面向海量电子凭据的分层可扩展存储架构

李凤华[1,2]，李丁焱[1,2]，金伟[1,2]，王竹[1,2]，郭云川[1]，耿魁[1]
（1. 中国科学院信息工程研究所，北京 100093；2. 中国科学院大学网络空间安全学院，北京 100049）

摘 要：电子商务等网络服务的兴起极大地促进了电子凭据业务的发展，传统数据存储方案无法满足海量电子凭据数据的快速访问需求。针对上述问题，提出了一种面向海量电子凭据的分层可扩展存储架构，结合 hash 取模算法和一致性 hash 算法实现快速的数据定位，设计了基于 hash 取模算法的横向扩展方案，减少了节点增删时迁移的数据量。此外，设计并实现了基于热数据的缓存方案和基于访问时延的负载均衡方案，进一步提升数据访问的速度。最后通过实验证明了所提架构与方案的有效性。

关键词：海量数据；电子凭据；数据定位；数据缓存；负载均衡
中图分类号：TP311
文献标识码：A
doi: 10.11959/j.issn.1000−436x.2019104

Hierarchical scalable storage architecture for massive electronic bill

LI Fenghua[1,2], LI Dingyan[1,2], JIN Wei[1,2], WANG Zhu[1,2], GUO Yunchuan[1], GENG Kui[1]
1. Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
2. School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China

Abstract: The rise of e-commerce and other network services have greatly promoted the development of electronic bill service, while traditional data storage schemes can no longer satisfy the rapid access requirements of massive electronic bill data. To solve these problems, a hierarchical scalable storage architecture for massive electronic bill was proposed, which combined hash modular algorithm and consistent hash algorithm, and supported fast data locating. A horizontal expansion scheme based on hash modular algorithm was designed to reduce the amount of data that needed to migrate when adding or deleting data nodes. Besides, a data caching scheme based on hot data and a load balancing scheme based on access delay were designed and implemented, further improving the speed of data access. Finally, the experiments prove the effectiveness of proposed architecture and schemes.

Key words: massive data, electronic bill, data locating, data caching, load balancing

1 引言

随着互联网在各个领域的不断深入，云计算、电子商务、电子支付、社交网络等新型服务模式得到了迅猛发展，各类网络业务的兴起极大地促进了电子凭据的发展，产生了海量的电子凭据数据。以电子发票服务体系为例，自 2016 年起，我国电子发票进入广泛推广期，根据智研咨询集团《2018—2024 年中国电子发票市场运营态势及前景预测报告》预计，到 2022 年我国电子发票开具总量高达 545.5 亿张，并保持年均超过 100%的高速增长。

海量的数据使单个节点难以存储，节点的访问请求无法被及时处理，且无法满足用户的快速访问需求，因此必须扩展系统的性能，加快节点访问请求的处理速度。传统的集中式存储方案更倾向于纵

收稿日期：2019–02–22；**修回日期**：2019–05–09
通信作者：耿魁，gengkui@iie.ac.cn
基金项目：国家重点研发计划基金资助项目（No.2017YFB0802702）；国家自然科学基金资助项目（No.61672515）
Foundation Items: The National Key Research and Development Program of China (No.2017YFB0802702), The National Natural Science Foundation of China (No.61672515)

向扩展，不断升级设备的硬件配置，但是这种方式带来的性能提升远不及数据快速增长造成的压力。海量的数据更适合采用分布式的存储方案[1-2]，易于水平扩展，更符合实际的使用需求，然而，目前，大数据存储系统在数据定位、数据缓存和负载均衡方面都存在显著挑战。

在数据定位方面，目前，常用的数据节点定位方式主要有以下3种。1) 基于文件目录进行定位[3]。将数据节点的元信息集中存储到中心服务器上，访问数据时先到中心服务器定位数据节点，然后访问相应的数据节点。但是当数据量和访问量很大时，数据定位所需的开销会比较大，中心服务器容易成为系统瓶颈。2) 基于一致性hash算法进行定位[4]。将数据映射到一个环状区间，在增删节点时，只有邻近的节点会受到影响，这避免了节点规模变动时数据迁移代价过高的问题。为了保证节点较少时数据依然能够均匀分布，亚马逊公司在Dymano系统中引入了虚拟节点的概念，增加了从虚拟节点到物理节点的映射。但是由于采用了完全去中心化的设计，要定位到数据所在的节点，需要的时间复杂度为 $O(n)$，当系统规模很大时，定位的开销将会很大。3) 基于索引进行定位[5]。通过跳表、位图等数据结构构建分层索引，访问数据时先通过索引快速定位数据所在的节点，避免了额外的查询开销，这种方法的缺点是实现比较复杂、索引的维护代价比较高。

在数据缓存方面，目前，使用较为广泛的大数据存储系统大多直接对磁盘进行读写，这为加快上层应用的访问速度，进一步提升了系统性能，且出现了基于缓存的分层式架构[6]，将部分数据放置到随机访问性能更好的固态硬盘或内存中。在分布式缓存中，Redis和Memcached这两款内存数据库应用的较为广泛[7]，它们使用最近最少使用（LRU, least recently used）算法作为默认的缓存替换策略。LRU算法是一种常见的内存页面置换算法，简单易用，但是该算法只考虑了“时间”因素，忽略了对“频率”因素的考虑，因此在缓存策略方面依然存在着很大的改进空间。

