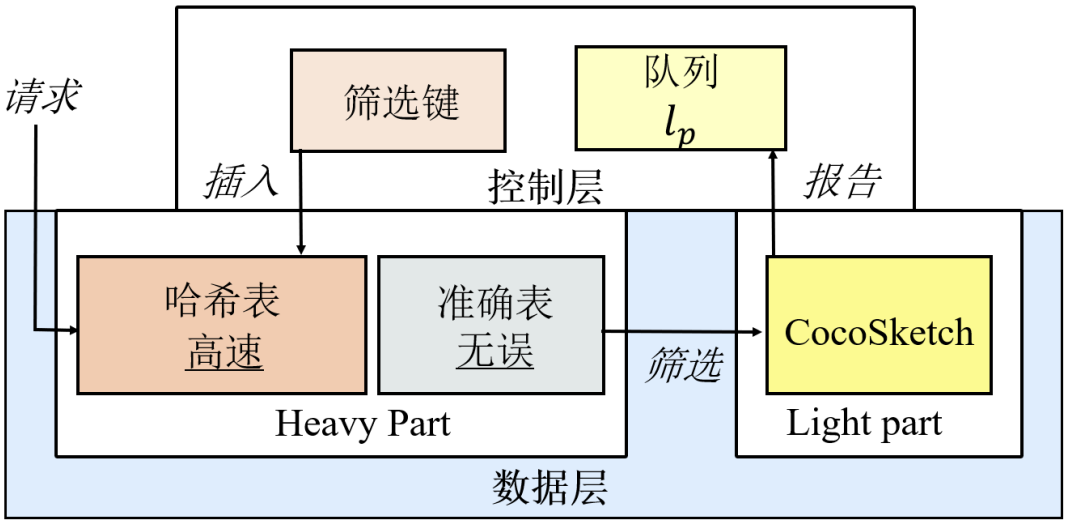


图 1Lake 简单模型

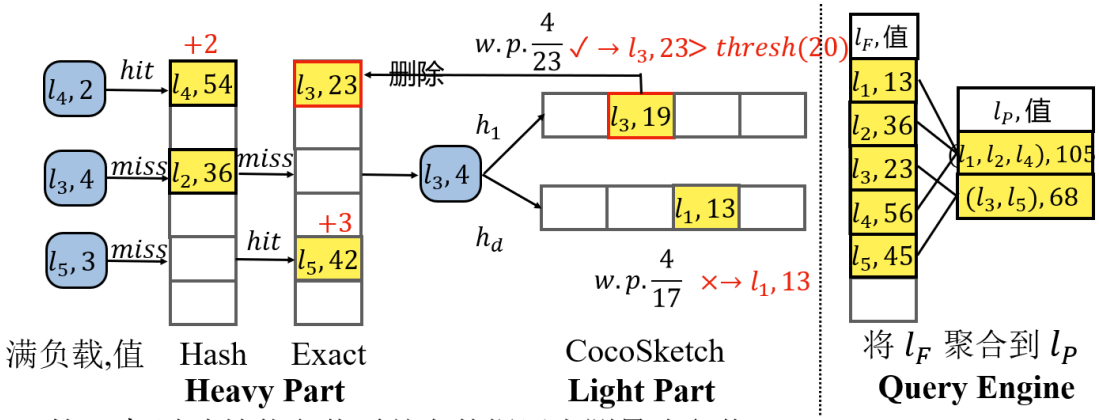
本项目旨在没有复杂一致性机制的情况下高效的感知多类型负载。同时，本项目希望用户可以灵活的定制负载感知。为了更好的实现这个目标，本项目采用了 Lake 模型。

Lake 的主要实现部分包括一个重型部分和一个轻型部分。

为了实现多类型负载感知的构建，**Lake 会从轻型部分筛选出其中的重负载，从而确保重负载的保真度。**于此同时，Lake 将替换掉 Bloom 过滤器，以减少重负载的重复报告，从而降低控制平面带宽的消耗。

在轻型部分中，**本项目最小化 LF 的所有子集和的随机方差（利用 3.4 提及的随机方差最小化），以减少所有 LP 的估计误差。**具体的操作是在轻型部分使用 CocoSketch，来报告并移除重负载(即计数超过阈值的负载)到控制平面。随后，将重载插入重型部分。

对于每个负载，**重型部分直接在数据平面中以增加其计数，以此来越过轻型部分。**这样，重型部分计数重负载的误差就可以忽略不计。这个操作同时也提高了重负载感知的保真度。此外，**这个操作也能够减少轻型部分的哈希碰撞。**



第一步:过滤掉热负载再并在数据层中测量冷负载 第二步: 在控制层查询l_P

图 2 Lake 具体实现

如上图所示，数据结构由两部分组成：**记录热点负载的重部分和记录冷负载的轻部分。**重部分 H 包含一个高速哈希表和一个无误差的精确表。重部分的每个桶记录一个负载的计数：负载键和负载值；轻部分则是一个支持多类型负载查询的 sketch（例如 CocoSketch）。（注：CocoSketch 维护 d 个包含 l 个键值对的数组，每个数组与一个独立的哈希函数相关联。）

3.3 随机方差最小化

由于将热负载和冷负载分离，Lake 在大多数情况下具有高准确性。在分析之前，本项目将要定义如下符号。本项目用 $l(e)$ 表示负载 e 的真实值，用 $\hat{l}(e)$ 来表示 e 的估计值，用 (e_i, w) 来表示每个新的负载和其值。

随机方差最小化。本项目首先分析负载方差和的最小化。本项目的最终目标是最小化负载综合的估计。

$$minimize \sum_{e} (l(e) - \hat{l}(e))^2 (1)$$

在轻量级部分，本项目需要考虑每个插入的新方差所导致的负载方差和的逐渐最小化。

$$minimize \sum_{e} \Delta(l(e) - \hat{l}(e))^2 (2)$$

本项目确保每次插入引起的方差变化最小化，以便 Lake 可以减少整体估计多类型负载误差。

引理 1。更新桶 (e_j, l_j) 的最小方差增量为

$$\sum_{e} \Delta(l(e)-\hat{l}(e))^2 = \begin{cases} 2wl_j, e_i \neq e_j \\ 0, e_i = e_j \end{cases} (3)$$

