

基于并发跳表的云数据处理双层索引架构研究

周 维 路 劲 周可人 王世普 姚绍文
(云南大学软件学院 昆明 650091)
(zwei@ynu.edu.cn)

Concurrent Skiplist Based Double-Layer Index Framework for Cloud Data Processing

Zhou Wei, Lu Jin, Zhou Keren, Wang Shipu, and Yao Shaowen
(School of Software, Yunnan University, Kunming 650091)

Abstract Cloud data processing plays an essential infrastructure in cloud systems. Without efficient structures, cloud systems cannot support the necessary high throughput and provide services for millions of users. However, most existing cloud storage systems generally adopt a distributed Hash table (DHT) approach to index data, which lacks to support range-query and dynamic real-time character. It is necessary to generate a scalable, dynamical and multi-query functional index structure in cloud environment. Based on the summary and analysis of the double-layer index systems for cloud storage, this paper provides a novel concurrent skiplist based double-layer index (referred as CSD-index) for cloud data processing. Two-layer architecture, which can breakthrough single machine memory and hard drive limitation, is used to extend indexing scope. Online migration algorithm of skiplist’s nodes between local servers is used to make dynamic load-balancing. The details of the design and the implement of the concurrent skiplist are discussed in this paper. Optimistic concurrency control (OCC) technique is introduced to enhance the concurrency. Through concurrent skiplist CSD-index improves the load bearing capacity of the global index and enhances the overall throughput of the index. Experimental results show the efficiency of the concurrent skiplist based double-layer index and it has viability as an alternative approach for cloud-suitable data structures.

Key words cloud computing; double-layer index; concurrent skiplist; range query; optimistic concurrency control

摘 要 云数据处理在云计算基础设施中占有极其关键的地位。然而,当前的云存储系统绝大部分都采用基于分布式 Hash 的健-值对模式来组织数据,在范围查询方面支持不理想、且动态实时性差,有必要构建云环境下辅助动态索引。通过总结、分析云环境中辅助双层索引机制,提出一种基于并发跳表的云数据处理双层索引架构。该架构采用两层体系结构,突破单台机器内存和硬盘的限制,从而扩展系统整体的索引范围。通过动态分裂算法解决局部服务器中的热点问题,保证索引结构整体的负载均衡。通过并发跳表来提高全局索引的承载性能,改善了全局索引的并发性,提高整体索引的吞吐率。实验结果表明,基于并发跳表的云数据处理双层索引架构能够有效支持单键查询和范围查询,具有较强的可扩展性和并发性,是一种高效的云存储辅助索引。

关键词 云计算;双层索引;并发跳表;范围查询;乐观并发控制

中图法分类号 TP393

由于能够提供海量存储、可靠服务,近年来,云计算系统日益受到重视.现有的云计算基础设施主要包括亚马逊的 EC2 (amazon's elastic computing cloud)^[1]、IBM 的 Blue cloud^[2] 以及 Google 的 Bigtable^[3]、微软的 Azure^[4] 等.在这些基础设施中,由成千上万台互相连接在一起的计算机构成提供服务的“云”,大量的用户可以同时共享这块“云”,并根据自己的实际需求,对所需资源进行剪裁.

云数据处理系统在云计算基础设施中占有极其关键的地位,其性能对部署在云计算基础设施中的应用系统有很大的影响.如果没有一个高效的云数据处理系统,云计算就不能准确定位资源并及时为成百万的用户提供服务.作为云数据处理中的一个重要组成部分,当前的云存储系统绝大部分都采用分布式 Hash 表 (distributed Hash table, DHT) 的方式来构建数据索引,数据被组织成键-值 (key-value) 对的形式.因此,这类的云存储系统只支持关键字查找,并通过点查询 (point-query) 来访问数据.然而,实际的应用效果表明,这类基于 key-value 模型的系统还存在一些亟待提高的地方.例如,对于一个在线视频点播系统来说,用户们往往倾向于采用多于一个的键值来进行查询,或需要查询特定属性处于某一个数据范围之内的视频信息.为了满足上述的应用需求,当前的解决方案主要是通过运行一个后台批处理任务 (例如运行一个 Hadoop 的任务),来扫描整个数据集然后得到查询结果.然而,这类解决方案缺乏时效性,新存入的数据元组不能被及时地查询到,必须等到后台的批处理任务完成了完整的扫描,数据才会可查.上述分析表明,当前云存储系统在高维度查询和范围查询方面支持的都不是很理想、且时效性差,有必要构建云环境下辅助动态索引.

基于以上考虑,本文提出一种基于并发跳表的云数据处理双层索引架构 (concurrent skiplist based double-layer index, CSD-index),论文的主要贡献是:

1) 提出一种基于并发跳表的云数据处理双层索引架构,该架构能够有效支持单键查询和范围查询,可扩展性强,且动态实时性好,是一种有效的云存储辅助索引.

2) 对双层索引构造过程、范围查询,以及局部索引向全局索引发布元节点的机制进行了详细分析,并给出了分裂和合并算法.动态分裂算法解决局部服务器中的热点问题,保证了索引结构整体的负载均衡.

3) 相对于已经提出的其他云数据处理双层索引架构,本文首次提出在上层全局索引中引入并发跳表 (concurrent skiplist),给出了并发跳表相关的操作说明,并对其正确性进行了深入分析.并发跳表提高了全局索引的承载性能,改善了全局索引的并发性,提高整体索引的吞吐率.

1 相关工作

已有的云存储系统包括:Google 的谷歌文件系统^[5]和 Bigtable^[3]以及其开源实现 Hadoop^[6]等.这些系统被设计来支持大规模分布式数据密集型应用.亚马逊的 Dynamo^[7]是用来对亚马逊的商业平台提供核心服务.Dynamo 采用一致性 Hash 函数在计算机节点之间分配数据.Facebook 的 Cassandra^[8]综合采纳了 Dynamo 的分布式系统技术和 Google 的 Bigtable 数据模型,它能够提供基于列的数据模式.上述这些系统基本上都采用了一种键值对的模型来组织数据,能通过主键高效的获取数据,但是这一类解决方案在区间查询和高维度查询方面支持的都不是很理想.

近期的研究表明,索引技术能显著改善云存储系统的性能.按照索引构建方式的不同,可称为嵌入式索引模式 (embedded-index model) 和旁路索引模式 (bypass-index model).所谓嵌入式索引模式是指直接在云存储系统中构建索引.已有的研究工作包括文献[9-13].文献[10]提出了一种 trojan 索引机制来提高运行时效率.它将索引整合进 Hadoop 系统内部,为 mapreduce 作业的选取提供优化.文献[9]则在 Map-Reduce-Merge 过程中提出一种新的算法,建立了一种基于树的索引结构.嵌入式索引模式是一种紧耦合的模式,与云存储系统结合起来的性能提高快.然而,这种紧耦合的模式也带来一些困扰.例如:所有上述工作^[9-13]都需要修改 Hadoop 云存储系统的底层文件系统或调度系统.这使得建立这种索引本身具有难度,且可扩展性不强.一旦 Hadoop 发布新的版本 (如下一代 Hadoop, YARN),那么这些索引需要很多额外的工作才能进行升级.此外,由于索引本身的庞大,这类嵌入式模式并不总是在内存中维护一个完整索引,而是采取运行时建立的策略,针对不同的请求建立不同维度的索引.这会在少量请求访问时,造成较高的创建开销.由于索引无法预测用户的行为,当大量用户同时提交分布不同的请求时,嵌入式模式也将面临索引建立选取的问题.

旁路索引模式则是在云存储系统之外建立,并总在内存中维护一个完整的索引.它可以通过应用编程接口同云存储系统进行交互.自 2008 年以来,少量基于不同数据结构的云存储辅助旁路索引相继在 VLDB, SIGMOD 等重要会议和期刊上发表出来^[14-19].文献[14]首次提出了一种全局分布式 B 树算法来构建云中大规模数据集的辅助索引结构.通过在服务器集群中构建一棵全局的 B 树来索引所有资源,分布式事务处理被用来保持事物一致性.文献[15]提出了一种全球云索引结构 CG-index (cloud global index), 每一个计算机节点建立一个局部的 B+ 树索引,然后各个计算机节点被组织成一个结构化的 Overlay, 每一台计算机将自己的局部 B+ 树通过某种特定的策略抽出一部分发布到这个 Overlay 中.文献[16]阐述了一种称为 RT-CAN 的多维索引结构. RT-CAN 的底层基于一些局部的 R 树索引,会动态地选择一部分局部 R 树节点发布到全局索引中,而全局索引在逻辑上被组织成一个 CAN Overlay. 由于其松耦合机制,旁路索引模式没有嵌入式索引模式效率高.但是,旁路索引模式容易建立,维护开销小.