在负载均衡方面，文献[8]对当前分布式系统中的任务分配与负载均衡模型进行了分析，并从控制模型、资源优化、可靠性、协作性和网络结构这5个方面对当前的研究进行了讨论，提出了最小化响应时间、最小化任务完成时间、最大化任务吞吐量以及最大化任务可靠性这4个优化目标，指出了没有任何一种方案能在4个方面都达到最优效果，因此在实现时需要有所侧重。文献[9]对大数据系统中的任务调度框架进行了研究，从任务粒度、执行时间、调度时机、实现架构这4个方面进行了分类总结，指出了当前大数据系统中的调度技术主要聚焦于集群环境下的任务批处理，在动态资源供应、分布式与异构网络、维持稳定执行时间等方面还有很大的改进空间。

针对海量电子凭据数据存储时面临的问题，本文提出了一种面向海量电子凭据的分层可扩展存储架构，贡献如下。

1) 分层可扩展存储架构

为了快速地定位数据所在的节点，提出了基于hash取模算法与一致性hash算法的分层存储架构，该架构的数据定位的时间复杂度为 $O(1)$。同时针对hash 取模算法模数变化时数据迁移成本过高的问题提出了改进方案，增强了系统的可扩展性。

2) 基于热数据的缓存方案

为了进一步加速数据访问的过程，减少对下层数据节点的访问，设计了基于热数据的缓存方案，识别用户高频访问的数据并且缓存到中间层，当用户再次访问这些数据时，可以直接从缓存中返回结果，不需要再访问下层的数据节点。

3) 基于访问时延的负载均衡方案

为了避免节点负载不均衡对系统整体性能的影响，设计了基于访问时延的负载均衡方案，基于访问时延评估当前节点的负载状况，并依据评估结果调整下一时刻的节点负载。

2 相关工作

2.1 分布式存储架构

目前，主要的分布式存储架构大体可以分为主从架构和对等网络（P2P, peer to peer）架构两类。在主从架构中[10]，主节点负责管理整个系统，监视从节点的状态，对从节点的负载进行调度，这种架构设计和维护相对简单，但是主节点可能会成为系统瓶颈。在P2P架构中[11]，每个节点都是对等的，负责管理自己的区域，可以灵活地增删节点，并且不会对系统性能造成较大影响，但是系统设计复杂，不易实现。未来的研究趋势是将这2种架构结合，灵活地运用两者的优势。

2.2 数据缓存

对于访问频繁的数据，如果每次都从磁盘上读

取，势必会造成 I/O 瓶颈，使用缓存技术可以很好地解决这个问题。虽然内存的访问速度远大于磁盘的访问速度，但是由于价格比较昂贵，因此一般不会将磁盘的数据全部缓存到内存中，而是在达到一定容量后再进行替换。常用的缓存替换算法[12-13]包括 LRU 算法、最近最不常用（LFU, least frequently used）算法、自适应缓存替换（ARC, adaptive replacement cache）算法、最短最近使用（LIRS, low inter-reference recency set）算法等，这些算法以访问时间和访问频率等信息作为替换标准，但是适用的访问模式往往比较固定，例如，LRU 算法适用于高局部性的访问模式，LFU 算法适用于顺序或随机的访问模式，当访问模式变化时，缓存命中效果比较差。文献[14]指出了云环境下内存缓存与传统的 CPU 缓存区别很大，不能采用固定的行为模式，需要动态地对应用进行感知。文献[15]提出了一种针对 HBase 的索引热点数据缓存方案，不断统计数据的访问频率，利用指数平滑的思想识别热点数据，比 LRU 算法拥有更高的命中率。

2.3　负载均衡

负载均衡是分布式设计中的关键问题之一，在实际运行环境中，难以准确预测节点的任务量，可能会出现部分节点负载过重的情况，此时需要对节点的负载进行动态的调整，并且尽可能地减小调整过程的开销。实施负载均衡的第一步是对节点的负载进行估算，常用的方法包括如下 3 类。

1) 资源权重法[16]

资源权重法通过获取节点的 CPU 使用率、内存使用率、带宽使用率、磁盘 I/O 使用率等指标，为每一个指标赋予一个权重，综合评估节点的负载。但是这种方法通用性差，容易产生很大的偏差。根据多项指标评估节点的状态是一个典型的组合优化问题，这类问题更适合用智能算法[17]来解决。智能算法对于解决复杂的 NP 问题或者非线性问题有较好的效果，但是会消耗过多的资源和时间，对于实时性的访问并不适用。

2) 状态探测法[18]

状态探测法是周期性地获取各个节点的状态，例如是否空闲、任务队列长度等，根据节点状态进行选取，但是获取节点状态的过程可能会有较大的开销。

3) 负载预测法[19]

负载预测法通过记录节点的历史负载信息，依据数学模型以及当前负载状态预测下一阶段的负载。这种算法在负载比较稳定的情况下可以得到较好的预测结果，但是却忽略了对服务器性能差异的考虑。

针对上述问题，本文提出了一种面向海量电子凭据的分层可扩展存储架构，采用 hash 取模算法和一致性 hash 算法实现快速的数据定位，同时还增强了系统的可扩展性。此外，本文设计并实现了相应的数据缓存和负载均衡方案，进一步保障了系统整体的访问性能。