证明。对于重负载，每次在重负载中插入数据不会改变方差和，因为它具有无误差统计数据。因此，可以保证每次 Lake 的插入都会使方差和增量最小化。

错误界限。然后，本项目讨论 Lake 的估计误差。考虑一个 *d** *l* 大小的 CocoSketch 在轻量级部分。本项目使用 *R*(*e*)来表示负载 *e* 的相对误差。下面的定义证明的 *R*(*e*)在轻量级部分的误差界。

引理 2。让$l = 3 \cdot \epsilon^{-2} and d = O(log\delta^{-1})$。对于多类型负载$l_p < l_F$中的任意负载 e，有

$$\mathbb{P}\left[R(e) \geq \epsilon \cdot \sqrt{\frac{f(\bar{e})}{f(e)}}\right] \leq \delta (4)$$

证明。对于存储在重部分中的热负载，只有在插入到重部分之前在轻部分中发生哈希冲突才会产生错误。由于轻量级部分被冷负载所主导，哈希冲突较少，因此产生的错误也较少。同时，轻负载中的值占热负载的比例很小，因此热负载的误差可以忽略不计。在最坏的情况下，即重部分中的桶在插入后未更新。通过定理 2 仍然可以获得 R(e)的误差界。

定理 1。让$l = 3 \cdot \epsilon^{-2} and d = O(log\delta^{-1})$。对于多类型负载 *kp* <*kF* 中的任意负载 e，有

$$\mathbb{P}\left[R(e) \geq \epsilon \cdot \sqrt{\frac{f(\bar{e})}{f(e)}}\right] < \delta(5)$$

总的来说。可以得出结论，Lake 的 $R(e)$的误差界可以通过定理 3 得到。

3.4 优化方案

Lake 提供了高效的多类型负载感知技术，适用于网络中的键值存储。基于负载统计数据，接下来，本项目综合并确定了 4 种优化技术。本项目的目标不是提出一种负载均衡的最优算法，而是展示使用 Lake 进行简单调度的负载均衡的好处。对于三种类型的负载，本项目利用这些结果来解决性能瓶颈。此外，本项目定义了每种技术的原语，即基本和自动的 P4 代码。本项目在表 1 中列出了技术的适用性。

NO	负载类型	负载感知	负载平衡	阈值	原语	适用性
优化1	网络负载	可用的带宽	首先安排利用率低的通道	T_c	load-bal	缓存、路由、复制
优化2	存储负载	服务器负载	首先安排利用率低的服务器	T_S	re-route	路由、复制
优化3	存储负载	进程负载	首先安排利用率低的进程	T_p	re-route	路由
优化4	内容负载	写入密集型项目	清除缓存中的写入密集型项目	T_w	filter_key	缓存、路由

表 1Lake 优化

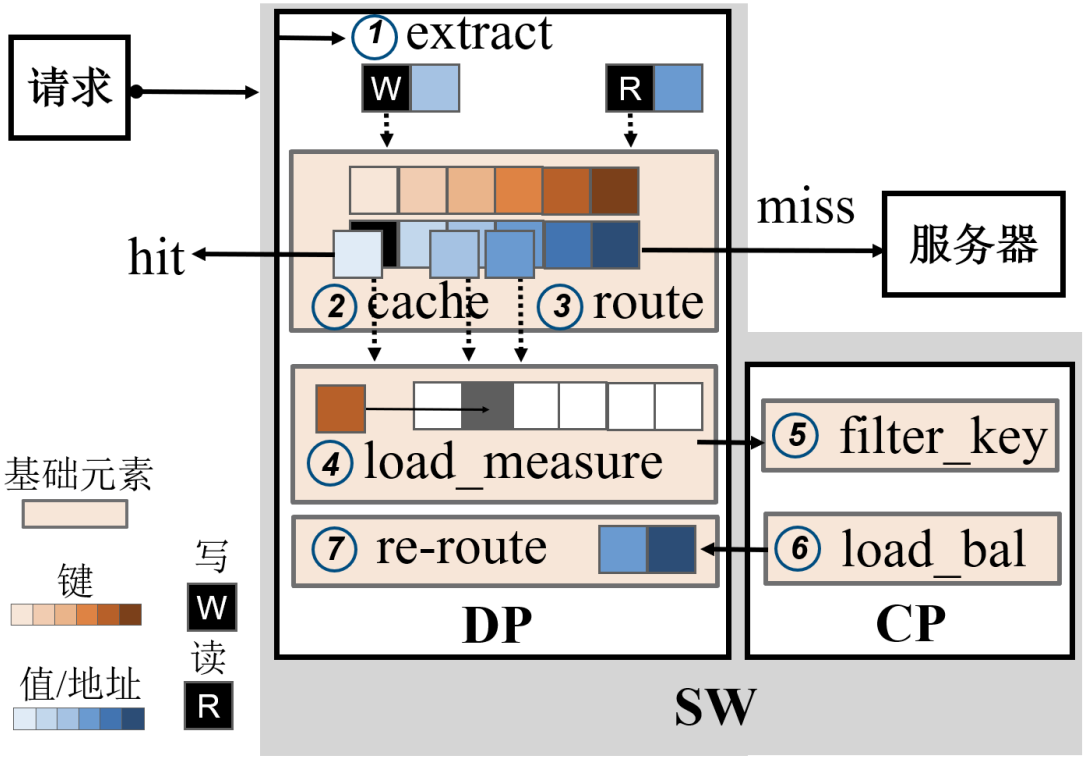


图 3 优化方案

3.4.1 优化 1 通道平衡：首先安排未利用的通道以避免网络拥塞。由于网络拥塞可能会降低系统吞吐量，Lake 计算每个通道的请求次数，并将请求安排到未利用的通道中以避免拥塞的通道。具体来说，超过阈值 Tc 的访问次数的通道被认为是过载的。然后，Lake 将请求安排到那些未利用的通道中。

