软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(3):1193−1212 [doi: 10.13328/j.cnki.jos.006782]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

跳跃滤波: 一种面向大数据治理的动态数据摘要设计*

符鹏涛[1], 罗来龙[1,2], 郭得科[1], 赵 翔[1], 李尚森[1], 王怀民[2]

[1](国防科技大学 系统工程学院, 湖南 长沙 410073)

[2](国防科技大学 计算机学院, 湖南 长沙 410073)

通信作者: 罗来龙, E-mail: luolailong09@nudt.edu.cn; 郭得科, E-mail: dekeguo@nudt.edu.cn

摘 要: 随着信息技术的迅速发展, 数据体量维持指数增长, 数据价值挖掘困难, 这为数据采集、清洗、存储、共享等数据生命周期中各环节的高效管控带来极大的挑战. 数据摘要技术利用哈希表/矩阵/位向量对数据的频数、基数、成员关系等核心基础特性进行追踪, 使得数据摘要自身成为元数据, 并在共享、传输、更新等场景得到广泛应用. 大数据的快速流转特性更是催生了动态数据摘要技术. 现有的动态数据摘要技术通过动态维护链状或树状结构的概率数据结构列表, 具有其容量随数据流大小而扩增或缩减的优势, 然而也存在空间开销过大以及时间开销随数据基数增加而增长的缺陷. 基于先进的跳跃一致性哈希理论, 设计了一种面向大数据治理的动态数据摘要技术. 该方法可以同时实现随数据基数线性增长的空间开销以及数据处理分析常数级别的时间开销, 能够有效地支撑要求苛刻的多种大数据处理分析任务. 在多种合成和真实数据集上, 通过与传统方法实验对比, 验证了所提方法的有效性和高效性.

关键词: 大数据; 大数据治理; 元数据; 动态数据摘要; 概率数据结构

中图法分类号: TP311

中文引用格式: 符鹏涛, 罗来龙, 郭得科, 赵翔, 李尚森, 王怀民. 跳跃滤波: 一种面向大数据治理的动态数据摘要设计. 软件学报, 2023, 34(3): 1193−1212. http://www.jos.org.cn/1000-9825/6782.htm

英文引用格式: Fu PT, Luo LL, Guo DK, Zhao X, Li SS, Wang HM. Jump Filter: Dynamic Sketch Design for Big Data Governance. Ruan Jian Xue Bao/Journal of Software, 2023, 34(3): 1193−1212 (in Chinese). http://www.jos.org.cn/1000-9825/6782.htm

Jump Filter: Dynamic Sketch Design for Big Data Governance

FU Peng-Tao[1], LUO Lai-Long[1,2], GUO De-Ke[1], ZHAO Xiang[1], LI Shang-Sen[1], WANG Huai-Min[2]

[1](College of Systems Engineering, National University of Defense Technology, Changsha 410073, China)

[2](College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China)

Abstract: With the rapid development of information technology, the volume of data maintains an exponential growth, and the value of data is hard to mine. It brings significant challenges to the efficient management and control of each link in the data life cycle, such as data collection, cleaning, storage, and sharing. Sketch uses a hash table/matrix/bit vector to track the core characteristics of data, such as frequency, cardinality, membership, etc. This mechanism makes sketch itself metadata which has been widely used in the sharing, transmission, update and other scenarios. The rapid flow characteristic of big data has spawned the dynamic sketches. The existing dynamic sketches have the advantage of expanding or shrinking in capacity with the size of the data stream by dynamically maintaining a list of probabilistic data structures in a chain or tree structure. However, there are defects of excessive space overhead and time overhead increasing with the increase of the dataset cardinality. This study designs a dynamic sketch for big data governance based on the advanced jump consistent hash. This method can simultaneously realize the space overhead that grows linearly with the dataset cardinality and the constant time overhead of data processing and analysis, effectively supporting the demanding big data processing and analysis tasks for big data governance. The validity and efficiency of the proposed method are verified by comparing it with traditional methods on various

* 基金项目: 国家自然科学基金(U19B2024, 62002378, 61772544); 国防科技大学科研基金(ZK20-30)
本文由“大数据治理的理论与技术”专题特约编辑杜小勇教授、杨晓春教授和童咏昕教授推荐.
收稿时间: 2022-05-14; 修改时间: 2022-07-29; 采用时间: 2022-09-23; jos 在线出版时间: 2022-10-27

datasets, including synthetic and natural datasets.

Key words: big data; big data governance; metadata; dynamic sketch; probabilistic data structure

近年来, 随着大数据、云计算、人工智能、区块链、物联网、5G 等信息技术的迭代升级, 人类产生的数据量正在以指数级的速度在增长, 预计在 2025 年将达到 163 ZB 左右. 面对如此海量且价值密度稀疏的异构数据, 大数据治理应运而生[1]. 然而, 大数据的 4V 特性给大数据治理带来了巨大的冲击, 体现在数据集成存储困难、数据质量不高、数据共享不足、数据安全隐患突出等, 为数据采集、清洗、存储、共享等数据生命周期中各环节的高效管控带来了极大的挑战. 大数据已成为物质与能量之外的第三大社会资源, 如何充分利用数据资源、如何以合理代价处理不断增长的海量数据、如何更快捷地从海量异构的复杂数据中快速提取有价值的信息, 成为大数据治理的核心难题. 然而, 传统的数据治理技术, 例如底层基础设施扩容和抽样技术, 存在计算能力无法与海量数据处理需求匹配以及数据失真等缺陷, 无法为大数据治理提供高质量服务支撑. 为此, 当前大数据治理倾向于构建元数据来对原始信息资源或数据加以管理、利用并产生价值.

数据摘要技术能够以低空间开销实现海量数据的价值汇聚和压缩表示, 是解决上述大数据治理核心难题的有效手段. 具体来说, 数据摘要技术将海量异构数据元素作为输入, 通过哈希函数映射到哈希表、矩阵或位向量中, 并改变相应位或者计数器的值, 最终, 数据摘要技术将自身构建为包含数据集特定特征的元数据并输出, 以此对数据的频数、基数、成员关系等核心基础特性进行追踪, 并据此支持数据的灵活管理和高效挖掘.传统的数据摘要技术往往用来表示尺寸大小已知且固定的数据集, 其容量大小由数据结构初始化时分配空间的多少所决定. 这导致传统数据摘要技术的容量大小是固定的, 无法根据输入数据集的实时数据量大小进行调整, 也无法为大小未知的数据集分配合适的存储空间. 然而, 大数据具有实时动态特性, 例如互联网大数据、社交网络大数据、社会公共领域大数据等[2,3], 数据资料随时随地产生, 数据的收集、存储和处理也呈现出动态性. 此外, 这些数据的体量更加庞大且不断动态变化, 导致其相应的治理对各类资源的利用更加苛刻、对时延更加敏感. 因此, 为大数据治理提供数据处理分析支撑的数据摘要技术, 需要能够根据数据流实时大小调整数据结构的自身容量. 为此, 动态数据摘要技术应运而生. 然而, 大数据的快速流转特性不断凸显, 不仅要求动态数据摘要技术能够实现随数据集基数大小线性变化的存储空间开销, 最大限度地利用存储空间资源, 同时还要求动态数据摘要技术尽可能地降低数据处理分析的计算复杂度, 力图达到与数据集基数无关的常数级别时间开销. 这些设计原理如果得以实现, 就节省存储空间和服务质量而言, 将为动态大数据的表示和查询等处理分析任务以及大数据治理的服务支撑带来前所未有的好处.

典型的数据摘要技术有 Bloom 滤波[4]、Quotient 滤波[5]、Cuckoo 滤波[6]等. 一方面, 数据摘要技术通过对数据进行压缩表示构建元数据, 例如区块链系统利用 Bloom 滤波作为元数据来实现交互式集合同步[7], CDN (内容分发网络)中多个服务器通过Bloom滤波将数据信息构建为元数据进行存储[8]; 另一方面, 数据摘要技术能在元数据的共享、传输、更新等场景得到应用. 具体来说, 近年来, 业界更加趋向于依托多种 sketch 构建大规模数据摘要信息[9], 达成空间节约和价值汇聚的双重目的, 从而高效支撑各种基于元数据的大数据治理任务. 在数据采集方面, 例如网络测量任务利用数据摘要技术记录网络流的尺寸信息[10]; 在数据集成方面, 例如 IP 查找和网络包分类等任务利用数据摘要技术对目的 IP 地址等数据信息进行集成表示, 从而快速响应查询任务[11]; 在数据存储方面, 例如 P2P 网络通过数据摘要技术存储各节点上的对象内容[12]; 在数据共享方面, 例如集合同步利用数据摘要技术实现不同主机的文件信息共享[13]; 在数据安全方面, 数据摘要技术通过哈希计算仅表示数据的摘要信息, 天然地实现数据加密操作, 确保数据隐私安全. 数据摘要技术在其他领域也得到广泛应用, 例如分布式数据流监控、异常检测、突发检测、大流检测[14−18]等.