3　分层可扩展架构设计

3.1　分层可扩展架构概述

如图 1 所示，系统的整体架构包括 3 层，分别是应用网关层、hash 取模层和一致性 hash 层。

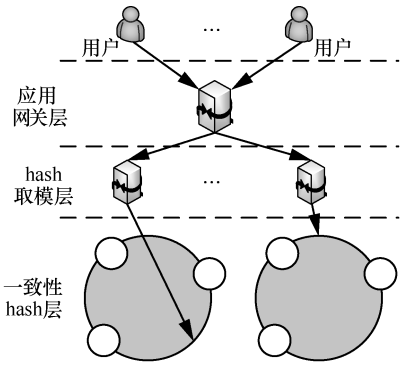


图 1　分层可扩展架构

应用网关层是分层可扩展架构的第一层，负责对外提供用户级接口，包括插入、查找等操作；提供基于 hash 取模算法的数据映射规则，将数据定位到 hash 取模层的节点上；可以在应用网关层指定基于 hash 取模算法的横向扩展规则，在增删节点时减少迁移的数据量。

hash 取模层是分层可扩展架构的第二层，负责管理下层的数据节点，转发来自应用网关层的操作请求，缓存访问频繁的数据。hash 取模层的节点提供基于一致性 hash 算法的数据映射规则，是一个中心化的节点，中心化的结构可以避免在分布式结构中的定位开销，加快数据定位的速度。在数据访问的过程中，hash 取模层对热数据进行识别，并将其缓存到内存中，当有重复的数据访问请求时可以直接返回结果。此外，还对下层数据节点进行负载及异常行为监测，根据监测结果进一步实现节点间动态的负载均衡。

一致性 hash 层位于架构的第三层，是数据节点所在层，负责实际的数据存储和备份。为了保证数

据的可用性，一致性 hash 层采用主从模式进行数据备份，而且采用读写分离的方式，即其中一个节点作为主节点，负责写操作，其余节点作为副节点，用于同步主节点的数据，负责读操作。在访问副节点时，需要一定的策略来保证节点间的负载均衡，防止部分节点过载导致的系统性能下降。

3.2 横向扩展

hash 取模算法在增删节点时会导致数据映射关系失效，需要迁移大量的数据。一种改进的方案[20]是在增删数据节点时，采用成倍增加或大幅减少的方式，可以在系统扩展的同时减少迁移的数据量。但是该方案的一个明显缺点是节点数必须成倍变化，不够灵活，难以满足实际的使用需求。为了在减少迁移的数据量的同时，能够更加灵活地增删节点，本文在上述方案的基础上分别对节点增加方案和节点删除方案进行了改进。

节点增加的过程可以分为节点倍增和节点枝剪 2 个步骤。假设当前有 $2^n(n>0)$ 个数据节点，编号为 $0\sim(2^n-1)$，数据与节点映射时对 2^n 取模。现在需要添加 $2^m(0\leqslant m<n)$ 个节点，编号为 $2^n\sim(2^n+2^m-1)$。对于新添加的每个节点，都需要获得 2 个集合，集合 set_1 中是添加节点后需要重定向到该节点的节点编号，集合 set_2 中是需要复制数据到该节点的节点编号。首先进行节点倍增，生成编号为 $0\sim(2^{n+1}-1)$ 的节点，但是由于编号为 $(2^n+2^m)\sim(2^{n+1}-1)$ 的节点不是真实存在的，因此需要进行节点枝剪，将不存在的节点重定向到编号为 $2^n\sim(2^n+2^m-1)$ 的节点上。集合 set_1 和集合 set_2 分别为

$$d=\frac{2^n}{2^m}=2^{n-m} \tag{1}$$

$$\text{set}_1=\{x+2^m i, i\in[0,d-1]\} \tag{2}$$

$$\text{set}_2=\{y-2^n, y\in\text{set}_1\} \tag{3}$$

其中，x 位于 $2^n\sim(2^n+2^m-1)$ 之间，集合 set_1 中的所有元素都位于 $2^n\sim(2^{n+1}-1)$ 之间，集合 set_2 中的所有元素都位于 $0\sim(2^n-1)$ 之间，set_2 中所有节点的数据需要复制到对应集合 set_1 中编号最小的节点上，数据与节点映射时对 2^{n+1} 取模，将 set_1 中所有的节点都重定向到集合中编号最小的节点上。如果需要把节点数目恢复成 2^n 个，可以将编号为 $2^n\sim(2^n+2^m-1)$ 中每个节点上的数据复制到对应 set_2 中每个编号对应的节点上，数据与节点映射时对 2^n 取模。

节点删除时，假设当前有 $2^p(p>0)$ 个数据节点，编号为 $0\sim(2^p-1)$，数据与节点映射时对 2^p 取模。现在需要删除 $2^q(0\leqslant q<p-1)$ 个节点，编号为 $(2^p-2^q)\sim(2^p-1)$。对于即将被删除的每个节点，都需要将其重定向到其他节点上。根据式(4)进行计算，其中，deleteNum 是要删除的节点编号，redirectNum 是重定向后的节点编号。

$$\text{redirectNum}=\text{deleteNum}-2^q \tag{4}$$

将被删除节点的数据复制到重定向后的节点上，数据与节点映射关系保持不变，仍然对 2^p 取模。如果需要把节点数再恢复成 2^p 个，根据式(4)找到重定向节点，然后将数据复制回本节点。数据与节点映射关系保持不变，仍然对 2^p 取模。

接下来，用示例进行说明。向已有的 2^2=4 个节点（$\text{node}_0\sim\text{node}_3$）中添加 2^1=2 个节点（node_4 和 node_5）的过程分别如图 2 和图 3 所示，初始时数据与节点映射对 2^2=4 取模。由式(1)可得 $d=2$，由式(2)可得 node_4和node_5 对应的 set_1 分别为 {4,6} 和 {5,7}，由式(3)可得 node_4和node_5 对应的 set_2 分别为 {0,2} 和 {1,3}。在修改数据与节点的映射关系前，将 node_0 和 node_2 中的数据复制到 node_4，将 node_1 和 node_3 的数据复制到 node_5。数据与节点映射时对 2^{2+1}=8 取模，由于 node_6 和 node_7 并不存在，因此将 node_6 中的数据重定向到 node_4 中，node_7 中的数据重定向到 node_5 中，这个过程迁移了一半的数据。图 3 中下划线标记的数据是在新的映射关系下失效的数据，需要进一步删除。