3.4.2 优化 2 服务器平衡：首先安排未利用的服务器以平衡存储负载。本项目计算服务器负载并将请求安排到未利用的服务器上。具体来说，被识别为过载的服务器是访问频率超过阈值 T_s 的服务器。本项目重新路由请求到未利用的服务器上（原语 re-route）。在存储服务器中，键可以有多个副本，服务器平衡只修改服务器访问的优先级以确保正确性。

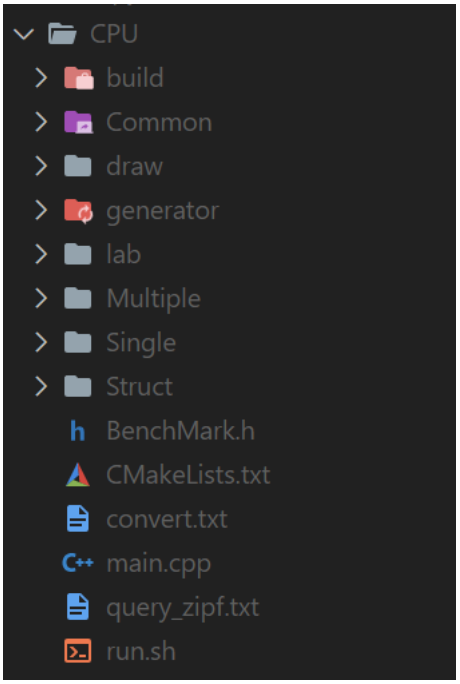
3.4.3 优化 3 进程平衡：首先安排未利用的进程以平衡存储负载。类似于上一种优化方式，本项目计算并安排存储负载，但存储节点是一个进程。具体来说，阈值被记录为 T_p，原语为 re-route。进程平衡只修改进程访问的优先级以确保正确性。

3.4.4 优化 4 消除写入项目：消除缓存中的写入密集型项目，以计数更多的读取密集型项目。网络内部缓存的一个限制是它不能提高写入项的吞吐量。更糟糕的是，写入密集的项可能占用缓存，减少缓存读取密集型项的机会。因此，可以计算出频率超过阈值 T_w的写入项，并在网络内缓存中过滤它们（原语 filter_key）。写入密集型项目将触发缓存未命中并直接转发到服务器。这些写入密集型请求在交换机故障时会丢失，但这些请求会直接转发到服务器，不需要机制保证一致性。

3.4.5 将前四项一起使用。为了提供硬件优化，本项目识别出了基本操作，以便消除重复的代码并减少硬件资源的消耗。例如，本项目将前 4 中优化方式放在一起，首先从客户端中提取请求，然后将这些请求进行缓存或路由。然后，本项目使用 Lake 进行多类型负载感知（负载感知），并从控制平面的缓存中过滤出写入密集型键（04，filter_key）。接下来，控制器基于感知结果平衡负载（01，load_bal）。然后，本项目通过将这些项目重新路由到未被充分利用的服务器（02，re-route）或进程（03，re-route）来进行调度。

四. 方案实现

如图是本项目的文件目录树。接下来将会具体解释各个关键模块的功能，具体代码请见附录。



4.1 实验环境

测试环境包括两台服务器和一个 6.5Tbps Barefoot Tofino 交换机。每台服务器配备一个 20 核 CPU（Intel Xeon Silver 4210R）和 64 GB 总内存。两台机器都配备了一个 100G NIC（Mellanox MT27800）。一台机器作为客户端生成键值查询，并将请求发送到另一台作为键值存储服务器的机器上。

4.2 基础数据结构模块

4.2.1 哈希。Lake 使用哈希来存储和管理键值对，其主要运用于两个方面：

（1）将 key 映射到哈希中的一个桶，加速查找过程。

（2）桶中存储的是一个指针数组，该数组中的每个元素指向一条 KV 记录。主要在 Common/hash.h 文件中实现。

4.2.2sketch。Lake 中多次使用了 sketch，主要运用了 Cocosketch 和 CMsketch。在 Multiple/LASketch.h 文件和 Struct/CMSketch.h 文件中实现。

（1）Cocosketch。Lake 在 light part 使用了 Cocosketch 来实现高效的去重和计数操作。具体来说，lake 将每个数据项映射到 Cocosketch 中，每个位置报错了一个计数器和一个指向该计数器的指针。在插入新数据项中，lake 首先会在其中查找是否已经存在相同的数据项，如果存在，则直接增加该数据项对应的计数器;否则 lake 会尝试将新数据项插入到 Cocosketch 中，并利用 hash 来避免冲突。在删除数据项是，lake 会将响应的计数器减少，当计数器减少到 0 时，则将该数据项从其中删除，通过 Cocosketch，lake 可以以较小的空间开销实现高效的去重和计数操作。详情请见 4.3.2。

（2）CMsketch。在 lake 中，CMsketch 被用来实现频率估计功能，他用来估计每个数据块的出现频率，对于每个数据块，lake 会计算出其哈希值，并将其插入到 CMsketch 中。当需要查询某个数据块是否已经出现在数据集中出现过，Lake 会进行查询，如果该计数器值超过了阈值，那么就判断其出现过。核心代码请见附录。

4.2.3Priority Queue。Lake 使用 Priority Queue 来维护数据流中出现频率最高的元素。通过 Priority Queue，lake 能够高效的找到频率最高的元素，以及快速更新其元素。代码见

附录。

4.2.4Bloom Filter。在 Lake 中，Bloom Filter 用于过滤掉不可能匹配的 key，从而加速查询的过程。代码见附录。

4.3 Lake 实现模块

Lake 的数据结构由两部分组成：记录热点负载的重部分和记录冷负载的轻部分。其主要在 Multiple/LASketch.h 文件中实现。

4.3.1 **重部分** H 包含一个高速哈希表和一个无误差的精确表。重部分的每个桶记录一个负载的计数：**负载键和负载值**。

接下来是实现 heavy part 的关键代码，在 hash_tbl 中存储出现次数较多的数据项，而在 exact_tbl 中存储出现次数较少的数据项，他们的出现次数是准确的，即 exact count。因此，不需要再在轻量级计数器中继续进行计数，也不需要再随机采样，直接在 heavy part 中进行统计即可。