本文采用旁路索引模式.在研究时,考虑到现在服务器普遍采用多核架构,为了更好发挥多核硬件的优势,在上层全局索引设计时,我们采用并发跳表来提高全局索引的承载性能,改善了整个云索引结构的并发性,提高整体吞吐量.

2 基于并发跳表的双层索引架构

2.1 skiplist 简介

skiplist 是一种概率数据结构,其形式表现为有序的链表,其与普通链表最大的区别在于:各节点之间有一些附加的并行指针相连.每一个插入到 skiplist 的 key 首先会被插入到最底层的链表中,接着存储这个 key 的节点会以 1/2 的概率将自己发布到上层中. skiplist 的查询、插入和删除操作能够到达 $O(\log N)$ 的期望值(最坏情况下为 $O(N)$,退化为线性查找).图 1 给出了一个在 skiplist 中查找 17 的过程,图 1 中虚线为实际查询操作的处理路径.

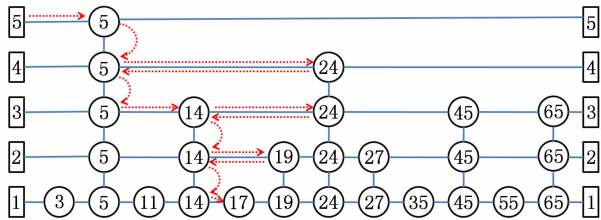


Fig. 1 Example of finding 17 in skiplist.
图 1 skiplist 中查找 17 的处理示意图

2.2 双层索引整体架构

图 2 是基于并发跳表的可扩展索引结构的两层架构示意图.整个索引结构分为上下两层,索引的数据具体存放在下层的索引中,而上层的索引则起到

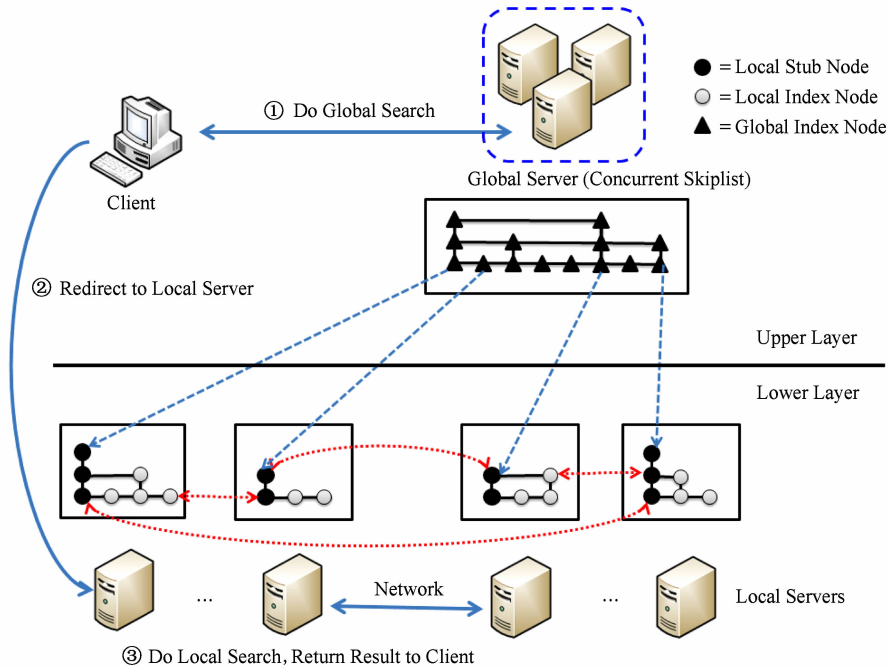


Fig. 2 The architecture of concurrent skiplist based double-layer index.
图 2 基于并发跳表的可扩展双层索引的架构示意图

一个定位和导向的作用. 在索引建立的时候, 首先会对待索引的数据集进行切分, 按照平均的原则, 分成包含等量数据的子集(划分的个数与下层的局部索引服务器相等). 然后, 划分好的数据子集与下层索引服务器一一对应, 在各下层索引服务器中以 skiplist 为基础建立局部索引. 在局部索引建立完成的基础上, 各局部索引会挑选一部分节点作为自己索引范围的“代表”, 并发布到上层的全局索引中(其中, 必须包括局部索引中的首节点, 即图例中的黑色圆圈部分). 发布时, 并不是直接将下层节点原封不动的拷贝给上层节点, 而是抽取这些被发布节点的元数据(包括索引的键、局部索引服务器 IP 地址、局部索引服务器磁盘物理块号), 仅将元数据发送到上层索引中, 以达到减轻上层索引的内存开销, 并存储更多节点的目的. 全局索引接收到下层各局部索引发布的元数据后, 通过 skiplist 的形式将这些元数据组织成一个全局的索引, 在逻辑上将下层各独立的局部索引关联起来, 维持了索引空间的整体一致性. 上层的全局索引作为整个索引的入口, 通过全局索引的定位, 查询操作转到下层某一个具体的局部索引上, 最终在下层找到需要的数据, 然后返回.

2.3 双层可扩展索引的构造过程

当前的互联网应用通常采用分布式存储系统来保存海量的业务数据, 而这些分布式存储系统一般以 DHT 的形式来提供访问的入口, 并不能很好地支持范围查询. 本文提出了一种构建于分布式存储系统上的二层索引, 整个索引构造过程如图 3 所示:

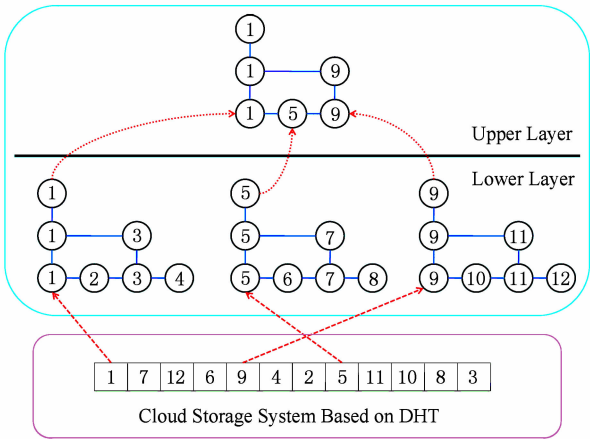


Fig. 3 The construction of CSD-index.
图 3 可扩展索引结构的构造示意图

图 3 所示的整个索引构造过程解释如下:

1) 该索引结构对下层的分布式存储系统的存储空间进行划分, 按照等量和有序的原则设定好各

局部索引管理的空间范围. 如图 3 所示, 分布式存储系统的存储空间为 1~12, DHT 由于其散列映射的特性, 其中各键的存储是无序的. 假定我们采用 3 个局部索引来保存分布式存储中的数据, 按照等量的原则, 每个局部索引应该存储 4 个数据. 因此, 从左往右, 1 号局部索引管理 1~4, 2 号局部索引管理 5~8, 3 号局部索引管理 9~12.

2) 按照第 1 步分配好的局部索引管理范围, 将分布式存储系统中的数据映射到对应的局部索引中. 当映射过程完成以后, 各局部索引内部将会是有序的, 同时各局部索引之间也是有序的.

3) 下层的各局部索引分别将其最高层的节点发布到上层的全局索引中, 全局索引通过下层发布过来的节点, 构造全局的 skiplist 索引, 然后将各局部索引关联起来, 构成完整的索引空间. 基于 skiplist 的特性, 其第 1 个节点必然是属于最高层的, 我们称之为桩节点. 在图 3 中, 从左往右, 1 号局部索引的桩节点为 1, 2 号局部索引的桩节点为 5, 3 号局部索引的桩节点为 9. 当这些最高层的桩节点发布到上层全局索引后, 该全局索引构成了一个包含 1, 5, 9 这 3 个节点的全局 skiplist.

4) 下层各局部索引逐步向下进行节点的迭代发布. 根据预估的发布后查询速度增加比和发布后全局索引内存占用的增长比, 来判断是否要继续向下发布局部索引的节点. 以图 3 为例, 在构造的过程中, 从左往右, 1 号局部索引发布了 1, 2 号局部索引发布了 5, 3 号局部索引发布了 9. 假设继续向下发布时, 索引结构整体能够取得正向收益, 则 1 号局部索引将会再发布 3, 2 号局部索引将会再发布 7, 3 号局部索引将会再发布 11. 因此, 当再向下发布后, 上层的全局索引中将会包含 1, 3, 5, 7, 9, 11 一共 6 个数据. 若预估查询速度增速比和全局索引内存占用增加比得到的收益是负向的, 则停止向下层发布.

2.4 局部索引向全局索引发布元节点

本结构中上层的全局索引则起到一个定位和导向的作用. 由于内存有限, 不可能把下层所有节点都发布到上层索引中, 下层局部索引向上层全局索引发布元节点进行关联时, 引入动态发布调整算法, 下层的局部索引选择性地发布节点. 图 4 描述局部索引向全局索引发布元节点的情况.