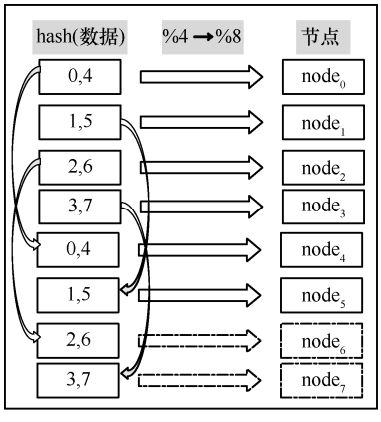


图 2 节点倍增

在已有的 2^3=8 个节点（$\text{node}_0\sim\text{node}_7$）中删除 2^1=2 个节点（node_6和node_7）的过程如图 4 所示，初始时数据与节点映射对 2^3=8 取模。根据

式 (4) 可得 $node_6$和$node_7$的重定向节点分别为$node_4$和$node_5$，在删除$node_6$和$node_7$之前将数据分别复制到$node_4$和$node_5$，数据和节点的映射关系保持不变。图 4 中下划线标记的数据是重定向后的数据。

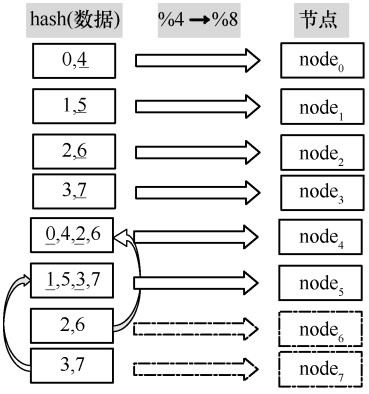


图 3　节点枝剪

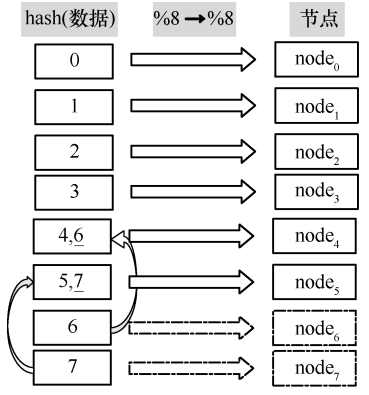


图 4　节点删除

从图 3 和图 4 中可以看出，无论是增加节点还是减少节点，最终都会打破原先数据均衡分布的局面。但需要注意的是，横向扩展方案针对的是 hash 取模层节点的变动，存储数据并不由该层节点负责，而是由下层的多个一致性 hash 层节点负责，所以对于横向扩展后数据量较多的 hash 取模节点，可以在下层为其部署更多的一致性 hash 节点，这样能够保证最终每个一致性 hash 节点上存储的数据依然比较均衡。

3.3　数据缓存

现实中的数据访问往往遵循“二八定律”，即 80%的业务访问集中在 20%的数据上，这 20%的数据被称为热数据。如何准确地识别热数据对于数据缓存来说十分重要，将访问频繁的热数据缓存到内存中，能够加快数据访问的速度、提升系统的性能。

在对数据的访问信息进行量化时，如果只考虑当前时间段内的访问信息，会将一部分用户随机访问的冷数据误当作热数据而进行缓存，为了避免这种情况的发生，需要同时结合历史访问信息和当前访问信息来识别热数据。

选择固定时间段内的数据访问次数作为数据热度的量化指标，采用式(5)进行计算。

$$\text{heat}_t = \alpha \text{count}_{\Delta t_1} + (1-\alpha)\text{heat}_{t-1} \tag{5}$$

其中，α 用于决定当前时间段内的访问信息和历史热度信息各自所占的比重，也称作衰变系数，满足 $0 \leqslant \alpha \leqslant 1$；$\text{count}_{\Delta t_1}$ 是时间段 Δt_1 内统计到的数据访问的次数；heat_{t-1} 是数据的历史热度信息；heat_t 是更新后的当前热度信息。α 值越大，表明当前时间段内访问信息所占比重越大，历史热度信息在迭代的过程中减小得越快；反之则是当前时间段内访问信息所占比重越小，历史热度信息在迭代过程中减小得越慢。

由于内存空间有限，本文无法将全部的数据都进行缓存，因此事先指定数据缓存空间的大小，在每次更新完成数据热度值后，按照热度值进行排序，如果排序后的数据量小于缓存空间，就把所有的数据进行缓存；如果排序后的数据量大于缓存空间，就从热度值最低的数据开始淘汰，直至剩余数据量小于缓存空间。数据热度更新和缓存替换算法如算法 1 所示。

算法 1　数据热度更新和缓存替换算法

输入　Δt_1 内的访问信息集合 $\text{count}_{\Delta t_1}$，历史热度信息集合 heat_{t-1}，缓存空间大小 cachesize

输出　当前热度信息集合 heat_t

1) 初始化结果集 heat_t 为空
2) for 遍历 $\text{count}_{\Delta t_1}$ 中的元组 $<\text{data},\text{count}_{\Delta t_1}>$ do
3)　　从 heat_{t-1} 中查找 data 的历史热度 heat_{t-1}
4)　　$a \leftarrow \alpha \text{count}_{\Delta t_1}$
5)　　$b \leftarrow (1-\alpha)\text{heat}_{t-1}$
6)　　$\text{heat}_t \leftarrow a+b$
7)　　将元组 $<\text{data},\text{heat}_t>$ 加入集合 heat_t
8) end for
9) 根据热度值对 heat_t 中的元组进行排序
10) if heat_t 的规模小于或等于 cachesize
11)　　返回 heat_t
12) else if heat_t 的规模大于 cachesize
13)　　while heat_t 的规模大于 cachesize
14)　　删除热度值最小的数据
15)　　end while