代码实现：

```
   if(hash_tbl[hash_position].count != 0 && hash_tbl[hash_position].ID
== item){
        hash_tbl[hash_position].count += 1;
       return ;
   } //判断当前数据项在 hash_tbl 中是否存在，如果存在，则将该数据项在
hash_tbl 中的计数加 1，并直接返回。
   else if(exact_tbl.find(item) != exact_tbl.end()){
       exact_tbl[item] += 1;
       return ;
   }//判断当前数据项在 exact_tbl 中是否存在，如果存在，则将该数据项在
exact_tbl 中的计数加 1，并直接返回。
```

4.3.2 **轻部分**是一个支持多类型负载查询的 sketch（例如 CocoSketch）。CocoSketch 维护 d 个包含 l 个键值对的数组。每个数组与一个独立的哈希函数相关联。

```
   for(uint32_t i = 0;  i < HASH_SUM; ++i){ //遍历哈希函数的数量
       uint32_t position = hash(item, i) % LENGTH;//计算第 i 个哈希函数的哈希值，然后确定其在计数器中的位置
       conter[i][position].count += 1;
       if(randomGenerator() % counter[i][position].count == 0)
       //该条件以 1/count 的概率选择要更新计数器的位置的 ID 值。这可以确保更新的 ID 与目前已存储的 ID 均匀分布，从而降低估计误差
           counter[i][position].ID = item;//将计数器位置的 ID 设置为当前 item
       if(counter[i][position],count > threshold){//如果计数器超过阈值，那么将其从 light part 移到 heavy part
           if(hash_tbl_size < HASHTBL_SIZE && hash_tbl[hash_position].count == 0){//如果哈希表还有空间，并且哈希表中该位置未被占有，则将该位置的计数器信息移动到哈希表中
               hash_tbl[hash_position] = counter[i][position];
               //将计数器信息赋值给哈希表中
               hash_tbl_size++;
               //增加哈希表的项数
               memset(&counter[i][position], 0, sizeof(counter));
               //将原来位置的计数器信息清零，以释放内存
           }
           else if(exact_tbl.size() < EXACTBL_SIZE){
           //如果哈希表已满，但精度表还有空间，则将该位置的计数器信息移动到精度表
```

```
                exact_tbl[counter[i][position].ID]                    =
counter[i][position].count;
                memset(&counter[i][position], 0, sizeof(counter));
                //将原来位置的计数器信息清零，以释放内存
            }
        }
    }
```

4.4 工具函数模块

在 Lake 的基础实现后，本项目提供了 Lake 解释器中的一个工具函数，其主要保存在 Common/utils.h 文件中。其实现了对于文件的读取、字符串的转换、输出的调试等功能。

五. 运行结果

5.1 硬件资源

Resource Type	NC1(%)	NC2(%)	LA(%)	R1(%)	R2(%)
Exact Match xbar	9.83	11.39	6.38	35.10	43.99
Hash Bits	5.37	7.93	4.81	10.43	39.34
Hash Dist Unit	20.83	31.94	13.89	33.32	56.51
SRAM	5.21	8.96	5.52	-5.95	38.39
Map RAM	7.81	14.06	7.12	8.83	49.36
TCAM	0.00	0.00	0.00	0.00	0.00
Meter ALU	14.58	22.92	22.92	-57.20	0.00
Stats ALU	0.00	0.00	0.00	0.00	0.00
Stages	58.33	91.67	41.67	28.56	54.54

在这里展示了 Lake 能够有效地计数多种类型的负载。本项目将 NetCache 的感知部分作为基准，即 CM Sketch 和 Bloom Filter，并消除其他功能。本项目遵循原论文中的相同配置。对于每个负载，本项目使用相同配置的 CM Sketch（遵循 A1）进行增量计数。如表 V 所示，本项目计数两种负载：（1）热键（NC1）和（2）热键和 *addrphy*（NC2）。NC2 消耗昂贵的 91.67%的管线阶段，并无

法编译 3 个负载（key_w）。而 Lake 在消耗一定硬件资源的同时，计数了所有 5 个负载。此外，Lake 在大多数情况下显著降低了资源消耗，除了 SRAM 和 Meter ALU 比 NC1 更高。但 Lake 的资源消耗不比 NC2 更高，同时还测量了 3 个以上的负载。由于随机变量最小化，Lake 实现了资源效率，从而降低了所有负载的误差。

5.2 负载感知

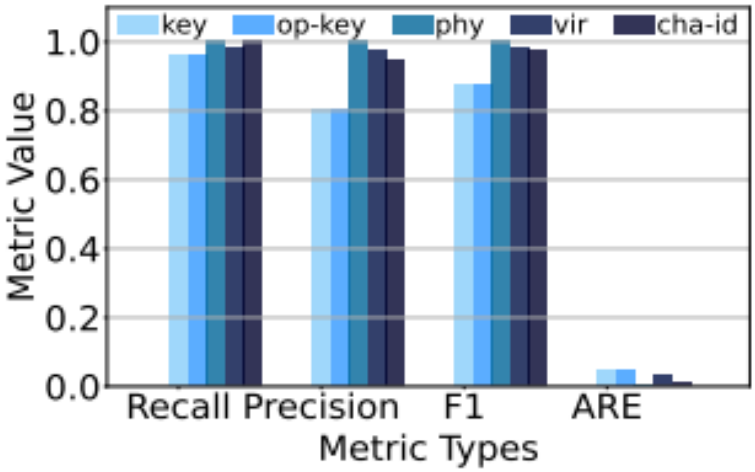


图 8 Lake 以低误差计算不同负载

本项目使用 Lake 计数 5 个负载。总体而言，Lake 保持高精度，但对于不同的负载表现不同。造成这种差异的原因是，具有较大键空间的负载可能消耗更多的内存占用，并引发更多的哈希冲突。此外，Lake 从轻负载中过滤出重负载，从而确保重负载具有高精度。