本索引结构采用下层向上层发布元节点, 在上层中构建全局的索引, 来维护索引结构的整体性. 在下层局部索引发布向上层发布节点时, 采用的是自顶向下的方式逐步增加发布的元节点数量. 首先, 每

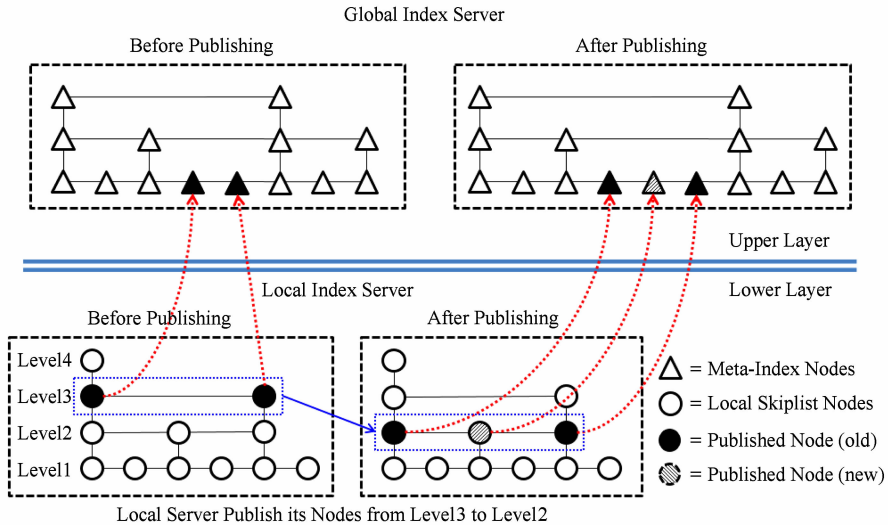


Fig. 4 Example of publishing local index to global index.
图 4 局部索引向全局索引发布元节点的示意图

一个局部索引将最高层的节点发布到全局索引中 (skiplist 的特性保证了最高层的节点必然包括第 1 个节点,即图 2 中标明的桩节点),接着各局部索引会根据预估的收益,判断是否要继续往下层发布. 预估策略的依据是局部索引发布之后,整体索引结构的查询速度变化率和全局服务器的内存变化率为基准. 图 4 给出了一个局部索引发布的节点从 Level3 向 Level2 延伸时,上层索引中节点变化的情况. 因为 skiplist 本身的特性,下一层的节点总是包含着上一层的节点,所以在向下扩展发布的时候,仅需要将之前没有包含的新节点的元数据发送给上层的全局索引(仅在全局索引中插入之前没有的节点).

对于 skiplist 来说,其索引的数据是存储于最底层的节点中,同时各节点以概率 p 向上升高,升高的部分作为查询的加速节点使用. 因此,在 skiplist 中自顶向下节点的数量将会以幂级的形式增加.

2.5 可扩展索引结构的查询处理

本文提出的可扩展索引结构是采用了 2 层体系结构,索引数据实际上存储于各局部索引中,而上层的全局索引则用来关联各局部索引,维护索引空间的整体一致性. 对本索引结构实施查询操作时,首先会以上层的全局索引作为查询的入口,通过查询全局索引,来确定哪一个局部索引实际包含着待查数据. 其次,查询处理将会转交给该局部索引,由该局部索引查询到确定的数据后,直接返回给查询请求的发起者. 图 5 描述了范围查询的过程.

图 5 给出了一个范围查询的具体处理流程(其中:索引结构的索引空间和实施例 1 中一致,为 1~

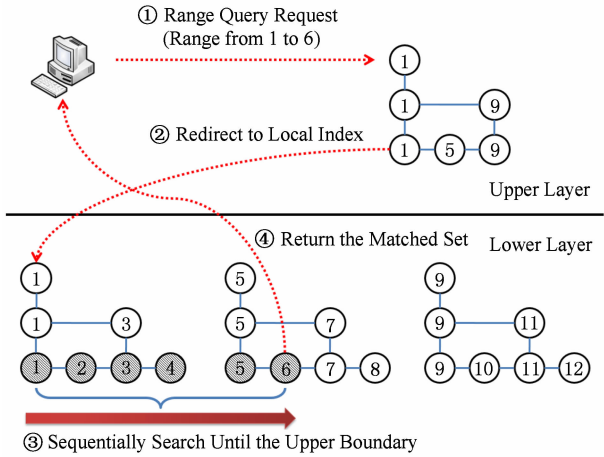


Fig. 5 Process of range query.
图 5 范围查询处理流程示意图

12,待查询的数据为 1~6). 详细解释如下:

1) 待查询的区间会被发送给上层的全局服务器,全局索引以区间的下界作为查询的入口键(即数据 1),在全局索引中进行检索.

2) 当上层的全局索引根据下界的键定位到具体局部索引后,查询处理会被转交给发布该键的下层局部索引. 在图 5 中,全局索引的数据 1 是由下层的左边第 1 个局部索引发布的,所以查询处理将会转移给该局部索引来继续处理.

3) 当局部索引接收到转交来的查询处理请求时,首先会根据待查询的区间,遍历自己的索引. 因为各局部索引内部是有序的,故只需不断向后遍历即可,直至满足查询区间的上界为止. 假如待查询的区间查过了一个局部索引的管辖范围,则需要将查询请求转交给该局部索引的后继兄弟. 由于各局部索引

之间也是互相有序的,故该转交能够保证查询的完整性和正确性.以图 5 为例,因为待查区间共有 6 个数据,而每个局部索引只管理 4 个数据,所以需要进行局部索引直接的递进转交.当 1 号局部索引查询到键 4 后,发现并没有满足查询区间的上界,故其将查询请求转交给右边的后续局部索引(我们称之为 2 号局部索引),2 号局部索引接受到查询请求后,继续在本空间中顺序向后检索,直至检索到数据 6,满足了待查区间的上界.至此,本范围查询处理结束,查询到的数据集直接从 2 号局部索引返回给查询的请求端.

范围查询是本索引结构的主要特征之一,而单键查询作为范围查询的一个特殊情况(即待查区间为 1 的情况),其处理过程和上述介绍的流程是一致的.主要区别在于,单键查询不涉及局部索引内部的遍历,也不涉及局部索引间的转交.对于单键查询的情况,上层全局索引转交给下层某一局部索引后,直接在该局部索引内部找到待查数据,即可返回.

2.6 分裂合并算法

本索引结构中,整个索引空间被划分为互不相交的子集,各子集分别由单独的局部索引来维护.随

着索引的动态插入及删除等调整操作,局部索引的大小有可能会差异,即有的局部索引逐渐变大,而有的局部索引反而会变小.局部索引的大小发生变化有可能会各局部索引之间的负载出现不均衡,因为相对较大的局部索引,被访问的概率会加大.因此,需要相应的动态分裂算法来解决局部索引中可能存在的热点问题.为了描述该分裂算法,首先给出 4 个变量定义:

1) S 为一个 skiplist,它由若干条有序的链表组成,所有的数据保存在 Level1,以一定概率选择出的部分节点保存在 Level2,再上层则以此类推.

2) $key(x)$ 表示保存在 S 中的元素 x 对应的键.

3) $level(x)$ 表示保存在 S 中的元素 x 对应的上层节点的高度, $level(S)$ 则表示所有保存在 S 中的元素的最高高度.

4) $wall(x, l)$ 表示保存在 S 中的元素 x 最右边第 1 个元素 y ,该元素 y 满足条件 $key(x) < key(y)$ 并且 $level(y) > l$.图 6 给出了一个以节点 5 为起始节点求 $wall(5, 3)$ 的例子,虚线框中的节点 24 即为所求.

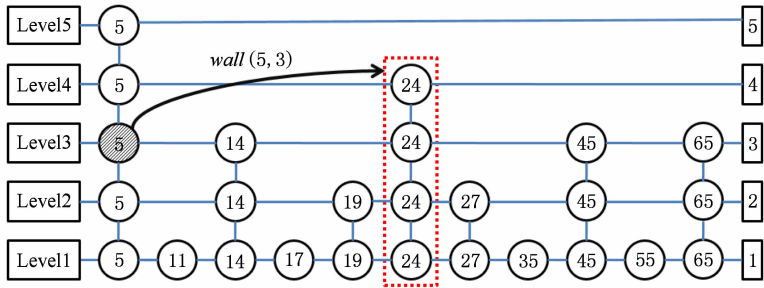


Fig. 6 Example of $wall(5, 3)$.