16) 返回 heat_t

17) end if

3.4 负载均衡

当进行读操作时，需要从多个副节点中选取负载最小的节点进行访问。访问请求响应时间的长短是衡量节点负载状况的一个重要标准，文献[21]指出单位时间内的平均访问时延与在该单位时间内处理的并行请求总数，能比较准确地反映节点的状况，因此本文也基于访问时延对节点的性能进行评估。

根据线性关系，可得

$$\text{Resp} = k\text{Req} + c \tag{6}$$

其中，Resp 表示请求的平均访问时延；k 表示直线的斜率，反映了随着请求数增加导致平均响应时间增长的快慢，是节点性能的评价指标；Req 表示请求的数目；c 表示其他因素导致的响应时间的增量。为了获得更加准确的估计值，根据多次采样的结果进行拟合，多点拟合直线常用的方法是最小二乘法，假设多次采样的结果分别为 $\text{Req}=[\text{req}_1,\cdots,\text{req}_n]$ 和 $\text{Resp}=[\text{resp}_1,\cdots,\text{resp}_n]$，可以根据式(7)~式(11)进行计算。

$$\overline{\text{req}} = \frac{1}{n}\sum_{i=1}^{n}\text{req}_i \tag{7}$$

$$\overline{\text{resp}} = \frac{1}{n}\sum_{i=1}^{n}\text{resp}_i \tag{8}$$

$$N=\sum_{i=1}^{n}(\text{req}_i - \overline{\text{req}})^2 \tag{9}$$

$$k' = \frac{1}{N}\sum_{i=1}^{n}(\text{req}_i - \overline{\text{req}})(\text{resp}_i - \overline{\text{resp}}) \tag{10}$$

$$c' = \overline{\text{resp}} - k'\overline{\text{req}} \tag{11}$$

其中，$\overline{\text{req}}$ 是访问请求数多次采样结果的平均值，$\overline{\text{resp}}$ 是请求访问时间多次采集结果的平均值。每隔 Δt_2 时间段进行一次估算，确保能实时地反映节点的性能。同时要对节点的负载状况进行估计，既要考虑到当前时间间隔内的请求数，也要考虑历史的请求数，同样采用基于指数平滑的方法进行评估，有

$$\text{Req}_t = \theta x_{\Delta t_2} + (1-\theta)\text{Req}_{t-1} \tag{12}$$

其中，Req_t 为节点的当前负载估算结果；Req_{t-1} 为节点的历史负载；$x_{\Delta t_2}$ 为当前时间间隔内的请求数；θ 为衰变系数，满足 $0 \leqslant \theta \leqslant 1$，$\theta$ 越大，表示历史信息的影响越小，反之历史信息的影响越大。

选取访问节点时，使用最新估算的节点性能指标与节点负载指标，根据式(6)进行计算并选取结果最小的节点，意味着该节点可以使请求的平均响应时间最小，提供更加快速的访问。

在系统横向扩展时，新的数据查询访问请求立即按照新的规则进行分发，但是被影响的 hash 取模节点上可能仍然存在一些旧的数据查询请求，这部分请求仍然会在失效数据被删除前得到服务，导致当前访问的副节点的负载偏高，在下个周期会选择其他副节点进行访问，同样也导致新选择的副节点负载偏高，最终残留的查询请求被多个副节点分摊，而且由于这部分请求数量有限，所以每个副节点负载偏高的幅度很低，处理过程很快，系统在短时间内就可恢复稳定。

4 实验分析

4.1 实验环境

基于本文架构在局域网中搭建实验环境，实验环境包括一个应用网关层网关 g_0，2 个 hash 取模层网关 g_1 和 g_2，4 个数据主节点 node_1~node_4，以及若干客户机模拟用户访问。根据存储单元的硬件配置，g_1 管理 node_1 和 node_4 这 2 个数据主节点，g_2 管理 node_2 和 node_3 这 2 个数据主节点。本文中副节点的配置与各自主节点相同，其中，node_1 的 2 个副节点分别为 node_{11} 和 node_{12}，node_2 的 2 个副节点分别为 node_{21} 和 node_{22}，node_3 的 2 个副节点分别为 node_{31} 和 node_{32}，node_4 的 2 个副节点分别为 node_{41} 和 node_{42}，各个节点的配置如表 1 所示。

表 1 各个节点的配置

设备	CPU	内存	硬盘	操作系统	数据库
g_0	8 核，i7 3.4 GHz	8 GB	500 GB，7200 rpm	Centos7，64 位	—
g_1	8 核，i7 3.4 GHz	8 GB	500 GB，7 200 rpm	Centos7，64 位	—
g_2	8 核，i7 3.4 GHz	8 GB	500 GB，7 200 rpm	Centos7，64 位	—
node_1	4 核，i7 3.4 GHz	4 GB	1 TB，7 200 rpm	Centos7，64 位	MySQL 5.6，64 位
node_2	4 核，i7 3.4 GHz	8 GB	1 TB，7 200 rpm	Centos7，64 位	MySQL 5.6，64 位
node_3	4 核，i7 3.4 GHz	4 GB	1 TB，7 200 rpm	Centos7，64 位	MySQL 5.6，64 位
node_4	4 核，i7 3.4 GHz	8 GB	1 TB，7 200 rpm	Centos7，64 位	MySQL 5.6，64 位