5.3 通信开销

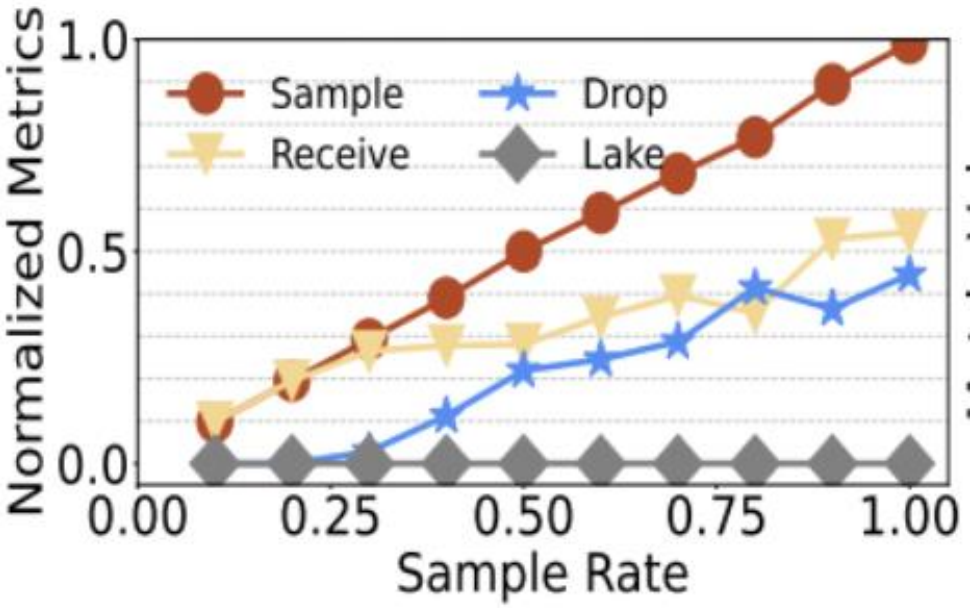


图 4 采样率

本项目使用 Pktgen 发送了 1M 个数据包，速率为 10K。Tofino 交换机的数据面采样数据包并将其报告给交换机操作系统。图 4 显示，随着采样率的增加，由于控制平面带宽的限制，交换机操作系统丢弃的数据包数量增加。交换机操作系统接收到的最大数据包约为 50%。这意味着只有约 50%的数据包会有益于感知结果。Lake 保持低丢包率，因为交换机操作系统只读取 Lake 寄存器，这消耗带宽非常有限。

5.4 可扩展性

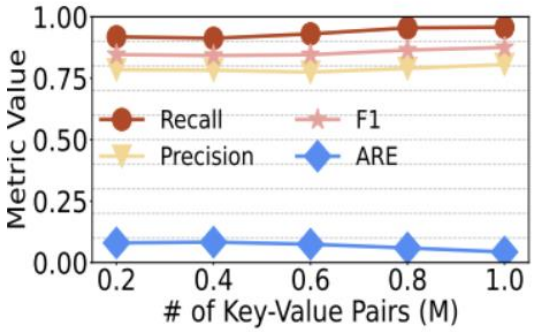


图 5Lake 键值对数量对于 Lake 性能的影响

为了测试 Lake 的可扩展性，本项目改变了请求数量。随着

请求数量的增加，估计误差保持稳定甚至略微减少。Lake 是可扩展的，因为重量部分防止多种类型的负载共享同一个计数器，从而可以准确地计算重载。此外，当键值对的数量增加时，重部分还可以增加准确性，因为当重载的计数增加时，负载可以被过滤到重部分。

5.5 多类型负载可扩展性

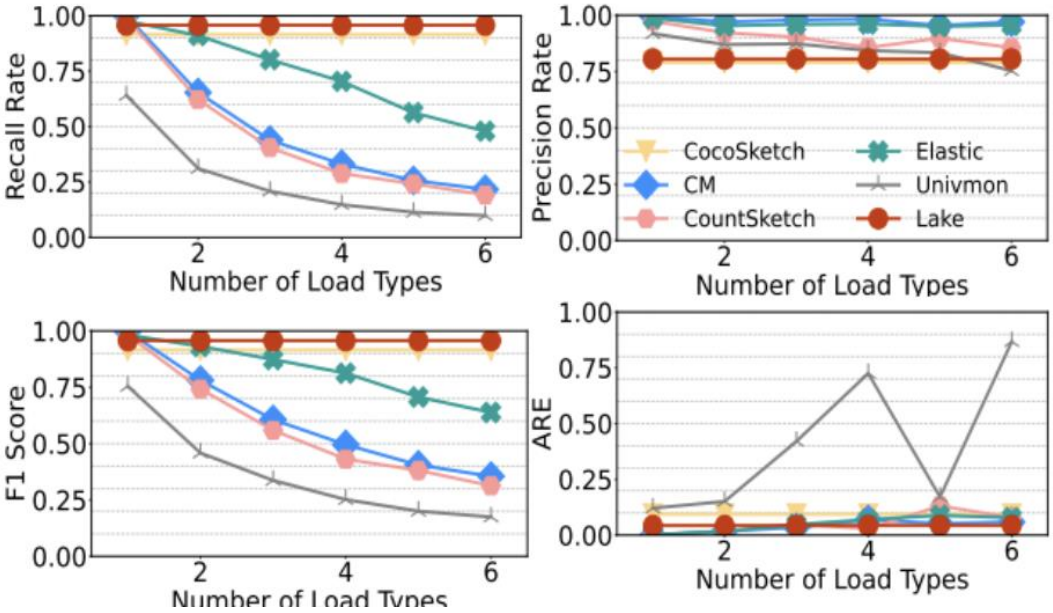


图 5 与其他优化方案比较

本项目逐步为每个草图添加新负载。随着负载数量的增加，一些单键草图(即 CM、Count sketch、Elastic sketch 和 UnivMon)会受到性能惩罚。尽管它们的精度比其他草图高，但几乎无法识别热负载（召回率较低）。Lake 和 CocoSketch 在所有准确性指标上都保持高可扩展性。Lake 始终具有比 CocoSketch 更高的性能，因为 Lake 将重载从轻部分中过滤出来，因此可以准确地识别重负载和轻负载。因此，基于网络的键值存储系统可以由于对热负载的准确识别而做出更精确的决策。