图 6 一个 $wall(5, 3)$ 的例子

局部索引的分裂算法主要依赖于 $wall(x, l)$ 的定义来完成.算法接受 3 个参数:1) S_1 为欲进行分裂处理的局部索引 skiplist; S_2 为接收 S_1 分裂出来的后半部分数据的局部索引 skiplist; 3) l 为决定分裂位置的参数.其算法具体流程为:

1) 选择 $wall(S_1, i)$, 其中 i 从 $level(S_1)$ 递减到 l , 逐步尝试直到找到第 1 个确定的 $wall(S_1, i)$ 为止.

2) 以 $wall(S_1, i)$ 为界,将前半部分的 skiplist 中指向 $wall(S_1, i)$ 节点的后继指针修改为 NULL, 即从 S_1 中删除后半部分的节点.以 $wall(S_1, i)$ 为界,将后半部分的节点发送给 S_2 , 逐个插入到 S_2 的 skiplist 中.

算法 1. Split.

输入:

- S_1 ——需要分裂成局部 skiplist 的服务器;
- S_2 ——接收 skiplist 分裂部分的服务器;
- L ——分裂范围参数.

- ① /* validate whether $wall(S_1, l)$ exists, if not adjust l */
- ② FOR $i := l$ DOWNT0 1 {
- ③ IF $wall(S_1, i)$ exists THEN BREAK; }
- ④ $l := i$;
- ⑤ /* record the links which need to be updated after splitting */
- ⑥ local $updateLink[1 \cdots S_1.StubLevel]$;
- ⑦ $x := S_1.list \rightarrow header$;

```

⑧ FOR  $i := S_1, StubLevel$  DOWNT0 1 {
⑨   WHILE  $x \rightarrow forward[i] < wall(x, l)$  {
⑩      $x := x \rightarrow forward[i]$ ;
⑪      $updateLink[i] := x$ ; }
⑫    $x := x \rightarrow forward[1]$ ;
    /* record the links */ }
⑬ /* migrate nodes and relink two servers */
⑭ IF  $x = wall(x, l)$  THEN {
⑮   FOR  $i := l + 1$  TO  $S_1, StubLevel$  {
⑯      $x \rightarrow forward[i] := updateLink[i]$ 
         $\rightarrow forward[i]$ ;

```

```

⑰    $updateLink[i] \rightarrow forward[i] := x$ ; }
⑱    $migrate(S_1, S_2, x)$ ; }

```

算法 1 将服务器 S_1 的局部 skiplist 分裂为 2 个部分,然后将后半部分迁移到服务器 S_2 . 参数 l 用来控制分裂的区间. 如果 l 接近桩节点的高度,在分裂操作实施以后,更多的节点会保留在服务器 S_1 . 反之,如果 l 离桩节点的高度越远,那么越多的节点会被迁移到服务器 S_2 . 图 7 给出了一个分裂的例子,配置参数为 $level(S) = 5, l = 4$. 服务器 S_1 的 skiplist 从节点 24(Level4 的第 1 个节点)分裂成 2 部分,后半部分的节点被迁移到了服务器 S_2 .

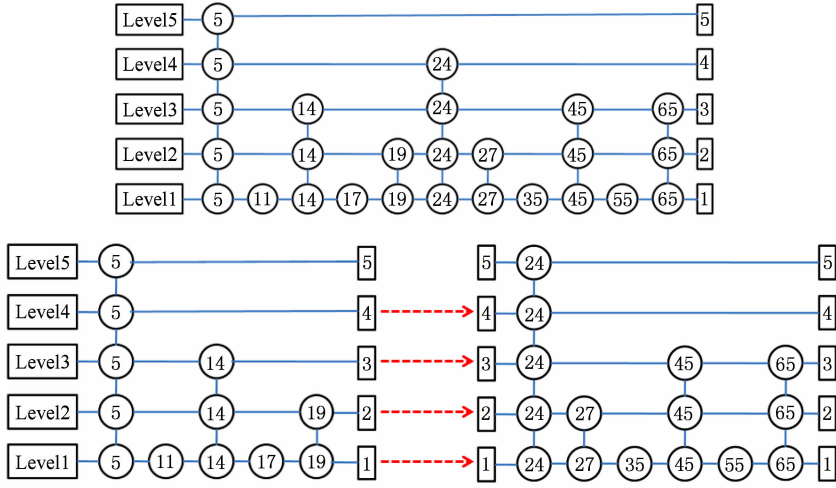


Fig. 7 Algorithm of splitting.

图 7 分裂迁移算法

分裂算法首先会坚持是否有一个在 Level1 的节点可以用来作为分裂的节点. 如果没有的话,将会逐步降低直到一个能够作为分裂节点的节点被成功找到(行②~④). 那些指向分裂节点的前半部分节点会被记录下来,以便能够在分裂之后成功的更新相关的链接指针(行⑥~⑫). 当所有的准备工作都完成之后,后半部分的节点将被迁移到一台新的服务器,而那些保留在原来服务器的前半部分节点将会更新他们的相关指针,将需要调整的指针指向迁移到的新服务器(行⑭~⑱).

合并算法和分裂算法类似,这里就不详细说明了.

3 基于并发跳表的上层索引设计

3.1 设计动机

本文提出的可扩展索引结构采用了双层索引结构,索引数据实际上存储于各局部索引中,而上层的

全局索引则用来关联各局部索引,维护索引空间的整体一致性. 在研究过程时,CSD-index 主要面临 3 个主要问题:

1) 对本索引结构实施查询操作时,首先会以上层的全局索引作为查询的入口,通过查询全局索引,来确定哪一个局部索引实际包含着待查数据. 上层的全局索引作为整个索引的入口,承受压力最大,因此如何提高上层全局索引的效率成为本研究一个重要思考的问题.

2) 由于现有服务器普遍采用多核架构,如何更好发挥多核硬件的优势,支持多核操作的数据结构成为重点考虑的方向.

3) 考虑到两层索引结构的动态性,下层索引会发布节点到上层全局索引中,全局索引存在插入、删除操作情况,但这类操作所占比例不会太高,更多的应该是大量查询操作.

基于上面的分析,在设计时,本研究在上层索引中采用并发跳表技术来实现.

3.2 并发跳表与乐观并发控制

传统跳表在多线程插入、删除时容易产生错误. 而并发跳表很好地解决了这个问题, 使得它很适合在多核、多线程环境中发挥作用. 并发跳表的每一层都是基于惰性同步的链表^[20], 其优点在于能够设置标志位来进行逻辑操作, 从而与节点删除这样的物理结构改变相分离, 保持了数据结构物理性质的完整. 然而, 其缺点在于插入操作和删除操作是阻塞的算法, 因此, 如果一个线程延迟, 会造成其他的线程也会延迟. 并发跳表的这个特点使得其主要适用于拥有大量查询操作, 少量插入、删除操作的场景. 这恰好与本文上层索引的应用场景很契合.

相对于平衡二叉查找树树型结构, 跳表不存在平衡调整的问题, 因此在多核、多线程环境下容易设计. 研究人员开展了大量工作, 如文献[21]中实现了一个 lock-free 的 ConcurrentSkiplistMap, 已经在 Java™ SE 6 中发布, 并且拥有很好的性能. 然而, 文献[21]中提出的实现过于复杂, 并且很难说明其正确性. 文献[22]中用基本的比较和交换原子操作 (compare and swap, CAS) 实现了 lock-free 的 linked-list, 并且在此基础上实现了 lock-free 的 skiplist, 但其缺点是对基本 skiplist 节点进行了过多的改造, 不能维持其原有 skiplist 的结构. 文献[23]提出一种分布式 skiplist, 通过一系列的算法来保障结构的并发性, 并做了理论分析. 文献[24]通过提出一种称为原子 3 元组 (atomic triples) 的概念便于并发分析推理. 文献[25-26]提出了基于消息驱动的分布式, 动态 skiplist, 通过利用新的算法能够在分布式机群环境下很好支持范围查找.

相对于前人提出的并发跳表, 本研究的特点在于采用了一种新的称为乐观并发控制的技术^[27]来设计跳表. 所谓乐观并发控制 (optimistic concurrency control, OCC) 是一种并发控制的方法. 它假设多用户并发的事务在处理时不会彼此互相影响, 各事务能够在不产生锁的情况下处理各自影响的那部分数据. 在提交数据更新之前, 每个事务会先检查在该事务读取数据后, 有没有其他事务又修改了该数据. 如果其他事务有更新的话, 正在提交的事务会进行回滚. 在数据冲突较少的环境中, 偶尔事务回滚的成本会低于读取数据时锁定数据的成本, 因此可以获得比其他并发控制方法更高的吞吐量.

在多核、多线程数据结构设计中, OCC 最早于 2003 年被引入软事务内存 (software transactional memory, STM), 被用来设计并发树^[28]和并发平衡

树^[29]. 近期, OCC 被用来对并发二叉树中查询操作进行优化^[30]. 受这些研究工作启发, 我们在并发跳表中采用 OCC 技术, 理论和实践证明, 乐观并发技术不但维护了 skiplist 的结构, 并且较容易说明其正确性.