4.2　效果分析

用户对数据的访问往往遵循“二八法则”，这满足 Zipf 分布的典型特征[22]，因此本文在实验中对数据节点发起 Zipf 分布的访问请求。

1) 存储均衡分析

在不同的数据规模下，将数据存储到 2 000 个节点上，测试数据分布的均衡状况。计算每个节点的数据偏差 δ 为

$$\delta=\frac{x-\overline{x}}{\overline{x}} \tag{13}$$

其中，x 表示节点上的真实数据量，$\overline{x}$ 表示理想情况下节点的数据量。实验结果如图 5 所示，实验数据表明，在不同数据规模下，大部分节点的数据偏差都在 8%以内，而且随着数据规模的不断增大，节点的数据偏差依然能保持稳定。这说明即使数据规模很大，本文所提方案也能有效地将数据均衡地存储在各个节点。

2) 访问均衡分析

节点在处理访问请求的过程中需要耗费一定的资源，包括 CPU 资源、内存资源和磁盘 I/O 资源，但是对不同资源的消耗程度并不相同，因此本文为上述 3 种资源赋予不同的权重来更客观地综合评估节点的负载。采用式(14)来对每个节点的负载状况进行量化，其中，C 代表 CPU 使用率；M 代表内存占用率；D 代表磁盘 I/O 占用率；α、β、γ 分别为这 3 种资源的权重值，满足式(15)。

$$L=\alpha C+\beta M+\gamma D \tag{14}$$

$$\alpha+\beta+\gamma=1 \tag{15}$$

实验发现，在访问请求处理的过程中，CPU 资源对于节点负载的影响程度要高于内存资源和磁盘 I/O 资源，内存资源和磁盘 I/O 资源对于节点负载的影响程度相近，所以为 CPU 资源赋予较高的权重值，为内存资源和磁盘 I/O 资源赋予相同的权重值，经过多次实验，最终确定 α、β、γ 的值分别为 0.4、0.3、0.3。在客户端持续访问的 3 h 内，每隔 0.5 h 对节点的负载率 L 进行一次测算，实验结果如图 6 和图 7 所示。从图 6 和图 7 可以看出，随着时间的变化，单个节点的实际负载一直在波动，但是多个节点之间负载的波动趋势大致相同，负载率也比较接近，说明了本文所提负载均衡方案的有效性。