5.6 内存可扩展性

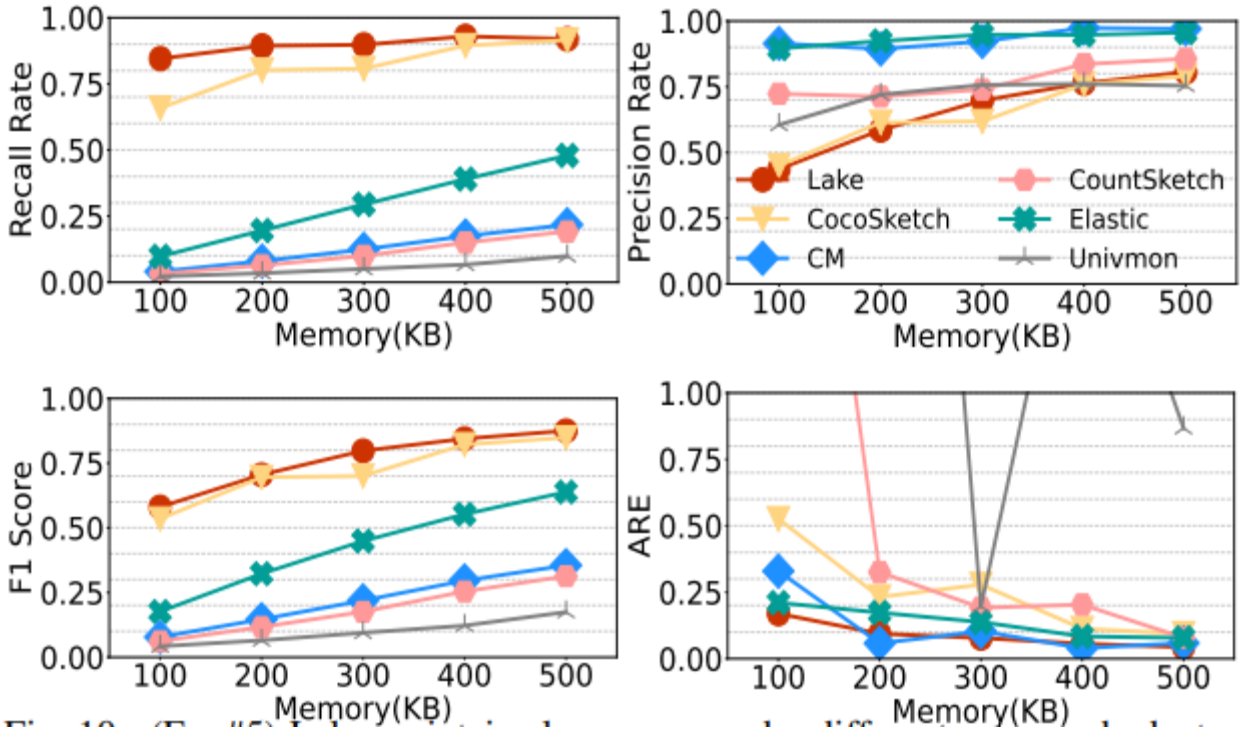


图 5 Lake 在不同的内存预算下保持低错误率

本项目改变内存预算，其他设置与实验 3 相同。如图 5 所示，Lake 在相同的内存预算下比其他草图更准确。此外，Lake 具有比 CocoSketch 更低的 ARE，因为重部分过滤了重负载从轻部分过来的两部分，以确保重负载的准确性。

5.7 工作倾斜负载

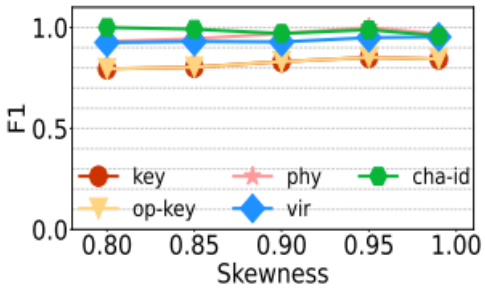


图 6 Lake 在不同负载偏度下保持低误差

本项目计算了 5 种不同的负载，以测试当请求的偏斜度从 0.8 变为 1 时，Lake 的鲁棒性。F1 保持稳定并略微提高。随着偏斜度

的增加，重负载的比例增加并向重负载部分移动。因此，当工作负载分布高度倾斜时，Lake 表现出鲁棒性。

5.8 写入比率

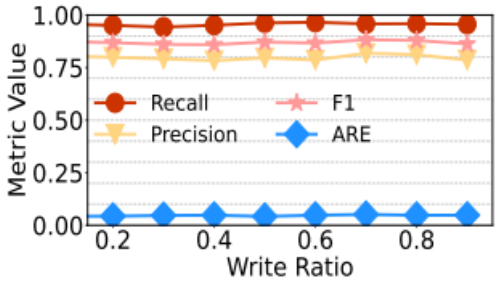


图 7 Lake 在不同的写率下保持低错误率

本项目在写入比率从 0.2 到 0.9 时计算每个度量值。每个度量值保持稳定，展现出在读密集型工作负载和写密集型工作负载中的高鲁棒性。当查询写密集型工作负载时，Lake 聚合相同操作的部分负载。操作的识别仅为 1 位，但 Lake 通过随机方差最小化进行了定制，用于子集和估计，并保持高精度。

5.9 负载平衡

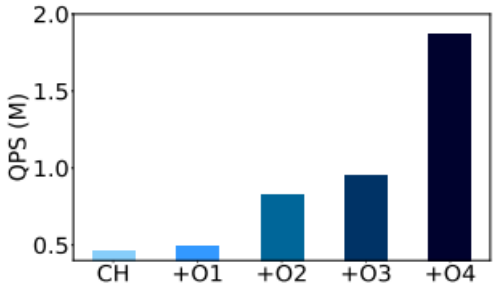


图 9 Lake 通过 O1-O4 提高了系统吞吐量

本项目证明简单的优化技术可以从网络内键值存储的已识别优化技术 (01-04) 中受益。使用一致性哈希（CH）作为基准，并逐步在系统上添加优化技术（例如，“+03”表示我们在系统上部署了 01、02 和 03）。通过多类型负载感知，可以在多个约束条