由于一个完整的并发跳表的分析说明相对较长, 在这里只对一些关键部分进行阐述.

3.3 并发跳表 (concurrent skiplist) 的节点定义

```
结构 1. Node { int key;  
               int level;  
               Type data;  
               Node[] next;  
               Lock lock;  
               bool marked;  
               bool linked;  
            }
```

除去和普通跳表相同的部分, 并发跳表每个节点中还包含 2 个标志位和 1 个锁. *marked* 标志位用于标识该节点是否正在被删除. *linked* 标志位标识该节点是否完全插入, 即所有层次的指针域都更新完毕. 因为在这里我们采用了细粒度的锁, 所以每一个节点分别维护一个 *lock*. 另外, 还定义了 2 个哨兵节点 *head* 和 *tail*, 其 *key* 值分别为 *min_int* 和 *max_int*.

3.4 并发跳表基本操作

由于在对并发跳表进行正确性分析时必须针对跳表操作中的具体代码进行, 有必要先对并发跳表的几个主要操作 (定位、查询、插入、删除) 分别进行说明.

3.4.1 定位操作.

算法 2. *Locate*.

```
输入: k—查询的键值;  
输出:  
levelmax——键值为 k 的节点的最大层数;  
prevs——键值为 k 的节点的前节点数组;  
succ——键值为 k 的节点.  
① Node prev := head;  
② level := -1;  
③ FOR i := head.levelmax - 1 TO 0 {  
④   cur := prev → next[i];  
⑤   WHILE cur.key != tail.key &&  
        cur.key > k {  
⑥     prev := cur;  
⑦     cur := pred → next[i]; }  
⑧   IF level = -1 && k = cur.key {
```



```

⑨    level := i;
⑩    succ = cur; }
⑪    prevs[i] = prev; }
⑫    RETURN <found, prevs, succs>.

```

查询、插入和删除操作都依赖于定位操作. 和普通的跳表一样, 首先从哨兵节点 *head* 的最高层开始查找, 依次下降, 每一层查找到键值 *k* 所在位置或者哨兵节点 *tail* 停止. 如果找到 *k* 对应的节点, 那么更新 *i* 标志表示该节点的最高层. 不论该节点是否存在, 都需要记录其每层对应的前驱节点 *prevs*[*i*], 由于跳表的随机性, 每层的 *prevs*[*i*] 都可能不同. 如果在某一层找到了 *k* 对应的节点, 那么在位于该层以下的当前节点 *succ* 的值显然都是相同的 (即其自身节点), 所以只需更新一次. *Locate* 操作本身不用加锁, 它的作用是返回一个 3 元组, 表示定位的结果 *level*, 当前节点的位置 *succ*, 前驱节点的位置 *preds*.

3.4.2 插入操作

算法 3. Insert.

输入: *k*——插入的键值;

输出: 插入操作是否成功.

```

① highest := randomLevel();
② WHILE true {
③    <found, prevs, succ> := locate(k);
④    IF found != -1
⑤        RETURN false;
⑥    FOR level := 0 TO prevs.levelmax {
⑦        prevs[level].lock();
⑧    IF !(prevs[level].marked = false
        && succ.next[level].marked =
        false && prevs[level].next[level]
        = succs.next[level]) {
⑨        FOR i := level TO 0 {
⑩            preds[i].unlock();
⑪            CONTINUE; } } }
⑫    makeNode(node, k);
⑬    FOR level := 0 TO highest {
⑭        node.next[level] := succs.next[level];
⑮        prevs[level].next[level] := node; }
⑯    node.linked := true;
⑰    FOR level := highest TO 0 {
⑱        prevs[level].unlock(); } }
⑲    RETURN true.

```

在插入操作中:

1) 首先调用定位操作, 返回定位的结果, 如果

找到当前节点, 那么说明键值为 *k* 的节点已经存在, 不能插入. 否则, 进行接下来的插入操作.

2) 对前驱节点数组 *prevs* 自下向上加锁 (行⑥~⑪).

3) 验证返回的 *prevs* 和后继节点数组 *succ*. *next* 是否发生变化 (行⑧). 假如 *prevs* 和 *succ.next* 发生了变化, 那么先释放刚才的锁, 然后返回行⑪, 重新定位 *prevs* 和 *succ*. 如果 *prevs* 和 *succ* 都没有发生了变化, 进行行④.

4) 从底层开始向上进行插入操作, 然后置 *linked* 标志位为 true (行⑯), 表示插入节点已经完全链接, 最后释放所有的锁 (行⑰~⑱).

3.4.3 删除操作

算法 4. Remove.

输入: *k*——删除的键值;

输出: 删除操作是否成功.

```

① okDelete := false;
② WHILE true {
③    <found, prevs, succ> := locate(k);
④    IF okDelete || (found != -1 && succ.
        next[found].linked = true && succ.
        next[found].marked = false) {
⑤        IF okDelete {
⑥            succ.lock(); }
⑦        IF succ.marked := true {
⑧            succ.unlock();
⑨            RETURN false; }
⑩        okDelete := true;
⑪        succ.marked := true;
⑫        highest := -1;
⑬        FOR level = 0 TO found {
⑭            IF prevs[level].marked = false
                && prevs[level].next[level]
                = succ.next[level] {
⑮                prevs[level].lock();
⑯                highest := level; }
⑰        ELSE {
⑱            FOR i := level - 1 TO 0 {
⑲                prevs[i].unlock(); }
⑳            CONTINUE; } }
㉑        FOR level := found TO 0 {
㉒            prevs[level].next[level] =
                succ.next[level]; }
㉓        succ.unlock();

```

```

24  FOR level=found TO 0 {
25      prevs[level].unlock(); }
26  RETURN true. }
27 ELSE
28  RETURN false. }
```

删除操作将指定的节点删除,基本步骤和插入操作一样,首先定位节点(行③),然后判定当前节点的状态是否合理,即该节点完全链接,且没有正在被删除.如果该节点状态合理,那么对该节点上锁,然而有可能该节点已经被其他线程删除,此时返回 false(行⑨),否则,置节点 *marked* 标志位为 true(行⑪).接下来和插入操作一样,自下向上对前驱节点上锁,如果 *succ* 和 *prevs* 的状态发生改变,那么释放之前的锁,然后返回行③,重新定位节点.最后,从行⑱~⑳进行节点的物理删除,然后释放所有锁,返回 true.

3.4.4 查询操作

查询操作先通过定位查找节点的位置,然后返回查询结果,以及相应的前驱节点和后继节点.这里没有采用任何的锁机制和同步机制,因此,查询操作是无等待的.如果没有找到相应节点,当前节点正在被删除,或当前节点没有完全连接,那么查询失败.如果找到相应节点,并且该节点没有正在被删除且完全链接,那么是一次成功的查询.

算法 5. Query.

输入:*k*——查询的键值;

输出:键值对应的节点是否存在.

```

① <found, prevs, succ>=locate(k);
② IF found!=−1 && succ.linked &&
    succs.marked=false {
③  RETURN true. }
④ RETURN false.
```

3.5 并发跳表正确性分析

并发数据结构的正确性分析一般分为 2 方面:安全性(safety)和活性(liveness).其中,对安全性的一个重要评价是可线性化,其基本思想是每一个并发的经历(history)都等价于一个顺序的经历.通常,用来说明并发数据结构的可线性化性质的方法就是指出该算法的可线性化点(linearizability point),然后指明在该线性化点的并发操作是如何映射为不同线程的等价顺序操作.活性指的是算法是否会返回一个期待的结果,具体的说明如无死锁、无饥渴、无锁、无等待等.如果保证算法调用不会因为互相等待而无法获取资源,那么称其为无死锁.如果保证算法

调用最终都能取得请求的资源,那么称其为无饥渴.如果保证算法某次调用能在有限的步骤内完成,那么这个方法是无锁的.如果算法每次调用总是在有限步骤内完成,那么这个方法是无等待的.

基于上述并发数据结构的正确性分析的要求,我们对并发跳表进行了安全性和活性的正确性分析.最后的结果表明:1)并发跳表中主要操作(查询、插入、删除)算法都是可线性化的,符合安全性要求;2)并发跳表中查询操作是无等待的,插入、删除操作是无死锁的,满足活性要求.

3.5.1 并发跳表可线性化分析

为了进行并发跳表的可线性化分析,首先,我们需要指出并发跳表数据结构的不变性质(invariants),这些性质在其被创建的时候就成立,并且任何操作都不能改变这些性质.并发跳表具有如下 3 条不变性质:

- 1)哨兵节点不能改变.
- 2)每一层的节点都按照关键字排序,并且不会出现重复的关键字.
- 3)下层节点的节点数一定不少于上层节点.