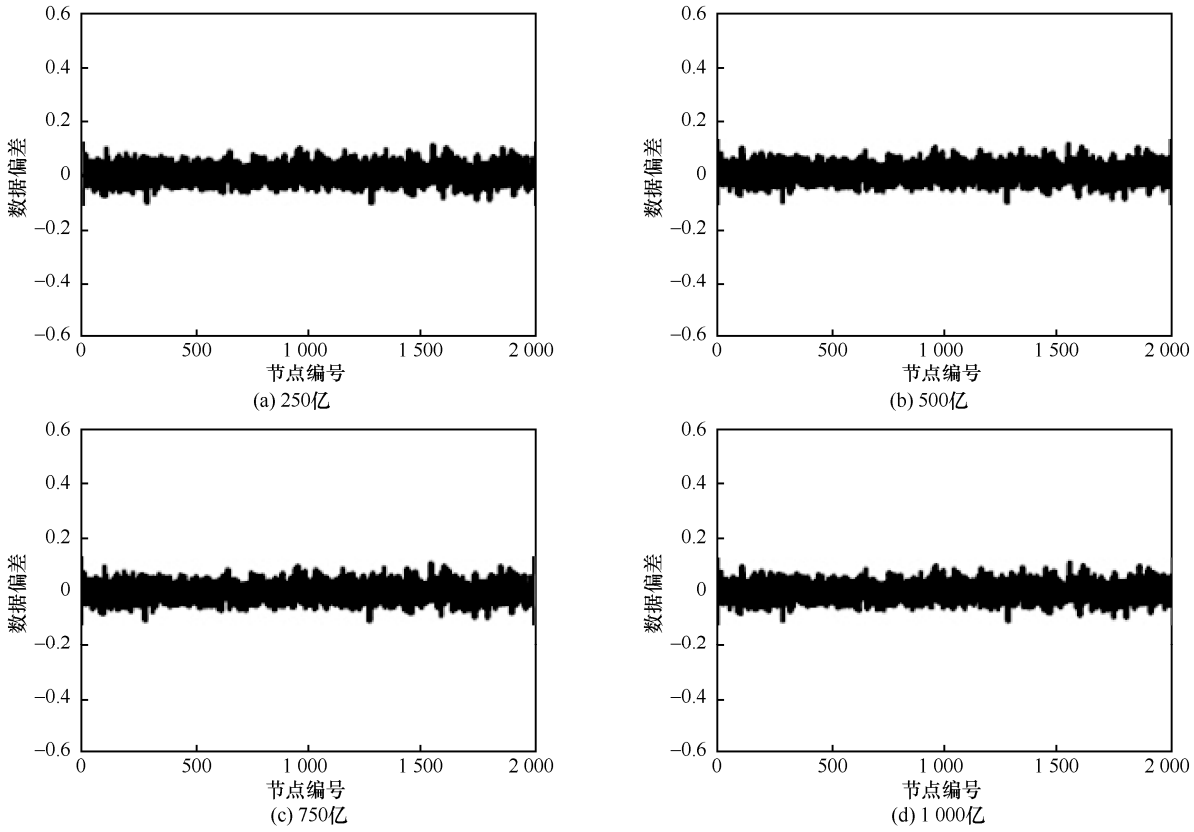


图 5　不同规模下的数据分布

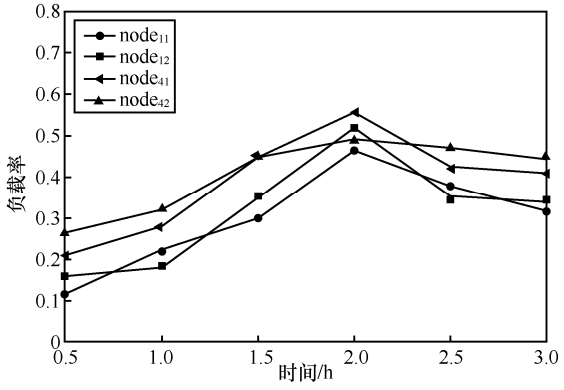


图 6　网关 g_1 下各个副节点负载

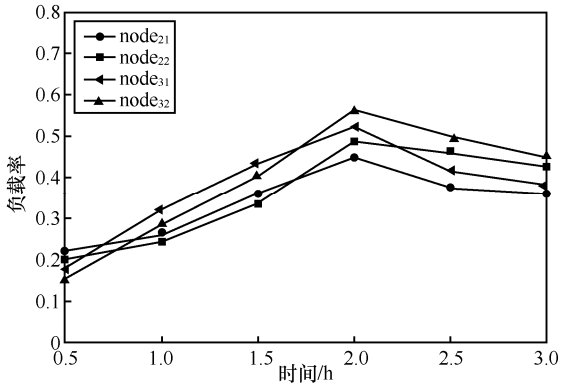


图 7　网关 g_2 下各个副节点负载

3) 查询效果分析

通过客户端对数据节点发起符合 Zipf 分布的数据访问请求，测试数据缓存对查询效果的影响。查询效果可以由平均访问时延体现，缓存大小可以用缓存数据量占数据总量的百分比表示，在本文架构下对平均访问时延进行测试。

图 8 对比了 LRU 算法、LIRS 算法、ARC 算法和本文算法的缓存命中率。实验数据表明，随着缓存规模的增大，所有算法的缓存命中率都在提升，但是本文算法的缓存命中效果总是优于其他 3 种算法。

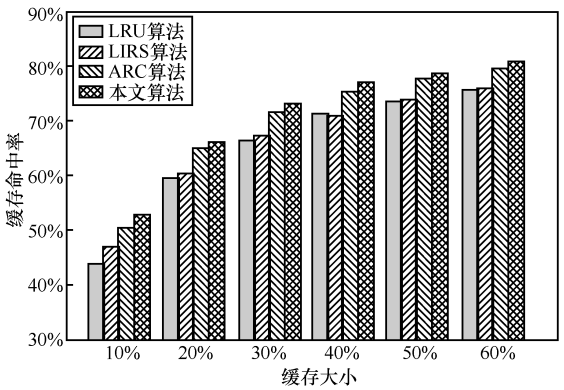


图 8　缓存命中率对比

图 9 给出了平均访问时延随缓存大小的变化情况。从图 9 可以看出，数据访问延时随缓存增加而降低。访问时延由计算时延、网络时延和查询时延这三部分组成，其中，计算时延包括 2 次 hash 计算，时间复杂度为 $O(1)$，经测试平均耗时为 5 ms；网络时延经测试一般不会超过 50 ms；当数据规模为 1 000 亿条、节点规模为 2 000 个时，平均每个节点的数据量为 5 000 万条，按照最差情况下会有 8%左右的偏差，本文在数据节点上存储 5 400 万条数据进行实验，在无缓存情况下，经测试平均查询时延约为 165 ms。由上述分析可知，即使在无缓存情况下，数据平均访问时延约为 220 ms，能满足实际需求。

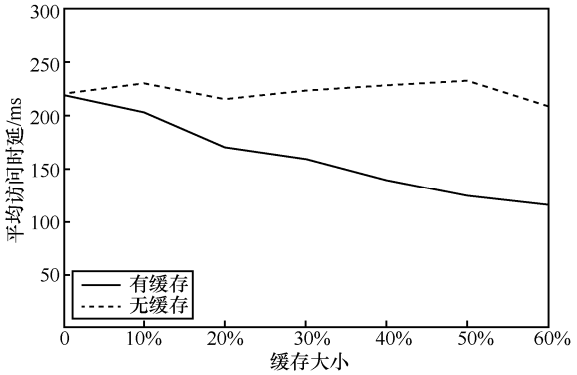


图 9　平均访问时延对比

5　结束语

本文针对海量电子凭据数据的存储与快速访问需求带来的挑战，从横向扩展、数据缓存和负载均衡三方面提出了改进方案，其中，横向扩展方案降低了数据迁移的成本，数据缓存方案对热数据的访问进行了优化，负载均衡方案可以将访问请求均匀地分布在各个节点上，并结合上述改进方案设计了一种分层可扩展存储架构，能够显著加快数据访问的过程。未来的工作还包括对分层可扩展架构中缓存方案和负载均衡方案的进一步优化。本文所提方案已应用于国家重点研发计划“安全电子凭据服务及其监管关键技术”项目中，能够满足千亿级数据毫秒量级查询响应的应用需求。

参考文献：

[1] MAKRIS A, TSERPES K, ANDRONIKOU V, et al. A classification of NoSQL data stores based on key design characteristics[J]. Procedia Computer Science, 2016, 97: 94-103.

[2] CORBELLINI A, MATEOS C, ZUNINO A, et al. Persisting big data: the NoSQL landscape[J]. Information Systems, 2017, 63: 1-23.