件下部署多个优化技术。

六. 创新与特色

6.1 Lake 的框架可以实现高效的多类型负载感知，用于网络内键值存储。

Lake 框架通过统计网络内键值存储的负载数据，可以感知不同类型的负载，并根据不同类型的负载选择不同的处理策略。例如，在高并发读写的情况下，Lake 可以采用一些高效的缓存策略来减少网络和存储的负担。在写入量较大的情况下，Lake 可以采用副本复制等策略来保证数据的可靠性和一致性。同时，Lake 框架可以对不同类型的负载进行统计和分析，通过识别和解决瓶颈问题，提高存储系统的性能。因此，Lake 的框架可以实现高效的多类型负载感知，为网络内键值存储系统提供了更加灵活和高效的处理方式。

6.2 Lake 消耗可接受的资源，并具有高准确性。

Lake 框架在实现负载感知时，可以达到高准确性的目标，这是因为它采用了多种负载识别技术和算法，如负载分布统计、聚类分析等方法，可以准确地识别出网络中各种类型的负载，并针对不同的负载类型采取相应的优化措施。同时，Lake 框架所消耗的资源是可接受的，它可以在较低的资源消耗下实现高准确性的负载感知，这也使得它在实际应用中更加具有可行性和可靠性。

6.3 在不同的工作负载分布下，Lake 具有高度的可扩展性和稳

健性。

Lake 框架的高度可扩展性和稳健性是指，该框架在不同的工作负载分布下仍然能够保持较高的性能表现，同时具有较好的稳定性和稳健性。Lake 框架基于负载统计数据进行工作负载的感知和调度，能够快速适应不同的负载分布。在负载分布发生变化时，Lake 框架能够自动调整负载感知和调度策略，以确保系统性能的高效和稳定。具体来说，Lake 框架通过动态调整资源分配和任务调度策略，以适应不同的负载分布。当系统中存在多种类型的负载时，Lake 框架可以根据负载类型的不同，调整资源的分配和任务的调度方式，从而实现不同类型负载的高效处理。此外，Lake 框架还支持动态伸缩，即根据负载情况自动调整系统的资源规模，以实现更好的性能表现。

6.4 基于负载统计数据，键值存储可以快速识别和解决瓶颈问题。

在 Lake 框架中，系统会实时监控不同负载类型的性能指标，并根据这些指标进行负载均衡。通过统计数据分析，Lake 可以帮助用户快速识别出系统中可能存在的瓶颈问题，并提供针对性的解决方案。例如，当某种类型的工作负载增加时，系统可能会出现性能下降或响应延迟等问题，这时候 Lake 会根据负载均衡策略，自动将该负载类型的任务分配到其他节点上，以避免单个节点过载而影响整个系统的性能。通过这种方式，Lake 可以保证系统的高可用性和可扩展性，并提高系统的整体性能。

6.5 Lake 框架允许用户为负载感知进行定制并自由调度工作负载。

Lake 框架允许用户根据具体的需求进行负载感知的定制，并能够自由调度工作负载。用户可以根据自己的需求，设置不同的负载类型和调度策略，以满足其特定的应用场景。例如，用户可以指定不同的负载类型，如读取、写入、删除等，以及每种负载类型的权重和优先级，Lake 框架会根据这些指定自动对工作负载进行调度。此外，用户还可以自定义调度策略，以更好地满足其特定的需求。这种灵活的定制性和自由度，使得 Lake 框架可以在不同的应用场景下发挥最佳的性能。

附录

CMSketch

```
template<typename DATA_TYPE,typename COUNT_TYPE>
class CMSketch{
public:
    CMSketch(uint32_t _MEMORY){
        LENGTH = _MEMORY / sizeof(COUNT_TYPE) / HASH_NUM;
        // 初始化哈希表
        sketch = new COUNT_TYPE* [HASH_NUM];
        for(uint32_t i = 0;i < HASH_NUM; ++i){
            sketch[i] = new COUNT_TYPE[LENGTH];
            memset(sketch[i], 0, sizeof(COUNT_TYPE) * LENGTH);
        }
    }
    ~CMSketch(){
```

```
        // 释放哈希表内存
        for(uint32_t i = 0;i < HASH_NUM;++i)
            delete [] sketch[i];
        delete [] sketch;
    }
    void Insert(const DATA_TYPE item) {
        // 将数据进行哈希，并在相应的位置增加计数器的值
        for(uint32_t i = 0; i < HASH_NUM; ++i) {
            uint32_t position = hash(item, i) % LENGTH;
            sketch[i][position] += 1;
        }
    }
    COUNT_TYPE Query(const DATA_TYPE item){
        COUNT_TYPE ret = 0x7fffffff;
        // 根据哈希值查询相应的计数器值，返回最小值
        for(uint32_t i = 0; i < HASH_NUM; ++i) {
            uint32_t position = hash(item, i) % LENGTH;
            ret = MIN(ret, sketch[i][position]);
        }
        return ret;
    }
private:
    uint32_t LENGTH;
    const uint32_t HASH_NUM = 3;
    COUNT_TYPE** sketch;
};
```