然而, 如表 1 所示, 现有的动态数据摘要技术却不能同时实现线性增长的空间开销和常数级别的时间开销这两个基本原理. 文献[19]通过使用多个同构计数 Bloom 滤波(counting Bloom filter, CBF)[20]组成链表, 设计了动态 Bloom 滤波(dynamic Bloom filter, DBF), 最早提出了支持容量缩减的动态数据摘要技术. 尽管 DBF 实现了随数据集基数增加而线性增长的空间开销, 但由于 CBF 因实现支持元素删除而需要昂贵的空间开销,

DBF 也继承了这一代价. 此外, DBF 的查询操作需要检查所有 CBF 块, 导致其时间开销也随数据集基数增加呈线性增长. 文献[21]采用和 DBF 类似的方法设计了动态 Cuckoo 滤波(dynamic Cuckoo filter, DCF), DCF 使用 CF 块替换了 DBF 中的 CBF 块, 在一定程度上节省了空间开销. 文献[22]在 DCF 的基础上, 利用一致性哈希环[23]改进了传统的 Cuckoo 滤波, 进而设计出一致性 Cuckoo 滤波(consistent Cuckoo filter, CCF). CCF 实现了细粒度的容量调整, 进一步节省了空间开销, 但由于计算复杂度过高, 导致其查询操作相应的时间开销显著增加. 文献[24]提出了对数动态 Cuckoo 滤波(logarithmic dynamic Cuckoo filter, LDCF), 其使用二叉树结构维护多个同构 CF 块组成的动态列表, 将查询对象从所有 CF 块降低到了部分或单个 CF 块, 实现了对数级别的查询开销. 然而, 二叉树结构也不可避免地造成了指数级别增长的空间开销.

表 1 动态数据摘要技术的经验对比(Y/N 代表是/否, +越多代表开销越大)

		DBF[19]	DCF[21]	CCF[22]	LDCF[24]	JF
空间开销	线性增长	Y	Y	Y	N	Y
	开销大小	++++	++	+	+++	++
时间开销	常数级别	N	N	N	N	Y
	开销大小	+++	+++	++++	++	+

基于以上分析不难发现, 已有的动态数据摘要技术都是在时间开销和空间开销之间进行权衡工作, 且未能实现支持常数级别开销的查询操作, 无法满足面向大数据治理的数据分析处理任务的严格要求. 动态数据摘要技术设计的难点主要体现在以下两个方面: (1) 如何在数据元素和基本数据结构块(例如 CBF 或 CF 块)之间建立起一对一的索引联系, 确保集合成员检测的时间开销和数据集基数大小无关, 为常数级别; (2) 如何支持增删单个基本数据结构块以实现弹性容量, 从而确保动态数据摘要技术的空间开销随数据集基数线性增长. 因此, 本文基于跳跃一致性哈希算法[25]和 Cuckoo 滤波设计了一种新颖的动态数据摘要技术, 称为跳跃滤波(jump filter, JF). 跳跃滤波在保证随数据集基数增加其空间开销线性增长的前提下, 实现了和数据集基数大小无关的常数级别开销的查询操作, 实现了上述设计原理, 有效地支撑了动态大数据的表示和查询等处理分析任务, 为大数据治理提供了更高质量的服务.

本文第 1 节介绍动态数据摘要技术以及跳跃一致性哈希算法的相关方法和研究现状. 第 2 节介绍本文构建的跳跃滤波及其自适应变种. 第 3 节对跳跃滤波这一最新动态数据摘要技术研究成果展开理论分析. 第 4 节通过综合对比实验验证跳跃滤波的可行性和有效性. 最后总结全文.

1 相关工作

数据摘要技术对服务大数据治理具有重要的理论价值和现实意义. 业界提出了众多数据摘要技术[9], 其中以 Bloom 滤波[4]和 Cuckoo 滤波[6]等概率数据结构应用最为广泛. 这些数据结构往往用来表示集合大小已知的数据集. 然而, 众多领域所产生的大数据天然具有动态性, 数据在短时间内的大量产生和移除已经成为常态, 并且其数据规模难以预测. 这使得传统的静态数据结构极易造成容量不足或者空间浪费问题. 为此, 业界提出了众多能够根据实时数据流大小按需扩展或缩减容量的动态数据摘要技术, 例如动态 Bloom 滤波[19]、一致性 Cuckoo 滤波[22]、动态 Cuckoo 滤波[21]以及对数动态 Cuckoo 滤波[24]. 表 2 对本文使用的关键符号进行了解释说明.

表 2 本文所使用的符号及其解释说明

符号	解释说明	符号	解释说明
k	数据结构所使用的哈希函数的数量	α	数据结构的负载系数
m	位向量或哈希表的长度	η	元素的指纹
ξ	数据结构的假阳性率	C	跳跃滤波的目标总容量
b	哈希表中单个候选桶的容量	B	跳跃滤波初始化时所含 CF 块数量
s	集合的规模大小, 即所含元素的数量	JS	跳跃滤波存储元素的实时数量
f	元素指纹的长度, 即其比特位的数量	Th	容量回收算法中的空间利用率阈值

1.1 Bloom滤波及其弹性设计

Bloom 滤波(Bloom filter, BF)[4]是一种广泛用于集合成员检测的概率数据结构. BF 通过 m 位的位向量表示元素, 所有位初始设为 0. 为了插入有 s 个元素的集合 S 中的任意元素 x, BF 需要使用 k 个独立的哈希函数将元素映射到位向量中的 k 个位置, 然后将位数组中映射的 k 个位置都设置为 1. 图 1 是 BF 工作原理的简单示例, 其中, BF 维持 m=23, k=3, 用来表示集合 S={x_1,x_2,x_3}. 为了插入集合 S 中的元素 x_1, BF 首先使用 3 个独立哈希函数 h_1, h_2 和 h_3 将 x_1 分别映射到位向量上第 2、6、11 个比特位上, 接着, 将这 3 个比特位的值置为 1. 为了查询 x, BF 只检查 k 个对应的位置. 如果所有 k 个位都为 1, 则 BF 返回正结果, 表示 x 是 S 的成员.例如图 1 中 BF 查询 x_1, h_1, h_2 和 h_3 映射到的所有比特位的值都为 1, 说明 $x_1 \in S$. 如果任意映射位为 0, 则 BF 判断 $x \notin S$ 并返回负结果. 例如图 1 中 BF 查询 x_5, 其中某个哈希函数映射到的第 12 个比特位为 0, 表明 $x_5 \notin S$. 然而在集合成员关系查询时, BF 存在潜在的假阳性概率 ξ, 而无假阴性概率. 具体来说, 由于不可避免的哈希冲突,当位向量的 k 个哈希位置都为 1 时, BF 可能会将一个不存在目标集合中的元素误认为集合成员. 例如图 1 中 BF 查询某不属于集合 S 的元素 x_4, 由于哈希冲突检测到 x_4 的 3 个映射位都为 1, BF 将错误判断 $x_4 \in S$. 出现这种错误的概率是 $\xi_{BF}=(1-e^{-ks/m})^k$. 直观上, 标准 BF 不支持元素删除, 除非它重构整个位向量. 原因是直接将对应位从 1 置为 0, 可能会导致其他元素的假阴性结果. 许多 Bloom 滤波变种被提出来, 以改进 Bloom 滤波在不同环境下的性能[26−29].

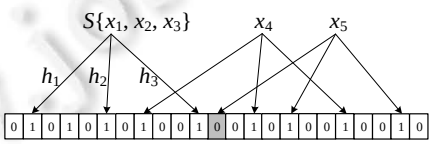


图 1 Bloom 滤波工作原理示意图

为了解决标准 Bloom 滤波不支持元素删除的问题, Guo 等人[20]提出了计数 Bloom 滤波. CBF 的基本观点是, 将 Bloom 滤波中的比特位替换为具有多个比特的计数器. 当插入元素时, 元素被映射到 CBF 向量中的 k 个位置, 并将这些位置的计数器累加 1. 查询元素时, CBF 核对该元素在相应 k 个位置的计数器值: 倘若都不为 0, 则认为元素属于集合; 否则认为元素不是集合成员. 删除元素时, 只需将相应 k 个位置的计数器值递减 1 即可. 在 CBF 的基础上, Guo 等人[19]针对已有 Bloom 滤波设计由于无法支持容量的动态缩减而不能应对动态数据集合表示的问题, 提出了动态 Bloom 滤波. 为此, 动态 Bloom 滤波提出按需动态维持多个同构的计数 Bloom 滤波向量, 且每个计数 Bloom 滤波的容量固定. 当数据集合体量增加时, 动态 Bloom 滤波启用新的空计数 Bloom 滤波; 当数据集合体量减小时, 动态 Bloom 滤波对具有较低利用率的计数 Bloom 滤波向量进行合并, 从而实现空间回收. DBF 的假阳性误判概率为

$$\xi_{DBF}=1-(1-(1-e^{-kc/m})^k)^{\lfloor s/c \rfloor}.$$

其中, k 为使用的哈希函数数量, c 和 m 分别为单个计数 Bloom 滤波的容量和计数器个数, s 为所表示数据集的基数.