下面通过 5 条引理来证明无论通过多少次查询,插入,和删除等操作,并发跳表描述的抽象映射集合仍将满足以上 3 个性质.

- 引理 1. 每一次 locate 操作都满足不变性质.
证明. 假设调用 locate 操作前,并发跳表满足不变性质,因为 locate 操作本身不对其做任何改变,所以每一次 locate 操作都满足不变性质. 证毕.
- 引理 2. 每一次 query 操作都满足不变性质.
证明. 假设调用 query 操作前,并发跳表满足不变性质,由引理 1 可知,每一次 locate 操作都满足不变性质,又因为 query 操作不对并发跳表做任何改变,所以引理成立. 证毕.

- 引理 3. 每一次 insert 操作都满足不变性质.
证明. 假设调用 insert 操作前,并发跳表满足不变性质,首先,由引理 1 可知,locate 操作满足不变性质.之后可能存在以下 3 种情况:
 - 1)找到重复的键值,返回 false,在这种情况下,显然没有对并发跳表做任何改变.
 - 2)对前节点的上锁操作失败,那么首先释放所有的锁,然后重试,没有对并发跳表做任何改变.
 - 3)对前节点的上锁操作成功,那么插入键值为 *k* 的节点,插入以后并发跳表仍然满足不变性质.综上所述,引理成立. 证毕.

引理 4. 每一次 delete 操作都满足不变性质.

证明. 假设调用 delete 操作前, 并发跳表满足不变性质, 首先, 由引理 1 可知, locate 操作满足不变性质. 之后可能存在以下 4 种情况:

- 1) 没有找到对应键值的节点, 或者相应节点正在被插入或删除, 那么返回 false, 在这种情况下, 显然没有对并发跳表做任何改变.
- 2) 对当前节点的上锁操作失败, 即当前节点已经被标记删除, 那么首先释放当前节点的锁, 返回 false, 没有对并发跳表做任何改变.
- 3) 对前节点的上锁操作失败, 那么首先释放所有的锁, 然后重试, 没有对并发跳表做任何改变.
- 4) 对前节点的上锁操作成功, 那么删除键值为 k 的节点, 并发跳表仍然满足不变性质.

综上所述, 引理成立. 证毕.

引理 5. 一旦节点的 *linked* 标志位为 true, 且 *marked* 标志位为 false, 就说明该节点一定存在于抽象映射集合中. 而一旦该节点的 *marked* 标志位为 true, 就说明该节点已经从抽象映射集合中删除.

证明. 除去插入操作, 其他操作不能将 *linked* 标志位置为 true. 并且, 只有节点的 *linked* 标志为 true 时, 才能确认该节点的每一层都与并发跳表相连; 除去删除操作, 其他操作不能将 *marked* 标志位置为 false. 并且, 一旦该点标记 *marked* 为 true, 那么该节点一定会被物理删除.

综上所述, 引理成立. 证毕.

基于上述 5 条引理, 下面针对并发跳表的 3 个主要操作来进行可线性化分析.

1) 查询操作可线性化分析

对于查询操作, 有 3 种可能:

① 如果该节点在调用查询操作前就被标记为 *marked*=true, 且没有任何改变, 那么说明该节点之前一定在集合中(引理 5), 并且没有相同键值的其他节点(引理 1~5: 所有操作满足不变性质), 那么如果找到该节点, 因为该节点的 *marked*=true, 所以该节点已经被删除.

② 如果该节点被判定 *marked*=true, 然后查询操作未返回前该节点被物理删除, 其他线程新增加了一个相同键值的节点. 在这种情况下, 该找到的节点实际上是一个相同键值的未标记节点. 在这种情况下也是可以线性化的, 具体的形式化证明可以参考文献[22].

③ 该节点 *marked*=false, 那么判定该节点的 *linked* 标志位判断是否存在于并发跳表的抽象集合

中. 因此, 一次成功的查询操作, 其可线性化点是找到查询节点, 并且节点 *linked*=true, 且 *marked*=false(算法 5 行③). 而一次失败的查询操作的, 其可线性化点是未找到查询节点, 或者节点 *marked*=true.

2) 插入操作可线性化分析

成功的插入操作的可线性化点在算法 3 行⑩, 根据引理 5, 只有节点的 *linked* 标志位为 true 时, 才能判定节点已经在抽象映射集合中. 一个失败的插入操作可线性化点在算法 3 行④, 即找到已经存在的节点, 那么判定当前的插入操作失败.

3) 删除操作可线性化分析

一个成功的删除操作的可线性化点在算法 4 行⑪, 根据引理 5, 一个节点只有标志 *marked*=true 时才能够从抽象映射集合中删除. 如果没有找到需要删除的节点, 或者该节点已经被其他线程删除, 或者当前节点没有完全链接时(算法 4 行④), 一个失败的删除操作也是可线性化的.

3.5.2 并发跳表活性分析

1) 无死锁

插入和删除操作都是自底向上进行上锁操作. 删除操作首先对当前节点上锁, 然后对前节点上锁, 根据不变性质 3 和不变性质 2, 下层节点的节点数一定不少于上层节点, 并且当前节点的键值一定大于所有前节点的键值. 假设当前节点为 a_i , 前节点 $a[j]$, 对于 $\forall a_j \subset a[i], 0 \leq \text{key}(a_j) < \text{key}(a_i)$. 又因为删除操作只需对前节点进行上锁, 所以, 我们可以定义插入和删除操作获取锁的顺序为 L_i (其中 $\text{key}(L_1) \geq \text{key}(L_2) \geq \text{key}(L_3)$). 设死锁发生的条件为: 假设线程 T_i 当前获取的锁为 L_i , 接下来需要获取锁 $L_{((i+n)-1) \bmod n}$.

显然, 根据引理 3 和引理 4, 插入操作和删除操作保持不变性质, 所以哨兵节点不会发生改变. 根据刚才定义的插入、删除顺序 L_i , 节点只会向左边节点上锁, 而不会向右边节点上锁, 右边节点的键值一定比节点 0 大, 所以节点 0 不会向 L_{n-1} 上锁. 所以, 插入和删除操作是无死锁的.

2) 无等待

查询操作是无等待的, 因为它不考虑并发 skiplist 中的锁和并且没有对并发 skiplist 进行任何改变, 并且没有任何回退、自旋的操作, 所以查询操作一定能在有限步骤内完成.

综上所述分析结果表明: 1) 并发跳表中主要操作(查询、插入、删除)算法都是可线性化的, 符合

安全性要求;2)并发跳表中查询操作是无等待的,插入、删除操作是无死锁的,满足活性要求.

4 实 验

本文设计了2组实验来验证基于并发跳表的云数据处理双层索引架构的性能.

实验 1. 云数据处理双层索引架构可扩展性和范围查询性能测试.

本文基于 Peersim^[15] 模拟器进行了可扩展性验证. 测试的服务器使用的是 Intel Core i3-350M 2.26 GHz CPU, 2 GB RAM, 320 GB 硬盘, 操作系统为 CentOS 6.0 (64 b). 模拟不同尺度的云计算系统, 节点服务器数量从 32 递增至 256. 每一个节点管理 5 000 个资源文件, 每个资源文件的大小从 32~64 KB 不等. 为了对比, 同时也实现了一个文献[9]中介绍的分布式 B+tree 索引.

图 8 与图 9 为双层架构可扩展性测试的比较.

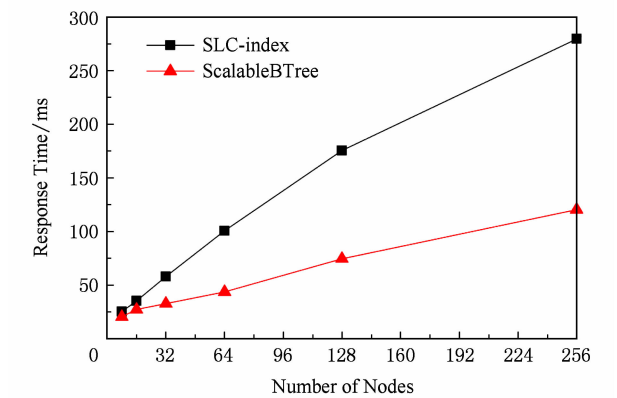


Fig. 8 Range query.

图 8 范围查询

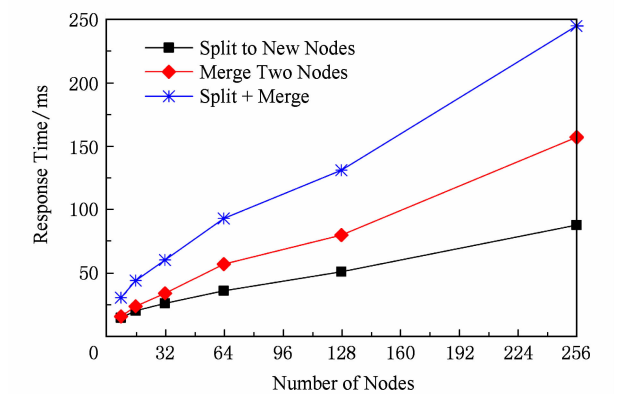


Fig. 9 Dynamic adjusting.