Priority queue

```
template<typename DATA_TYPE, typename COUNT_TYPE>
class PriorityQueue {
```

```
  public:
      PriorityQueue(uint32_t _TOP_K, uint32_t _MAX_ITEM, double
_ERR) :
          TOP_K(_TOP_K),       MAX_ITEM(_MAX_ITEM),       ERR(_ERR),
totalCount(0) {
          heap = new Item[TOP_K + 1];
          memset(heap, 0, sizeof(Item) * (TOP_K + 1));
      }
      ~PriorityQueue() {
          delete[] heap;
      }
      void Insert(const DATA_TYPE item, COUNT_TYPE count) {
          totalCount += count;
          // 如果 count 小于误差范围内的最小值，则忽略该 item
          COUNT_TYPE threshold = ERR * totalCount;
          if (count < threshold) return;
          // 如果堆还没有达到 TOP_K，则直接插入堆中
          if (heapSize < TOP_K) {
              heap[++heapSize] = { item, count };
              if (heapSize == TOP_K) MakeHeap();
              return;
          }
          // 如果 count 大于堆顶元素，则将堆顶元素替换成该 item
          if (count > heap[1].count) {
              heap[1] = { item, count };
              AdjustDown(1);
          }
      }
      uint32_t Size() const {
```

```
            return heapSize;
        }
        std::vector<std::pair<DATA_TYPE, COUNT_TYPE>> GetTopK() const {
            std::vector<std::pair<DATA_TYPE, COUNT_TYPE>> ret;
            // 将堆中的元素按照 count 从大到小排序
            std::sort(heap + 1, heap + heapSize + 1, [](const Item& a,
const Item& b) {
                return a.count > b.count;
            });
            // 将堆中的元素转成 vector 返回
            for (uint32_t i = 1; i <= heapSize; ++i) {
                ret.emplace_back(heap[i].item, heap[i].count);
            }
            return ret;
        }
    private:
        struct Item {
            DATA_TYPE item;
            COUNT_TYPE count;
        };
        uint32_t TOP_K;
        uint32_t MAX_ITEM;
        double ERR;
        uint64_t totalCount;
        uint32_t heapSize = 0;
        Item* heap = nullptr
        void MakeHeap() {
            for (uint32_t i = heapSize / 2; i >= 1; --i) {
                AdjustDown(i);
```

```
            }
        }
        void AdjustDown(uint32_t idx) {
            Item tmp = heap[idx];
            uint32_t i = idx;
            uint32_t j = i * 2;
            while (j <= heapSize) {
                if (j < heapSize && heap[j].count > heap[j + 1].count)
++j;
                if (heap[j].count < tmp.count) {
                    heap[i] = heap[j];
                    i = j;
                    j = i * 2;
                }
                else {
                    break;
                }
            }

            heap[i] = tmp;
        }
    };
```

Bloom filter

```
    #ifndef BLOOMFILTER_H
    #define BLOOMFILTER_H
    #include "Util.h"
    #include "MurmurHash3.h"
    template<typename DATA_TYPE>
    class BloomFilter {
```

```
public:
    BloomFilter(uint32_t _MEMORY, uint32_t _ITEM_NUM) :
MEMORY(_MEMORY), ITEM_NUM(_ITEM_NUM) {
        bits = new char[MEMORY];
        memset(bits, 0, MEMORY);
        hash_num = ceil(log(2) * MEMORY / ITEM_NUM);
    }
    ~BloomFilter() {
        delete[] bits;
    }
    void Insert(const DATA_TYPE item) {
        uint32_t* hash_value = new uint32_t[hash_num];
        MurmurHash3_x86_32(&item, sizeof(item), 0, hash_value);
        for (uint32_t i = 0; i < hash_num; ++i) {
            uint32_t pos = hash_value[i] % (MEMORY * 8);
            bits[pos >> 3] |= (1 << (pos & 7));
        }
        delete[] hash_value;
    }
    bool Query(const DATA_TYPE item) {
        uint32_t* hash_value = new uint32_t[hash_num];
        MurmurHash3_x86_32(&item, sizeof(item), 0, hash_value);
        for (uint32_t i = 0; i < hash_num; ++i) {
            uint32_t pos = hash_value[i] % (MEMORY * 8);
            if (!(bits[pos >> 3] & (1 << (pos & 7)))) {
                delete[] hash_value;
                return false;
            }
        }
```

```
        delete[] hash_value;
        return true;
    }
private:
    const uint32_t MEMORY;
    const uint32_t ITEM_NUM;
    uint32_t hash_num;
    char* bits;
};
#endif
```


[1] X. Jin, X. Li, H. Zhang, R. Soule, J. Lee, N. Foster, C. Kim, ´ and I. Stoica, “Netcache: Balancing key-value stores with fast in-network caching,” in Proc. of ACM SIGOPS SOSP, 2017.

[2] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, “Be fast, cheap and in control with switchkv,” in Proc. of NSDI’16, 2016.

[3] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. Ports, “Pegasus: Tolerating skewed workloads in distributed storage with in − network coherence directories,” in Proc. of OSDI’20, 2020.

[4] Barroso L, Marty M, Patterson D, et al. Attack of the killer microseconds[J]. Communications of the ACM, 2017, 60(4): 48-54.

[5] Dean, Jeffrey, and Luiz André Barroso. "The tail at scale." *Communications of the ACM* 56.2 (2013): 74-80.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, “Workload analysis of a large-scale key-value store,” in Proc. of SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems, 2012.

[7] J. Yang, Y. Yue, and K. Rashmi, “A large scale analysis of hundreds of in-memory cache clusters at twitter,” in Proc. of OSDI’20, 2020.

[8] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, “Workload analysis of a large-scale key-value store,” in Proc. of SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems, 2012.

[9] ——, “The cachelib caching engine: Design and experiences at scale,” in Proc. of OSDI’20, 2020, pp. 753–768

[10] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, “Characterizing, modeling, and generating workload spikes for stateful services,” in Proc. of ACM SoCC, 2010

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese et al., “P4: Programming protocol-independent

packet processors," ACM SIGCOMM CCR, 2014.
[12] Carter J B, Bennett J K, Zwaenepoel W. Implementation and performance of Munin[J]. ACM SIGOPS Operating Systems Review, 1991, 25(5): 152-164.
[13] J. Chen, L. Chen, S. Wang, G. Zhu, Y. Sun, H. Liu, and F. Li, "hotring: A hotspot-aware in-memory key-value store," in Proc. of FAST'20, 2020
[14] (2022) Barefoot tofino 2: Second-generation of world's fastest p4-programmable ethernet switch asics. [Online]. Available: https://www.barefootnetworks.com/products/brief-tofino-2/