1.2 Cuckoo滤波及其弹性设计

基于 Cuckoo 哈希[30], Fan 等人[6]提出了一种轻量级概率型数据结构, 称为 Cuckoo 滤波(Cuckoo filter, CF). CF 基于 Cuckoo 哈希表构建, 并支持常数量级成员关系查询. CF 并不是直接存储元素内容, 而是存储元素的指纹. 如图 2 所示, 结构上来看, CF 由 m 个桶组成, 每个桶有 b 个存储槽用于存储 b 个元素指纹(图中 b=4). 任何元素 x 有唯一的 f 位比特的元素指纹 η_x, 并通过哈希函数 h_0 计算而来. 在此种结构下, CF 面临的一大难题是, 在重分配过程中, 如何在不知道被踢出元素的原始内容的情况下, 快速确定被踢出元素的第 2 个候选桶. CF 采用部分键值 cuckoo 哈希策略来解决这一问题. 元素的第 2 个候选桶可以通过当前桶所在位置与元素指纹哈希值之间的异或结果来确定. 也就是说, 候选桶的计算方式为: $h_1(x)=hash(x)$, $h_2(x)=h_1(x)\oplus hash(\eta_x)$. 根据以上设计, 在插入元素 x 时, CF 首先计算元素指纹 η_x, 再通过 h_1 和 h_2 计算元素 x 的候选桶. 随后, 如果任意一

个候选桶有空的存储槽, η_x 将被存储在该候选桶中. 如果 x 的两个候选桶都存满, 不含有空槽, 那么 CF 将对已存储的元素指纹执行重分配操作, 随机踢出 x 的两个候选桶中已存储的某元素, 例如图 2 中的指纹 η_u. 接着, 将元素 x 存放在腾出的空槽中. 如果存储过程中的指纹重分配次数达到额定的阈值, 则认为 CF 已经存储满了, 此时存储会失败. 在查询元素 y 是否属于一个集合 A 时, CF 将在元素 y 的两个候选桶中搜索 η_y: 如果 η_y 能够在其中任何一个候选桶中找到, 则 CF 判定 $y\in A$; 否则, CF 认为 $y\notin A$. 由于元素指纹潜在的哈希冲突, CF 可能导致成员关系查询的假阳性错误(将一个不属于集合 A 的元素误判为集合 A 的成员). 理论上, 可以将假阳性误判概率界定为: $\xi_{CF}=1-(1-1/2^f)^{2b}=2b/2^f$. 其中, f 是元素指纹长度, b 是每个候选桶最多能够存储的指纹数量. 如果集合 A 中元素都被成功插入 CF 中, 则不会产生假阴性误判错误. 需要注意的是, 由于 CF 通过异或计算来推导元素的候选桶位置, 因此标准 CF 中必须满足 m 的值, 即哈希表中候选桶的总数, 必须为 2 的幂次方. 此外, CF 通过删除元素在候选桶中存储的指纹来支持元素删除操作. 但需要注意的是, 所删除的对象必须是已经存储的元素, 否则可能由于哈希冲突导致假阴性错误.

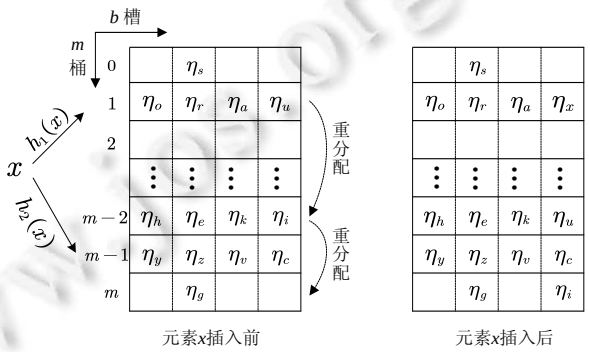


图 2 Cuckoo 滤波工作原理示意图

受到动态 Bloom 滤波的启发, Chen 等人[21]提出了动态 Cuckoo 滤波. DCF 同样动态地维持多个同构的 CF, 以支持 CF 容量的动态变更. 最初只存在一个 CF 并被标记为活跃 CF, 后续的 CF 将会以一种主动或被动的方式引入到 DCF 中. 同时, DCF 设计了一种空间回收策略, 将低利用率的 CF 元素迁移到其他 CF 中并实现空间回收. DCF 的假阳性误判概率为 $\xi_{DCF}=1-(1-\xi_{CF})^n=n\xi_{CF}=2bn/2^f$, 其中, n 是 DCF 中 CF 块的数量.

Luo 等人[22]注意到, 现存的数据结构, 包括 Bloom 滤波和 Cuckoo 滤波及其变种, 都存在存储元素的单元格和数据结构的长度维持着紧密联系的问题. 这种依赖会降低这些数据结构在表示动态集合时的容量弹性和空间效率. 为了解决这一问题, Luo 等人利用 Consistent 哈希[23]首先创造性地提出了索引独立 Cuckoo 滤波(index-independent Cuckoo filter, I2CF), 本质上, I2CF 维持一致性哈希环来为每个元素分配 k 个候选桶, 从而达到解耦过滤器长度与候选桶索引值的目的. 对于动态集合, I2CF 通过动态的增加、删除桶, 支持桶级别的容量改变. 对于集合基数突然增加或者减少的情况, Luo 等人进一步组合了多个 I2CF 作为一致性 Cuckoo 滤波, 从而支持数据结构级别的伸缩性. 通过增加未使用的 I2CF 或者合并低使用率的 I2CF, CCF 可以随时调整容量大小. CCF 实现了桶级别的细粒度容量调整, 同时也解决了标准 CF 中候选桶数量必须为 2 的幂次方的问题. 然而, 由于使用的一致性哈希策略较为复杂, 导致 CCF 存储、查询和删除的吞吐量相比标准 CF 都有大幅度下降. 此外, 经典的一致性哈希环需要数据结构的支撑, 因此存在一定的额外空间开销, 其空间复杂度为 O(N). I2CF 的假阳性概率为: $\xi_{I2CF}=1-(1-1/2^f)^{kb}$; 类似于 DCF, CCF 的假阳性为: $\xi_{CCF}=1-(1-\xi_{I2CF})^n=n\xi_{I2CF}$, 其中, ξ_{I2CF} 为每一个 I2CF 的假阳性误判概率, n 是 CCF 中 I2CF 的数量.

Zhang 等人[24]提出了对数动态 Cuckoo 滤波. LDCF 设计了一种可扩展的多级树结构来连接 CF 块, 而不是使用类似于 DCF 的经典链状结构. LDCF 进一步根据元素指纹的前缀比特位来索引该元素在树结构中每个层级上的候选 CF 块. 这样的设计显著降低了集合成员检测的时间开销,其计算成本达到了对数级别. Zhang 等人进一步通过将父节点 CF 块中的数据信息迁移到子节点 CF 块中, 设计了一种压缩型 LDCF. 该优化操作使得

集合成员检测任务只关注于某个特定的 CF 块, 计算成本和时间开销进一步降低. 然而, 只有在极特殊的情况下, 即当数据集的尺寸大小大约是单个 CF 块容量的 2^{N_+} 倍(N_+为正整数), 此时, 只有所有 CF 块都处在树结构上同一层级之中, 压缩型 LDCF 才有可能实现常数级别查询. 通常情况下, LDCF 及其压缩型变种需要遍历树结构来搜寻目标 CF 块, 因此, LDCF 实际上并不能将成员检测的计算复杂度减少到 O(1), 即常数级别. LDCF 压缩型变种的假阳性误判概率为: $\xi_{LDCF}=2bn/2^f$, 其中, n 是 LDCF 压缩型变种中的 CF 块数量.

除了动态变种以外, Cuckoo 滤波还有许多其他典型变种, 包括完美 Cuckoo 滤波[31]、自适应 Cuckoo 滤波[32]、简化 Cuckoo 滤波[33]、垂直 Cuckoo 滤波[34]和真空 Cuckoo 滤波[35]. 这些变种提升了 Cuckoo 滤波在误报率、理论保证、空间效率和查询吞吐量等方面的性能, 进一步拓展了其应用范围.

1.3 跳跃一致性哈希算法

传统的哈希表设计中, 添加或者删除一个存储单元格, 会造成全局数据的重新映射. 为了减少目标存储单元格数量动态变化引起的数据迁移量, 人们提出了多种一致性哈希算法, 包括最常用且最经典的一致性哈希环算法[23]. 然而, 哈希环算法存在执行速度慢、内存消耗大、依赖数据结构支撑以及映射不够均匀等缺点. 为了解决这些弊端, John 等人[25]提出了跳跃一致性哈希(jump consistent Hash).