[3] SIDDIQA A, KARIM A, GANI A. Big data storage technologies: a survey[J]. Frontiers of Information Technology and Electronic Engineering, 2017, 18(8): 1040-1070.

[4] DECANDIA G, HASTORUN D, JAMPANI M, et al. Dynamo: Amazon's highly available key-value store[J]. ACM SIGOPS Operating Systems Review, 2007, 41(6): 205-220.

[5] 马友忠, 孟小峰. 云数据管理索引技术研究[J]. 软件学报, 2015, 26(1): 145-166.
MA Y Z, MENG X F. Research on indexing for cloud data management[J]. Journal of Software, 2015, 26(1): 145-166.

[6] VICTOR Z, DIVYAKANT A, AMR E A. Caching at the web scale[J]. Proceedings of the VLDB Endowment, 2017(10): 2002-2005.

[7] ASAF C, ASSAF E, MOHAMMAD A, et al. Cliffhanger: scaling performance cliffs in web memory caches[C]//13th USENIX Symposium on Networked Systems Design and Implementation. USENIX, 2016: 379-392.

[8] JIANG Y. A survey of task allocation and load balancing in distributed systems[J]. IEEE Transactions on Parallel and Distributed Systems, 2016, 27(2): 585-599.

[9] LIU J, PACITTI E, VALDURIEZ P. A survey of scheduling frameworks in big data systems[J]. International Journal of Cloud Computing, 2018(7): 103-128.

[10] AKRAM E, LARBI H, ABDERRAHIM M. The main characteristics of five distributed file systems required for big data: a comparative study[J]. Advances in Science, Technology and Engineering Systems Journal, 2017, 2(4): 78-91.

[11] 邹立达, 李庆忠, 孔兰菊, 等. 基于 Chord 的多租户索引机制研究[J]. 计算机学报, 2016, 39(2): 270-285.
ZOU L D, LI Q Z, KONG L J, et al. Indexing mechanism of multi-tenant database based on Chord[J]. Chinese Journal of Computers, 2016, 39(2): 270-285.

[12] QAISAR J, AYESHA Z, MUHAMMAD A, et al. Cache memory: an analysis on replacement algorithms and optimization techniques[J]. Mehran University Research Journal of Engineering and Technology, 2017, 36(4): 831-840.

[13] ZAIDENBERG N, GAVISH L, MEIR Y. New caching algorithms performance evaluation[C]//International Symposium on Performance Evaluation of Computer and Telecommunication Systems. 2015: 1-7.

[14] CIDON A, EISENMAN A, ALIZADEH M, et al. Dynacache: dynamic cloud caching[C]//USENIX Conference on Hot Topics in Cloud Computing. USENIX, 2015.

[15] 葛微, 罗圣美, 周文辉, 等. HiBase:一种基于分层式索引的高效 HBase 查询技术与系统[J]. 计算机学报, 2016(1): 140-153.
GE W, LUO S M, ZHOU W H, et al. HiBase: a hierarchical indexing mechanism and system for efficient HBase query[J]. Chinese Journal of Computers, 2016(1): 140-153.

[16] RATHORE N. Performance of hybrid load balancing algorithm in distributed web server system[J]. Wireless Personal Communications, 2018(101): 1233-1246.

[17] REN G, JUEBO W. Dynamic load balancing strategy for cloud computing with ant colony optimization[J]. Future Internet, 2015, 7(4): 465-483.

[18] YANG Z X, ZHANG S Y, JI X Y, et al. Research on cloud service quality control implementation based on improved load balance algorithm[J]. Journal of Computational Methods in Sciences and Engineering, 2018, 18(3): 793-800.

[19] HERBST N, AMIN A, ANDRZEJAK A, et al. Online workload forecasting[C]//Self-Aware Computing Systems. 2017: 529-553.

[20] 邢屹. 大规模键值分布式存储系统的设计与实现[D]. 成都: 电子科技大学, 2013.
XING Y. The design and implementation of a large-scale key-value distributed storage system[D]. Chengdu: University of Electronic Science and Technology of China, 2013.

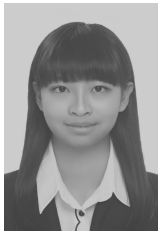
[21] 余楚玉, 温武少, 肖扬, 等. 一种自适应文件系统元数据服务负载均衡策略[J]. 软件学报, 2017, 28(8): 1952-1967.
SHE C Y, WEN W S, XIAO Y, et al. Adaptive load balancing strategy for file-system metadata service[J]. Journal of Software, 2017, 28(8): 1952-1967.

[22] JOANNA B, MACIEJ D. Comparing load-balancing algorithms for mapreduce under zipfian data skews[J]. Parallel Computing, 2018, 72: 14-28.

[作者简介]

李凤华（1966– ），男，湖北浠水人，博士，中国科学院信息工程研究所研究员、博士生导师，主要研究方向为网络与系统安全、信息保护、隐私计算。

李丁焱（1993– ），男，山西临汾人，中国科学院信息工程研究所硕士生，主要研究方向为网络安全防护与管控。



金伟（1994– ），女，北京人，中国科学院信息工程研究所博士生，主要研究方向为访问控制。

王竹（1972– ），女，山西太原人，博士，中国科学院信息工程研究所高级工程师，主要研究方向为信息安全、安全协议、人工智能。

郭云川（1977– ），男，四川营山人，博士，中国科学院信息工程研究所副研究员，主要研究方向为访问控制、形式化方法。

耿魁（1989– ），男，湖北红安人，博士，中国科学院信息工程研究所助理研究员，主要研究方向为网络安全。