图 9 分裂、合并动态调整

图 8 是双层索引 CSD-index 和 ScalableBTree

的对比,可以看出,随着服务器的增多,CSD-index 的范围查询性能要比 ScalableBTree 好很多. 其根本原因在于 CSD-index 的范围是按照一定顺序存储在局部索引上的,而 ScalableBTree 中节点则是随机存储,这使得在进行范围查找时,ScalableBTree 需要跳跃更多的服务器.

动态分裂和合并算法能解决局部服务器中的热点问题,保证了索引结构整体的负载均衡. 我们对不同局部索引服务器情况下的分裂、合并算法进行了测试. 如图 9 所示,随着局部服务器从 0 增长到 256,分裂、合并算法的时间呈现一个线性增长的趋势.

实验 2. 云数据处理双层索引架构上层并发跳表测试.

我们已经用 Java 语言实现了并发跳表,并将其与 Java. util. concurrent 中的 ConcurrentSkiplistMap 做了对比,后者是目前公认最快的 lock-free 跳表实现. 为了便于对比说明,分别选用 2 种硬件 (11Intel Xenon X5670 2.93 GHz 6 核心 12 线程 CPU, 24 GB RAM. 22Intel Core i5 460 M 2.53 GHz 2 核心 4 线程 CPU, 8 GB RAM) 作为测试平台,软件平台是 Java 2 Platform SE 6. 本文借鉴了 Herlihy^[31] 的实验方法,利用吞吐率,也就是每毫秒的操作数统计来衡量数据结构的性能. 每一次的测试都是从空的跳表开始,进行 1 百万个随机的插入、删除及查询操作,其中键值范围初始化为 200 000. 插入、删除操作的比例越高,就代表操作的冲突率越高. 因此本文针对不同的插入、删除和查询比率,在 2 个硬件平台上分别做了实验.

图 10 与图 11 为在 2 种不同硬件平台下,对不同工作负载时的吞吐率测试对比.

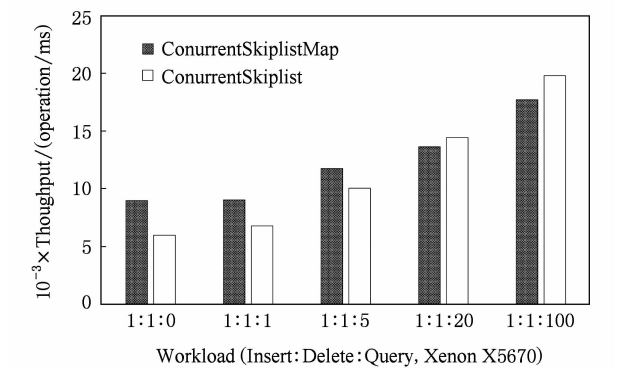


Fig. 10 Throughput comparing with different workload (X5670).

图 10 不同负载下吞吐率对比(X5670)

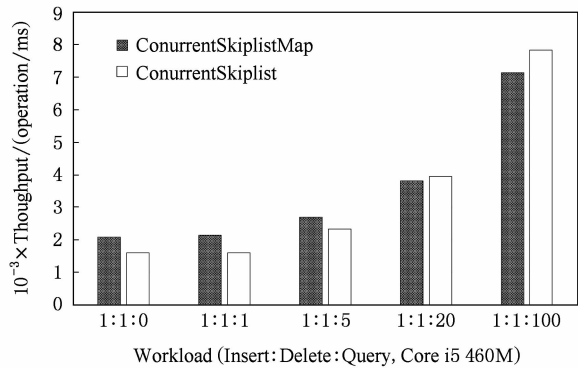


Fig. 11 Throughput comparing with different workload (i5 460M).

图 11 不同负载下吞吐率对比 (i5 460M)

图 10 所示的是 2 种数据结构在 Xenon X5670 平台 1~48 线程下的最大吞吐率. 可以看到, 随着查询操作的比率升高, 我们实现的 ConcurrentSkiplist 可以获得比 ConcurrentSkiplistMap 更好的性能. 在操作比率为 1:1:20, 即查询操作占 90% 的情况下, 相比 ConcurrentSkiplistMap, ConcurrentSkiplist 在 Xenon 平台上提升了 6%, 在操作比率为 1:1:100, 即查询操作占 98% 的情况下, 相比 ConcurrentSkiplistMap, 我们在 Xenon 平台上提升了 14% 的性能. 图 11 所示的是 2 种数据结构在普通笔记本 Core i5 460M 平台 1~16 线程的情况下的最大吞吐率. 与图 10 类似, 在操作比率为 1:1:20 情况下, 提升了 4% 的性能. 在操作比率为 1:1:100 情况下, 提升了 10% 的性能. 造成图 10、图 11 这种现象的最主要原因是 ConcurrentSkiplist 查询操作是 wait-free 的, 因此降低了系统开销.

图 12~15 分别表示在不同插入、删除和查询操作比率时的吞吐率变化.

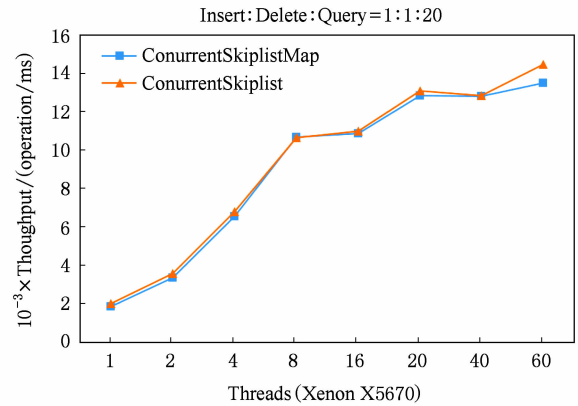


Fig. 12 Throughput comparing with different threads (X5670).

图 12 不同线程下吞吐率对比 (X5670)

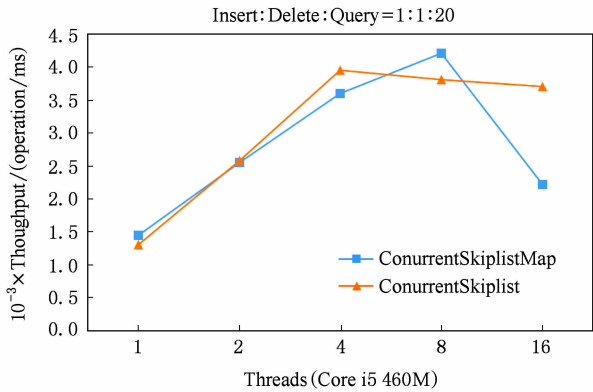


Fig. 13 Throughput comparing with different threads (i5 460M).

图 13 不同线程下吞吐率对比 (i5 460M)

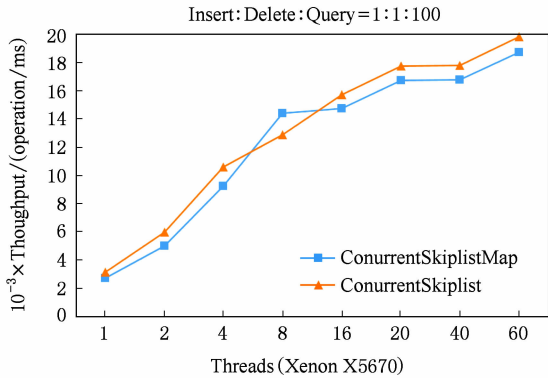


Fig. 14 Throughput comparing with different threads (X5670).

图 14 不同线程下吞吐率对比 (X5670)

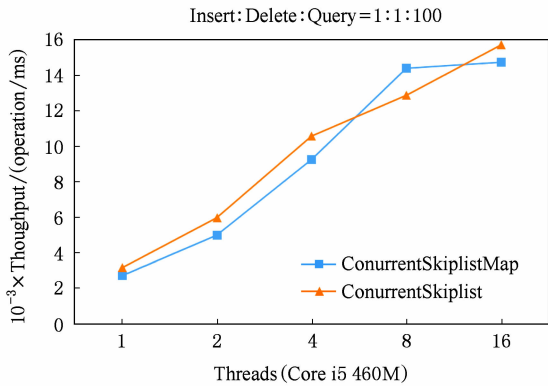


Fig. 15 Throughput comparing with different threads (i5 460M).