首先, 假设目标一致性哈希函数是 $ch(\tau,n)$, 其中, τ表示某数据元素的标签或指纹, n 表示存储单元格的数量, 数据元素的总量为 K. 为了映射均匀, 需要保证每个存储单元格要映射到 K/n 个元素. 当存储单元格数量由 n 变为 n+1 时, 则需要将 $K/(n+1)$个已存储的元素重新映射到第 n+1 个新增的空存储单元格中. 为此, ch 函数采用随机数来决定某已存储元素在存储单元格数量增加之后, 是否需要跳到新存储单元格中去. 具体来说, 当存储单元格数量由 n 变为 n+1 时, 对每一个已存储元素, ch 函数使用τ作为随机数种子, 生成一个关于τ的随机序列. 接着, 判断随机序列中第 n 个随机数是否小于 $1/(n+1)$: 若是, 则将该元素迁移到第 $1/(n+1)$个存储单元格中; 否则, 该元素将留在原来的存储单元格中.

上述构造的 ch 函数已经达到了一致性哈希算法的定义标准. 此时, ch 函数的时间复杂度为 O(n). 这是因为当存储单元格数量为 n 时, ch 函数需要根据随机序列进行 n–1 次比较判断来决定数据元素的存储位置. 然而在整个重新映射过程中, 数据元素跳到新存储单元格中的概率较低. 根据这一观点, 可以进一步优化 ch 函数. 具体优化操作为: 计算出数据元素连续不换存储单元格的概率, 当符合该概率时, 直接跳过不换存储单元格阶段的比较操作. 该优化操作使得 ch 函数的时间复杂度降低到了 O(ln(n))级别, 最终实现跳跃一致性哈希算法的构建. 跳跃一致性哈希利用伪随机数实现了最小化重新映射, 保证了完全的一致性和均匀性, 且不产生额外的空间开销. 然而, 跳跃一致性哈希只能在尾部增删存储节点. 另外, 需要注意的是, 跳跃一致性哈希函数返回结果的序列是从0开始计算的. 例如, 当只有一个存储节点时, 跳跃一致性哈希函数返回的结果为 0, 表示第 1 个存储节点的序列号.

综上所述, 现有的动态数据摘要技术, 例如动态 Bloom 滤波、动态 Cuckoo 滤波以及一致性 Cuckoo 滤波, 都维护多个 CBF 块或 CF 块构成的链表来实现动态变更的弹性容量. 当需要表示的数据集尺寸越大, 这些链状的动态数据摘要技术就需要维护更长的 CBF 块或 CF 块构成的链表. 然而, 当涉及查询或者删除某元素时, 这些动态数据摘要技术需要从头到尾依次检查链表中每个 CBF 块或 CF 块. 在最坏的情况下, 检查次数就是链表的长度. 这种情况下, 成员检测的计算复杂度为 O(n). 这意味着当数据集规模扩大时, 涉及集合成员检测的查询和删除操作的时间开销呈线性增长, 这对于大规模动态数据集的表示无疑是不友好的. 为此, 对数动态 Cuckoo 滤波和其压缩型变种采用一种多级二叉树结构来连接 CF 块. 相比于采用经典链表结构的动态数据摘要技术, 对数动态 Cuckoo 滤波只需检查树结构中所有层级上特定的 CF 块, 即可完成集合成员检测任务. 这样的设计显著降低了集合成员检测的时间开销, 其计算复杂度降低到了对数级别. 在特定情况下,压缩型 LDCF 能够维护所有的 CF 块在树结构中同一层级上, 使得计算成本和时间开销进一步降低. 通常情况下, LDCF 及其压缩型变种需要遍历树结构来搜寻目标 CF 块, 只是将成员检测的计算复杂度减少到对数级别.此外, 由于采用了树结构, 对数动态 Cuckoo 滤波及其压缩型变种的空间开销呈指数级别增长, 这对于大规模动态数据集的表示同样是难以接受的. 因此, 本文设计了跳跃滤波. 不同于链状或树状动态数据摘要技术中元

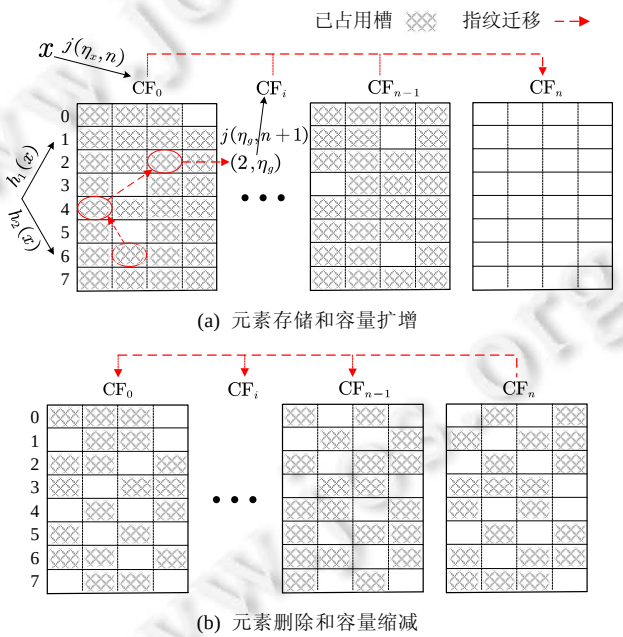
素与候选 CF 块是一对多的关系, 跳跃滤波使用先进的跳跃一致性哈希算法, 在元素和候选 CF 块之间建立起了一对一的索引联系. 因此, 跳跃滤波仅查询单个 CF 块就可完成集合成员检测任务. 此外, 跳跃一致性哈希算法的机制使得跳跃滤波支持增删单个CF块来实现弹性容量. 总的来说, 跳跃滤波可以同时实现常数级别的时间开销和线性增长的空间开销, 这是传统动态数据摘要技术所无法实现的.

2 跳跃滤波的设计

本文基于 Cuckoo 滤波和跳跃一致性哈希算法设计跳跃滤波, 是一种新颖的 Cuckoo 滤波变种. 不同于传统动态数据摘要技术的链表或树结构, 跳跃滤波维持多个同构CF块组成的动态列表, 通过跳跃一致性哈希算法, 在数据元素和CF块之间建立起一对一的映射关系, 即根据元素指纹的哈希计算结果为该指纹索引一个候选 CF 块提供存储. 同时支持在列表末端增删单个 CF 块来按需调整数据结构的容量, 实现线性增长的空间开销和常数级的查询时间开销两大设计原理. 下面就跳跃滤波的初始化、插入操作、查询操作、删除操作以及容量增减操作予以介绍.

2.1 跳跃滤波的初始化

如图 3 所示, 跳跃滤波由多个 CF 块构成的动态列表组成, 跳跃滤波同样以槽作为最小存储单元, 并且通过在哈希表中的特定位置存储元素的指纹来实现对元素的压缩表示.



(a) 元素存储和容量扩增

(b) 元素删除和容量缩减

图 3 跳跃滤波示例图

在此介绍跳跃滤波的初始化方法, 详细过程如算法 1 所示. 初始化的输入包括数据结构的总目标容量 C、整体目标假阳性ξ_{JF}、单个 CF 块所含桶的数量 m、单个 CF 块每个桶所含的槽数量 b、单个 CF 块的负载系数 (load factor) α、单个 CF 块的指纹重分配次数阈值 MAX、列表中初始 CF 块的数量 B. 其中, 负载系数α是指CF所能存储的元素数量与该CF的槽数量的占比. 首先, 跳跃滤波计算出其表示目标数据集所需的CF块数量 n. 接着, 判断 B 的值是否大于 n: 若是, 则令 B=n. 其次, 确定跳跃滤波为不超过目标假阳性率而需要使用的最小指纹长度 f. 那么, 跳跃滤波所使用的同构CF块就可以初始化为: $CF(m,b,f,MAX)$. 最后, 创建包含 B 个该CF块的列表并返回.

算法 1. 初始化跳跃滤波.

输入: C, ξ_{JF}, m, b, α, MAX, B.

输出: JF.

1. $n=\lceil C/(\alpha\times m\times b)\rceil$;
2. **If** $B>n$ **then**
3. $B=n$;
4. $f=\lceil \log_2(2bn/\xi_{JF})\rceil$;
5. JF=[[$CF(m,b,f,MAX)$] for _ in $range(B)$];
6. **Return** JF

另外, 值得注意的是, B 的数值越大, 指纹迁移次数越少, 存储吞吐量越大, 但潜在的空间开销也越大. 用户可以根据自身需求在存储时间开销和空间开销之间进行权衡, 从而确定合适的初始 CF 块数量. 为避免空间浪费, 本文推荐将 B 设为 1. 算法 1 中描述的初始化方法以不超过预定假阳性为目标, 根据输入信息确定滤波所需的最小指纹长度. 其中, 指纹长度的确定方法由跳跃滤波的假阳性公式推导而来, 这点将在第 4 节的理论分析展开介绍. 实际上, 用户可根据自身需求设计不同的初始化方法.

2.2 元素插入以及容量扩增的基本操作

在介绍跳跃滤波的插入算法之前, 需要明确以下定理:

定理 1. 已知 CF_a 在插入某元素过程中, 由于指纹重分配次数达到额定阈值而从第 u 号桶中踢出某指纹 η_g. 那么根据已知信息(u,η_g), 另一个同构的 CF_b 可以实现针对指纹 η_g 的插入操作.

证明: CF 利用部分键值 cuckoo 哈希策略, 即 $h_1(g)=hash(g)$, $h_2(g)=h_1(g)\oplus hash(\eta_g)$来索引存储元素 g 的两个候选桶, 进而结合指纹重分配策略完成存储任务. 对于定理 1 中的(u,η_g), u 可能是 $h_1(g)$, 也可能是 $h_2(g)$. 实际上, 无论 u 是哪一个候选桶, 都可以通过部分键值 cuckoo 哈希策略推导出 g 的另一候选桶 s. 一方面, 假设 $u=h_1(g)$, 那么有 $s=u\oplus hash(\eta_g)=h_1(g)\oplus hash(\eta_g)=h_2(g)$; 另一方面, 假设 $u=h_2(g)$, 那么有 $s=u\oplus hash(\eta_g)=h_2(g)\oplus hash(\eta_g)=h_1(g)\oplus hash(\eta_g)\oplus hash(\eta_g)=h_1(g)$. 同构 CF 由于哈希表构造一样、使用哈希函数一样, 因此在获取某元素的指纹及任一候选桶索引这两个基本信息后, CF_b 可以插入 CF_a 踢出的受害指纹. □

图 3(a)以存储数据元素 x 为例, 展示了跳跃滤波的插入以及扩容操作. 如图所示, 跳跃滤波首先通过哈希函数计算出 x 的指纹 η_x, 并获取跳跃滤波的当前列表长度 n, 即所含 CF 块数量; 接着, 通过跳跃一致性哈希算法 $j(\eta_x,n)$推导出 x 的候选 CF 块, 例如图中的 CF_0; 然后, 尝试在 CF_0 中插入 η_x, 如果插入成功, 则 x 的存储操作结束. 注意: 跳跃滤波维护一个全局计数器 JS 用来记录跳跃滤波存储的元素的实时数量, 一旦新元素存储成功, 则将 JS 的数值增加 1, 最后返回 True. 然而, CF_0 由于达到了负载系数而导致插入失败并踢出了某指纹 η_g, 此时, 跳跃滤波在列表末端增添了一个新的空 CF 块, 即 CF_n. 接着遍历从 CF_0 到 CF_{n-1} 中所有的指纹, 例如 η_y, 根据 $j(\eta_y,n+1)$的值是否为 n 来决定 η_y 是否需要重新映射到 CF_n 之中. 最后, 将受害指纹 η_g 重新映射到第 $j(\eta_g,n+1)$个 CF 块中. 注意: 如果上述指纹重新映射过程中发生插入失败, 跳跃滤波将再次增添新的 CF 块, 并重新遍历已有指纹, 重新映射需要迁移的指纹以及所有因插入失败而被踢出的指纹, 直到所有插入操作成功.此时, 关于 x 的存储操作结束, 返回 True. 跳跃滤波插入数据元素以及插入失败时自动扩容的详细过程如算法 2 所示.

算法 2. 跳跃滤波插入元素 x.

输入: x, JF.

输出: True.

1. $\eta_x=h_0(x)$, $n=length(JF)$, $i=j(\eta_x,n)$;
2. **If** (CF_i insert η_x)==True **then**
3. JS++, **return** True;
4. **Else**

5. *Extend*(*JF*);
6. 重新插入所有因插入失败而被踢出的指纹, **If** fails **then**
7. 返回步骤 5;
8. **Return** True.
9. Function *Extend*(*JF*):
10. n=*length*(*JF*);
11. **For** *num*=0; *num*<*n*; *num*++ **do**
12. **For** η in CF_{num} **do**
13. **If** $j(\eta,n+1)==n$, 将η迁移到 CF_n 中, 记录因插入失败而被踢出的指纹 *victims*;
14. **Return** *victims*.

2.3 元素的查询操作

相比于插入操作, 跳跃滤波的查询操作更为简洁. 算法 3 以查询元素 x 为例, 详细介绍了跳跃滤波的查询操作. 跳跃滤波首先通过哈希函数 h_0 计算出 x 的指纹η_x. 接着, 通过部分键值 cuckoo 哈希策略计算出 x 的两个候选桶的位置, 即 $h_1(x)$和 $h_2(x)$. 然后获取跳跃滤波当前的列表长度 n, 通过跳跃一致性哈希算法 $j(\eta_x,n)$计算出 x 的候选 CF 块, 即 $CF_i(i\in[0,n-1])$. 最后检查 CF_i 的候选桶 $h_1(x)$或 $h_2(x)$中是否存在指纹η_x: 若存在, 则返回 True, 表示元素 x 存在跳跃滤波所表示的集合之中; 否则返回 False, 表示元素 x 不属于目标集合.

算法 3. 跳跃滤波查询元素 x.

输入: x, *JF*.

输出: True 或 False.

1. $\eta_x=h_0(x)$;
2. $h_1(x)$=*hash*(x), $h_2(x)=h_1(x)\oplus$*hash*(η_x), n=*length*(*JF*), $i=j(\eta_x,n)$;
3. **If** bucket $h_1(x)$ or $h_2(x)$ in CF_i has η_x **then**
4. **Return** True;
5. **Else**
6. **Return** False.

2.4 元素的删除操作

算法 4 以删除元素 x 为例, 详细介绍了跳跃滤波的删除操作. 类似于查询操作, 跳跃滤波首先计算出 x 的指纹以及两个候选桶. 接着计算出 x 的候选 CF 块. 最后检查该 CF 块的两个目标候选桶中是否存在 x 的指纹: 若存在, 则从这两个桶中删除一个 x 指纹的副本, 并将 *JS* 的数值减去 1, 接着返回 True, 表示删除完成; 否则返回 False, 表示元素 x 不属于目标集合, 无法删除. 值得注意的是, 跳跃滤波同样不能删除未成功存储的元素, 否则可能由于哈希冲突导致假阴性错误. 举个例子, 跳跃滤波表示集合 S, 现有元素 $x\in S$ 和 $y\notin S$, 由于哈希冲突, 两者恰好具有相同的指纹和候选桶. 此时如果删除 y, 跳跃滤波会误删除掉 x. 这种情况下, 会导致后续查询元素 x 时, 跳跃滤波会返回 False. 这与 $x\in S$ 相违背, 产生了假阴性错误.

算法 4. 跳跃滤波删除元素 x.

输入: x, *JF*.

输出: True 或 False.

1. $\eta_x=h_0(x)$;
2. $h_1(x)$=*hash*(x), $h_2(x)=h_1(x)\oplus$*hash*(η_x), n=*length*(*JF*), $i=j(\eta_x,n)$;
3. **If** bucket $h_1(x)$ or $h_2(x)$ in CF_i has η_x **then**
4. Delete this η_x, *JS*––, **return** True;
5. **Else**

6. **Return** False.

2.5 容量缩减操作

当数据流大量离开时，跳跃滤波会因为元素的大量删除而导致空间利用率(指已使用的槽在所有槽中的占比)下降. 如图 3(b)所示，大量的存储槽处于未利用状态. 考虑到节省空间，此时应该及时回收未利用的空间. 算法 5 详细地介绍了跳跃滤波在符合压缩条件时，通过将列表中末端 CF 块中存储的指纹重新映射到其他 CF 块中，进而删除掉末端 CF 块，以此实现容量缩减操作. 具体来说，跳跃滤波首先获取当前存储元素的数量 JS 以及当前列表长度 n，接着计算 $JS/((n-1)mb)$，判断其是否高于空间利用率阈值 Th: 若是，则返回 False，表明跳跃滤波当前负载率较高，不能进行容量缩减；否则，跳跃滤波遍历 CF_{n-1} 中所有的指纹，例如 η，通过计算 $i=j(\eta,n-1)$确定 η的候选 CF 块，进而结合 η在 CF_{n-1}中的候选桶索引，将 η重新插入 CF_i中；如果发生插入失败,则重新插入被踢出的受害指纹，同时遍历 CF_{n-1}中此前已成功重新映射的指纹，将其从已插入的候选 CF 块(CF_0–CF_{n-1}之间)中删除，返回 False，表明跳跃滤波缩减容量失败，返回缩减操作前的状态. 如果 CF_{n-1}中所有指纹都成功迁移到 CF_0–CF_{n-1}之间的候选 CF 块，则从跳跃滤波的 CF 块列表中删除 CF_{n-1}，返回 True，表明容量缩减成功.

算法 5. 跳跃滤波缩减容量.

输入: 空间利用率阈值 Th, JF.

输出: True 或 False.