图 15 不同线程下吞吐率对比 (i5 460M)

图 12 和图 13 分别表示在插入、删除和查询操作比率为 1:1:20 的情况下, 随着线程的数量的不断增加, 不同平台下吞吐率的变化. 在图 12 中, ConcurrentSkiplist 相对于 ConcurrentSkiplistMap

只有微弱的优势.

图 14 和图 15 分别表示在插入、删除和查询操作比率为 1:1:100 的情况下,随着线程的数量不断增加,不同平台下吞吐率的变化. 在该实验条件下,查询操作比率占 90%以上,ConcurrentSkiplist 相比 ConcurrentSkiplistMap 效率更高.

实验结果表明,ConcurrentSkiplist 具有良好的可扩展性,即随着线程数的增加,性能逐步提升. 并且,ConcurrentSkiplist 在查询操作占高比率时,相对 ConcurrentSkiplistMap 具有更好的性能.

5 总 结

云数据处理双层索引是对当前基于分布式 Hash 键-值对模式的云存储系统的有效补充. 本文提出了基于并发跳表的云数据处理双层索引架构(CSD-index),通过采用 2 层体系结构,突破单台机器内存和硬盘的限制,从而扩展系统整体的索引范围. 设计的动态分裂算法解决局部服务器中的热点问题,保证索引结构整体的负载均衡. 相对于已经提出的其他双层索引架构,CSD-index 在上层全局索引中采用并发技术,可提高全局索引的承载性能和整体索引的吞吐率. 实验结果表明,基于并发跳表的云存储双层索引架构能够有效支持单键查询和范围查询,具有较强的可扩展性和并发性,是一种高效的云存储辅助索引.

参 考 文 献

[1] Amazon Inc. Amazon elastic compute cloud (Amazon EC2) [EB/OL]. [2014-03-25]. <http://aws.amazon.com/ec2>

[2] IBM. IBM introduces ready-to-use cloud computing [EB/OL]. [2014-03-25]. <http://www-03.ibm.com/press/us/en/pressrelease/22613.wss>

[3] Fay C, Jeffrey D, Sanjay G, et al. Bigtable: A distributed storage system for structured data [J]. ACM Trans on Computer Systems(TOCS), 2008, 26(2): No. 4

[4] Microsoft Inc. Windows azure cloud [EB/OL]. [2014-03-25]. <http://www.windowsazure.com>

[5] Sanjay G, Howard G, Leung S. The google file system [C] //Proc of the 19th ACM Symp on Operating Systems Principles(SOSP'03). New York: ACM, 2003: 29-43

[6] The Apache Software Foundation. Hadoop [EB/OL]. [2014-03-25]. <http://hadoop.apache.org/>

[7] Giuseppe D, Deniz H, Madan J, et al. Dynamo: Amazon's highly available key-value store [C] //Proc of the 21st ACM SIGOPS Symp on Operating Systems Principles(SOSP'07). New York: ACM, 2007: 205-220

[8] The Apache Software Foundation. Cassandra. [EB/OL]. [2014-03-25]. <http://cassandra.apache.org/>

[9] Yang H C, Parker D S. Traverse: Simplified indexing on large map-reduce-merge clusters [C] //Proc of Database Systems for Advanced Applications. Berlin: Springer, 2009: 308-322

[10] Dittrich J, Quiané-Ruiz J A, Jindal A, et al. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing)[J]. Proceedings of the VLDB Endowment, 2010, 3(1/2): 515-529

[11] Jimmy L, Dmitriy R, Kevin W. Full-text indexing for optimizing selection operations in large-scale data analytics [C] //Proc of the 2nd Int Workshop on MapReduce and Its Applications. New York: ACM, 2011: 59-66

[12] Dawei J, Beng C O, Lei S, et al. The performance of MapReduce: An In-depth study [J]. Proceedings of the VLDB Endowment, 2010, 3(1/2): 472-483

[13] Richter S, Quiané-Ruiz J A, Schuh S, et al. Towards zero-overhead static and adaptive indexing in Hadoop [J]. The International Journal on Very Large Data Bases, 2014, 23 (3): 469-494

[14] Macros K A, Wojciech G, Mehul A S. A practical scalable distributed b-tree [J]. Proceedings of VLDB Endowment. 2008, 1(1): 598-609

[15] Sai W, Dawei J, Beng Chin O, et al. Efficient B-tree based indexing for cloud data processing [J]. Proceedings of the VLDB Endowment. 2010, 3(1/2): 1207-1218

[16] Jinbao W, Sai W, Hong G, et al. Indexing multi-dimensional data in a cloud system [C] //Proc of the 2010 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2010: 591-602

[17] Zhang X Y, Ai J, Wang Z Y, et al. An efficient multi-dimensional index for cloud data management [C] //Proc of the 1st Int Workshop on Cloud Data Management (CloudDB'09). New York: ACM, 2009: 17-24

[18] Meng Biping, Wang Tengjiao, Li Hongyan, et al. Regional bitmap index: A secondary index for data management could computing environment [J]. Chinese Journal of Computers, 2012, 35(11): 2306-2316 (in Chinese)
(孟必平, 王腾蛟, 李红燕, 等. 分片位图索引: 一种适用于云数据管理的辅助索引机制[J]. 计算机学报, 2012, 35 (11): 2306-2316)

[19] Zhu Xia, Luo Junzhou, Song Aibo, et al. A multi-dimensional indexing for complex query in cloud computing [J]. Journal of Computer Research and Development, 2013, 50(8): 1592-1603 (in Chinese)
(朱夏, 罗军舟, 宋爱波, 等. 云计算环境下支持复杂查询的多维数据索引机制[J]. 计算机研究与发展, 2013, 50(8): 1592-1603)

[20] Steve H, Maurice H, Victor L, et al. A lazy concurrent list-based set algorithm [C] //Proc of the 9th Int Conf on Principles of Distributed Systems (OPODIS'05). Berlin: Springer, 2006: 3-16

[21] Keir F. Practical lock-freedom, UCAM-CL-TR-579 [R]. Cambridge; Cambridge University, 2004

[22] Mikhail F, Eric R. Lock-free linked lists and skip lists [C] // Proc of the 23rd Annual ACM Symp on Principles of Distributed Computing. New York; ACM, 2004; 50-59

[23] Sandip C, Ramesh S, Sushanta K. Concurrent deterministic 1-2 skip list in distributed message passing systems [OL]. 2014 [2014-03-25]. <http://www.tandfonline.com/doi/full/10.1080/VBlmiEqSyEM>

[24] Pedro R P, Thomas D Y, Philippa G. TaDA: A logic for time and data abstraction [EB/OL]. 2014 [2014-03-25]. <http://cs.au.dk/~tyoung/papers/tada.pdf>

[25] Sarwar A, Humaira K, Alan W. A scalable distributed skip list for range queries [C] //Proc of the 23rd Int Symp on High-Performance Parallel and Distributed Computing. New York; ACM, 2014; 315-318

[26] Sarwar A. Distributed skip list in fine-grain message passing interface; Implementation and analysis of a dictionary data structure that supports range queries [OL]. 2014 [2014-03-25]. http://circle.ubc.ca/bitstream/handle/2429/46286/ubc_2014_spring_alam_sarwar.pdf?sequence=4

[27] Kung H T, John T R. On optimistic methods for concurrency control [J]. ACM Trans on Database Systems, 1981, 6(2): 213-226

[28] Maurice H V, Victor L, Mark M, et al. Software transactional memory for dynamic-sized data structures [C] // Proc of the 22nd Annual Symp on Principles of Distributed Computing. New York; ACM, 2003; 92-101

[29] Lucia B. Conflict avoidance: Data structures in transactional memory [OL]. 2006 [2014-03-25]. <http://cs.brown.edu/research/pubs/theses/ugrad/2006/lballard.pdf>

[30] Nathan G B, Jared C, Hassan C, et al. A practical concurrent binary search tree [C] //Proc of the 15th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York; ACM, 2010; 257-268

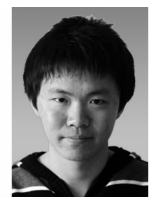
[31] Maurice H, Yossi L, Victor L, et al. A provably correct scalable concurrent skip list [OL]. [2014-03-25]. <http://people.csail.mit.edu/shanir/publications/OPODIS2006-BA.pdf>



Zhou Wei, born in 1974. PhD and associate professor. Member of China Computer Federation. His research interests include distributed computing, cloud computing and multi-cores data structure.



Lu Jin, born in 1986. Master. His research interests include distributed computing, cloud computing(lujin77@gmail.com).



Zhou Keren, born in 1992. Bachelor. His research interests include concurrent data structure(robinho364@gmail.com).



Wang Shipu, born in 1958. Professor. His research interests include computer network(spwang@ynu.edu.cn).



Yao Shaowen, born in 1966. Professor and PhD supervisor. His research interests include workflow, Petri network(yaosw@ynu.edu.cn).