1. n=*length*(JF);
2. **If** $JS/((n-1)mb)>Th$ **then**
3. **Return** False;
4. **Else**
5. **For** η in CF_{n-1} **do**
6. **If** relocate η into CF_i ($i=j(\eta,n-1)$) fails **then**
7. Insert the kicked victims in Line 6, delete all η that has been relocated during Line 6, **return** False;
8. Delete CF_{n-1}, **return** True.

2.6 自适应跳跃滤波的思考

当数据流在短时间内大量达到或离开时，会形成数据突发现象，数据突发是网络领域、工程领域以及社交生活等场景中常见的现象[16,17]. 当发生数据突发时，如果跳跃滤波按照算法 2 和算法 5，每次仅增加或者删除一个 CF 块来实现容量的动态增减，那么将频繁引发指纹重新映射，造成巨大的计算和时间开销. 为了解决该问题，本文进一步提出了自适应跳跃滤波(adaptive jump filter, AJF). 具体来说，自适应跳跃滤波首先定义了一个描述数据流突发大小的系数，称为突发系数 θ，有 $\theta=\Delta n/\Delta t$，其中，Δn 为 Δt 时间段内到达的数据元素数量减去离开的元素数量. 此外，自适应跳跃滤波还定义了一个突发时间窗口 T，用于描述数据流突发现象一般的持续时长. Δt 和 T 的大小通常由经验决定. 基于上述认识，当需要扩增或缩减容量时，自适应跳跃滤波通过计算 $\lceil \Delta n\times T/(\Delta t\times m\times b)\rceil$来决定增删 CF 块的数量，其数值的正负决定增删类型. 通过上述设计，自适应跳跃滤波适应性地为突发数据流表示提供合适粒度的空间缩放，从而避免了大量不必要的指纹重新映射操作，降低计算和时间开销. 然而，该优化设计会不可避免地增加潜在的空间开销，这是时间开销和空间开销权衡的结果.

3 理论分析

本节从插入失败期望、假阳性率和空间代价这 3 个方面展开对本文提出的动态摘要技术，即跳跃滤波的理论分析. 标准 Cuckoo 滤波利用部分键值 cuckoo 哈希策略将元素的指纹存储在哈希表中，该方法根据给定元素的当前位置和指纹推导出元素的候选桶. 这种设计下，指纹的长度直接影响数据结构的插入失败期望和假阳性率. 基于此认识，本文接下来探索跳跃滤波所需的最小指纹长度，并在此基础上分析其空间效率.

3.1 元素插入失败的期望

这里的插入失败期望, 是指容量一定的数据结构在存储某数据集的过程中, 发生插入失败次数的期望. 如果这个期望大于等于 1, 那么数据结构在存储过程中必定会发生插入失败. Cuckoo 滤波插入失败期望的大小决定其空间利用率的高低. 插入失败期望越大, Cuckoo 滤波的构建过程越容易达到指纹重分配次数的阈值, 其空间利用率也就越低. 因此, 这里通过插入失败期望来分析跳跃滤波为达到较高空间利用率所需的指纹长度下界.

两个不同的元素在跳跃滤波中因为哈希冲突而产生碰撞, 这种现象的发生需要满足以下 3 个条件: (1) 具有相同的指纹, 此情况发生的概率为 $1/2^f$(其中, f 为跳跃滤波采用的指纹长度); (2) 具有相同的候选桶, 此情况发生的概率是 $2/m$; (3) 具有相同的候选 CF 块, 此情况发生的概率为 $1/n$. 注意: 由于跳跃一致性哈希杰出的均匀性, 每个元素被映射到不同 CF 块中的概率是相同的, 因此有上述条件(3)成立. 实际上, 由于跳跃滤波所采用的跳跃一致性哈希算法以元素的指纹作为随机数种子, 因此一旦指纹确定后, 通过跳跃一致性哈希所计算的候选 CF 块也就唯一确定了. 也就是说, 上述条件(1)成立, 必定有条件(3)成立. 基于上述认识, 一个包含 q 个元素的集合共享相同的两个候选桶的概率是$(1/2^f\times 2/m)^{q-1}$. 假设跳跃滤波需要插入 N 个随机元素, 其中每个 CF 块包含 $m=\gamma N/n$ 个桶(γ为一个常数), 那么只要任一 CF 块中有 $q=2b+1$ 个元素映射到相同的两个候选桶中,就会引发插入失败. 基于这个认识, 可以推导出跳跃滤波插入失败期望的下界. 根据条件(3)可知, 每个 CF 块会映射到 N/n 个元素, 因此, 跳跃滤波构建阶段中, 某 CF 块有 $2b+1$ 个元素发生碰撞次数的期望为:

$$\binom{N/n}{2b+1}\left(\frac{2}{2^f\times m}\right)^{2b}=\binom{N/n}{2b+1}\left(\frac{2\times n}{2^f\times\gamma\times N}\right)^{2b}=\Omega\left(\frac{N}{n\times 4^{bf}}\right) \tag{1}$$

从上述公式可以得出结论: 为避免插入失败, 4^{bf} 的大小至少是$\Omega(N/n)$; 否则, 碰撞次数的期望是$\Omega(1)$. 可以看出, 跳跃滤波可以通过两个途径来提升插入成功的概率: 一是增加每个候选桶中槽的数量 b; 二是增加指纹长度 f. 在 b 确定的情况下, 从避免较大概率插入失败的角度出发, 跳跃滤波所采用指纹的最小长度必须为:

$$F=\lceil(\log_2\Omega(N/n))/2b\rceil \tag{2}$$

和 Cuckoo 滤波一样, 跳跃滤波在实际应用中, 可以通过增加上式分母中的 b 因子来避免插入失败, 而不必维持较长的指纹, 实现节省空间开销的目的. 换句话说, 只要使用大小合适的桶, 就可以通过较短的指纹避免显著的插入失败, 从而实现较高的空间利用率. Cuckoo 滤波已经证实, 当每个桶存储 4 个 6 位或更长的指纹时, 一个表示 40 亿个元素的 CF 可以有效地将其哈希表填满 95%[6].

3.2 假阳性率

假阳性率是集合成员检测的重要参数之一. 假设跳跃滤波总共维持 n 个 CF 块, 其中单个 CF 块维持 m 个桶, 每个桶维持 b 个槽, 且其假阳性率为ξ_{CF}. 当跳跃滤波查询某元素 x 时, 假设跳跃滤波已存储的某元素 y 具有和 x 相同的指纹(这种情况发生的概率为 $1/2^f$), 那么只要 y 的第 1 个候选桶和 x 的任一个候选桶相同(这种情况发生的概率为 $2/m$), 就会产生哈希碰撞, 导致假阳性情况发生. 而跳跃滤波所能存储的元素数量上界为 $n\times m\times b$, 因此, 跳跃滤波产生假阳性错误的概率上界为:

$$\xi_{JF}=nmb\times(1/2^f)\times(2/m)=2nb/2^f=n\xi_{CF} \tag{3}$$

通过上式注意到, 虽然跳跃滤波和压缩型 LDCF 实现了查询单个 CF 块的目标, 在执行查询操作的过程中要比 DCF 遍历更少的指纹对象, 但是二者的假阳性率大小却和 DCF 一样. 这是指纹值和候选 CF 块索引信息复用的结果. 具体来说, 导致摘要技术型数据结构的假阳性率大小的根本原因是哈希冲突概率的高低, 而不是遍历比较的指纹对象数量的多少. 跳跃滤波和压缩型 LDCF 都通过指纹值来定位元素唯一的候选 CF 块, 因此, 发生哈希冲突的概率并不会因为遍历对象数量的变少而降低. 根据公式(3), 给定目标最小假阳性率后, 可以推导出跳跃滤波所需的最短指纹长度为:

$$f=\lceil\log_2(2bn/\xi_{JF})\rceil \tag{4}$$

3.3 空间代价

在获得符合插入失败期望和假阳性率要求的最小指纹长度后，跳跃滤波的空间效率可以通过表示每个元素所需的平均比特数来度量. 由于跳跃一致性哈希算法的均匀性，跳跃滤波和其中单个CF块的空间效率是一致的. 因此，接下来推导单个 CF 块的空间效率，以反映跳跃滤波的整体情况.

假设单个 CF 块维护 m 个桶，每个桶都有 b 个槽，并使用 f 位的指纹，那么该 CF 块的尺寸是 mbf，其单位为比特数. 在 CF 块插入多个随机元素达到负载系数α之后，共有 $mb\alpha$个元素被成功存储. 此时，单个 CF 块，即跳跃滤波表示每个元素所需的平均比特数(bits per element, BPE)是:

$$BPE=mbf/(mb\alpha)=f/\alpha \tag{5}$$

平均比特数反映了跳跃滤波表示元素的平摊空间代价. 上式中，f 的值不应小于公式(2)和(4)共同确定的下界. 然而在实际应用中，由于较短的指纹足以满足较高的空间利用率，因此，f 的大小往往由较苛刻的假阳性率要求所决定，也就是公式(4)计算的下界.

3.4 二次扩容概率

本文注意到，在上述第 2.2 节描述的插入元素 x 的过程中，跳跃滤波存在扩容后因为重新映射失败而需要再次扩容的现象，称为二次扩容. 实际上，此现象仅发生在列表中 CF 块数量足够多的时候，此时新增的 CF 块可能由于重新映射的指纹相对较多而接近负载系数，从而可能会造成插入失败. 造成该现象的原因主要是因为每个 CF 块输入的数据元素有差别，此外，CF 采用随机重分配策略来提高空间利用率，这些原因导致 CF 块的负载系数并非恒定不变，而是围绕某数值有小范围波动，例如 95%. 因此，尽管跳跃一致性哈希算法有着出色的均匀性，可以保证所有 CF 块映射到的元素数量几乎相同，但由于 CF 本身的设计原因，跳跃滤波中的同构 CF 块也不能维持完全相同的负载系数. 所以，当 CF 块数量足够多时，新增 CF 块可能会发生插入失败的现象.

接下来分析发生二次扩容的概率. 根据上述认识，每个同构 CF 块的负载系数服从 $N(u,\sigma^2)$上的正态分布，其数学期望为 u，方差为σ^2，概率密度函数为 $f(x)$. 则跳跃滤波的整体负载系数为 u. 假设跳跃滤波当前维持 n 个 CF 块，当插入失败发生时，根据跳跃一致性哈希的均匀性，可认为此时每个 CF 块都达到或者接近其负载系数. 此时，跳跃滤波中存储的元素数量为 $nmbu$，需要迁移到新增 CF 块的元素数量为 $nmbu/(n+1)$. 这意味着迁移后新增 CF 块，即 CF_n 的空间利用率为:

$$SE_{CF_n}=\frac{nmbu}{mb(n+1)}=\frac{nu}{n+1}=u-\frac{u}{n+1} \tag{6}$$

如果公式(6)推导出的 CF_n 的空间利用率 SE_{CF_n} 超过了 CF_n 的负载系数 α_{CF_n}，则会发生指纹重新映射失败而导致二次扩容. 此时，根据 CF 块负载系数服从正态分布的特性，可计算出二次扩容发生的近似概率为:

$$P=\int_{-\infty}^{SE_{CF_n}} f(x)\mathrm{d}x=\int_{-\infty}^{nu/(n+1)} \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(x-u)^2}{2\sigma^2}\right)\mathrm{d}x \tag{7}$$

通过上述理论分析结果可观察出：跳跃滤波是否发生二次扩容，取决于CF块负载系数的正态分布参数以及 CF 块数量 n. 接下来就是如何确定其中 u 和σ^2 数值的问题. 实际上，CF 的负载系数和众多因素有关，例如哈希函数种类、指纹长度,桶的大小、指纹重分配次数阈值等. 此外，指纹重分配采取随机踢出重放的策略,因此负载系数还呈现随机波动的特征. 因此，无法通过理论分析来计算负载系数正态分布参数的 u 和σ^2 具体数值，只能是经验主义地依靠实验结果来判断. 本文在实验部分对公式(7)中的二次扩容概率进行了进一步的验证. 另外值得注意的是，自适应跳跃滤波根据实时数据流的大小来调整所需增加的 CF 块数量，是降低二次扩容概率的有效措施.

3.5 集合成员检测的计算开销

已知单个 BF 和 CF 执行查询操作的计算开销是常数级别的. 假设所有动态数据结构维护 n 个 CBF 块或 CF 块. 考虑执行一次集合成员检测任务的计算开销，例如查询元素 x 是否存在集合 S 中. DBF, DCF 以及 CCF

都需要检查 n 个基本 BF 或者 CF 块, 因此其计算开销为 O(n). 压缩型 LDCF 虽然将需要检查的 CF 块数量降到了 1, 但是需要查找二叉树结构来检索 x 的候选 CF 块, 因此其计算开销为 O($\log_2 n$). 跳跃滤波同样实现了仅查询单个 CF 块, 并且通过一次哈希计算来索引 x 的候选 CF 块, 因此其计算开销为 O(1), 真正意义上实现了动态数据结构的常数级查询开销.

4 实验分析

本节将展开综合的实验来全面评估跳跃滤波方案. 实验主要包括两部分: 第 1 部分是测试各种动态数据摘要技术的单一指标性能, 例如插入、查询、删除吞吐量及相应的空间开销; 第 2 部分是测试动态数据摘要技术在表示真实动态数据流情况下的综合性能, 例如构建的整体时间和空间开销. 由于 CCF 计算复杂导致极其昂贵的时间开销, 并且 CCF 的设计初衷是为了实现桶级别的细粒度容量调整, 除了基本数据结构块外, CCF 和 DCF 在设计上并无不同. 综合考虑, 本文没有将 CCF 作为对比方法. 此外, 需要注意的是, 实验部分中提及的 LDCF 算法都是其压缩变种, 即压缩型 LDCF.

4.1 实验配置

- 平台

所有的实验都是在配备 Intel Core i7 处理器和 16 GB DRAM 的机器上进行的. 本文用 Python 实现了跳跃滤波及其对比方法, 包括 DBF、DCF 以及 LDCF, 相关代码见 Github 网站[36].

- 数据集

由于所有数据结构都使用哈希函数直接映射元素, 元素内容对数据结构的性能没有影响, 因此本文使用合成的数据集来测试单一指标性能. 对于综合性能测试, 本文使用了两个真实数据集.

1) 网络流量数据集. 该数据集来自 WIDE[37]采集的网络流量信息, 本文以五元组信息作为元素内容, 从中截取了一段持续 84 s、包含 109 万条网络流量的子集用于测试.
2) 网络结构数据集. 该数据集来自 SNAP[38], 记录了用户在 Stack Overflow 网站上的交互信息. 每个数据有 3 个值 z, v, t, 表示用户 z 在时刻 t 答复或者评论了用户 v 的帖子. 文中以用户 z 和 v 作为元素内容, 从中截取了一段持续两年、包含 1 800 万条交互记录的子集用于测试.

- 对比方法

实验中采用的对比方法主要包括两类.

- 第 1 类是链状结构的动态数据摘要技术, 包括动态 Bloom 滤波(DBF)和动态 Cuckoo 滤波(DCF). 这类动态数据摘要技术通过维护多个 CBF 块或 CF 块构成的链表来实现随数据集大小而相应增减的弹性容量. 此外, 由于 CBF 将 Bloom 滤波中的比特位替换为具有多个比特的计数器以支持元素的删除, 因此动态 Bloom 滤波的空间开销要远高于动态 Cuckoo 滤波.
- 第 2 类是树状结构的动态数据摘要技术, 包括对数动态 Cuckoo 滤波(LDCF)和其压缩型变种. 该类方法将多个 CF 块维护成树状结构, 并根据元素指纹的前缀比特位来索引该元素在树结构中每个层级上的候选 CF 块, 大幅度减少了集合成员检测所需检查的 CF 块数量.

由于压缩型变种更为优异, 因此本文实验选取了压缩型对数动态 Cuckoo 滤波作为对比方法. 值得注意的是, 随着数据集规模的增大, 链状结构的动态数据摘要技术用于集合成员检测的时间开销呈线性增长, 数据结构所占的空间开销同样呈线性增长; 然而树状结构的动态数据摘要技术用于集合成员检测的时间开销呈对数增长, 数据结构所占的空间开销呈指数增长.

- 参数设置

实验中, 所有动态数据结构都维持相同存储空间大小的 CBF 块或 CF 块, 并且维持相同的整体假阳性率 ξ=0.001. 具体来说, DCF、LDCF 和 JF 维持的同构 CF 块的参数配置为: m=1024, b=4, MAX=50. DCF 和 LDCF 使用 3 个哈希函数, JF 使用 4 个哈希函数. DBF 维持的同构 CBF 块的参数配置为: m=1024×f, k=8, 其中, f 为 DCF 使用的指纹长度, 每个计数器的尺寸设置为 4 个比特. 上述配置使得所有动态数据结构维持相同存储空

间大小的 CBF 块或者 CF 块，且其初始化数量都为 1，确保动态数据结构以相同大小的粒度调整容量.

- 指标

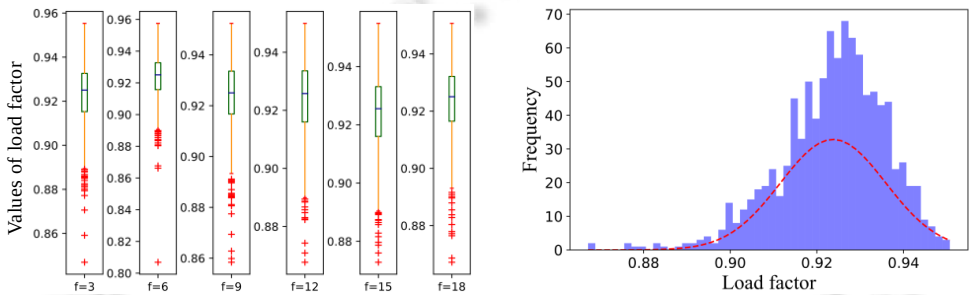
实验中主要使用了 3 个指标.

1) 吞吐量. 吞吐量反映了数据结构单位时间内执行操作的次数，包括插入、查询以及删除操作;
2) 空间开销. 空间开销反映了数据结构的存储空间大小，实验中以比特为单位;
3) 时间开销. 时间开销反映了数据结构执行操作所需花费的时间，实验中以 s 为单位.

4.2 单一性能测试结果

4.2.1 跳跃滤波的负载系数与产生二次扩容的概率

该节针对实验中所使用的单个 CF 块展开了负载系数的研究. 具体来说，研究实验中对配置为: m=1024, b=4, MAX=50 的 CF，执行了 1 000 次插入实验. 记录其在指纹长度分别为 3, 6, 9, 12, 15, 18 情况下，第 1 次发生插入失败时的空间利用率，以此来模拟 CF 的负载系数. 图 4(a)绘制了 CF 在不同指纹长度下负载系数的箱形图，说明实验中所使用的 CF 配置基本上能达到 93%左右的负载系数. 图 4(b)以 f=18 为例，进一步研究了负载系数的频数分布情况，并拟合其正态分布的密度函数曲线，该结果证实了 CF 的负载系数符合正态分布.



(a) 不同指纹长度下的负载系数统计　　(b) 负载系数的频数分布

图 4　跳跃滤波中单个 CF 块的负载系数

根据图 4(b)中实验结果拟合出的正态分布参数，即 u=0.9237, σ=0.0122，结合公式(7)即可推导出在使用上述配置的 CF 情况下，跳跃滤波发生二次扩容的近似概率，见表 3. 可以看出，只有当 CF 块数量增加到 60 左右时，跳跃滤波此时才会存在较为显著的概率发生二次扩容. 因此在实际应用中，跳跃滤波因为二次扩容而增加的计算开销可以忽略不计.

表 3　跳跃滤波发生二次扩容的近似概率

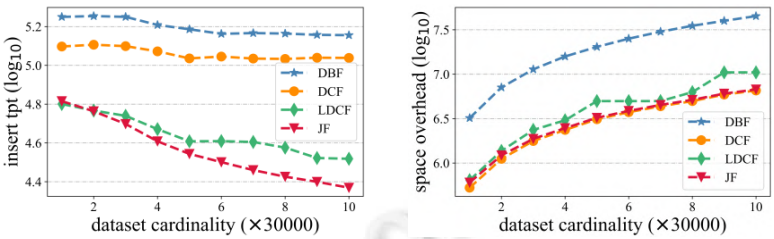
n	10	20	30	40	50	60	70	80	90	100
P	0	0	0.007	0.032	0.068	0.107	0.143	0.174	0.202	0.226

4.2.2 元素插入性能与容量扩增性能

这里，从插入吞吐量和空间开销两方面来评价动态数据摘要技术在存储方面的性能. 如图 5(a)和图 5(b)所示，将输入数据集的基数大小从 30 000 增加到 300 000，记录各动态数据摘要技术在不同数据集基数大小下的整体插入吞吐量和空间开销. 注意，图中结果以对数形式展现. 从图 5(a)可以看出，JF 和 LDCF 的插入速度不及 DBF 和 DCF，而且随着输入数据集基数的增大呈现下降趋势; 而 DBF 和 DCF 却能保持较稳定的高插入速度. 这是因为无论维持多少 CBF 块或 CF 块，DBF 和 DCF 总是仅在最后一个 CF 块中尝试插入需要存储的元素. 然而随着输入数据集基数增大, LDCF 和 JF 需要维持更多的 CF 块，同时也需要遍历和迁移更多已存储的数据，这些数据重新映射操作导致 LDCF 和 JF 的插入吞吐量低于 DBF 和 DCF.

从图 5(b)可以看出, JF 和 DCF 一样，实现了空间开销随数据集基数呈线性增长，同时空间开销最少. DBF 虽然也实现了线性增长，但 CBF 块和 CF 块相比，在相同假阳性下具有更大的空间开销，且 DBF 为了实现支持删除，将 BF 中每个比特替换成了 4 个比特位大小的计数器，这些设计不可避免地导致 DBF 需要昂贵的空

间开销. LDCF 采用二叉树结构来维护 CF 块, 因此其并没有实现空间开销的线性增长. 如图 5(b)所示, LDCF 在当数据集基数在某范围内增加时, 例如当横轴取值范围为[1,5]和[7,9], 其空间开销在对数方式的计量下呈线性增长, 实际上呈指数增长, 此时, LDCF 处于树结构深度增加 1 的过程. 当该过程结束后, LDCF 中 CF 块的数量翻倍, 空间利用率减半, 在这些 CF 块达到负载系数之前, LDCF 的空间开销会维持不变, 例如当横轴取值范围为[5,7]和[9,10]. 总的来说, 相比于 DBF 和 LDCF, JF 在插入过程中能节省 8.1%–85.3%, 6.1%–42.3%的空间开销.



(a) 元素插入吞吐量 (b) 存储空间开销(比特)

图 5 不同输入数据集基数下动态数据摘要技术的存储性能比较

4.2.3 元素查询性能

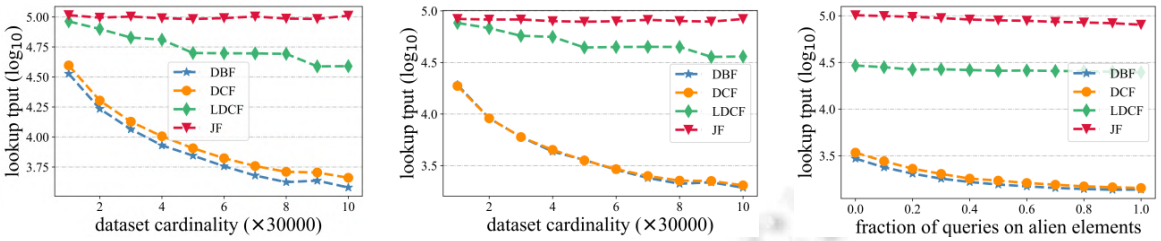
本实验进一步量化了动态数据摘要技术处理不同类型查询的查询吞吐量, 实验结果如图 6 所示. 通过改变输入数据集基数的大小, 并构造了两个查询数据集, 其中一个所含元素都已被存储, 另一个所含元素都未被存储, 称为外来元素, 以此来测试这些动态数据摘要技术在不同容量下的积极查询(查询对象为 100%存在的元素)和消极查询(查询对象为 100%不存在的元素)的吞吐量, 分别如图 6(a)和图 6(b)所示, 其中结果以对数形式展示. 随着动态数据摘要技术容量不断增长, 无论是积极查询还是消极查询, DBF 和 DCF 的查询吞吐量都随之线性下降, 且由于需要遍历所有的 CF 块, DBF 和 DCF 有着很低的查询效率. LDCF 的查询吞吐量整体上呈下降趋势, 但当其容量增长到某些阶段时, 例如横轴取值范围为[3,4]和[5,8]时, 其查询吞吐量不变, 其余阶段呈对数级别下降的趋势. 这是因为 LDCF 在树结构深度不变时, 遍历树结构的计算复杂度是恒定的, 加上仅检查单个 CF 块, 这就是为什么 LDCF 在某些阶段查询吞吐量维持恒定, 不随所表示的数据量多少而变化. 一旦数结构深度增大, LDCF 遍历树结构的时间开销也会随之增加, 导致其查询吞吐量呈对数级别下降. 相比之下, JF 仅执行一次额外的哈希计算和检查单个 CF 块来响应查询任务, 因此总是能够维持快速恒定的查询响应速度, 不受数据结构表示元素数量的影响, 真正实现常数级查询开销.

通过比较图 6(a)和图 6(b), 可以发现积极查询的吞吐量要高于消极查询的吞吐量. 实际上, 查询类型对动态数据结构的影响是两方面的.

- 第一是对单个数据结构的影响: 对于单个 CF 来说, 在检测到符合目标指纹后会马上停止检测候选桶剩下的指纹, 因此 CF 检测存在的元素要比外来元素响应更快. 对于单个 BF 来说, 在检测到数值为 0 的比特位后, 会马上停止检测剩下的候选比特位, 因此 BF 检测外来的元素要比存在的元素响应更快.
- 第二是对动态数据结构整体的影响: 对于 DBF 和 DCF 来说, 无论其维持的动态列表是由 BF 还是 CF 构成, 在检测到符合目标指纹后都会马上停止搜寻剩下的 CBF 块或 CF 块, 因此 DBF 和 DCF 检测已存储元素的响应速度更快、吞吐量更高.

为了探究查询数据集中外来元素占比对查询性能的影响, 实验中固定输入数据集的大小为 300 000, 构建了一个同时包含外来元素和存在元素的混合查询数据集, 并将其中外来元素的占比从 0%逐渐增加到 100%, 以此来测试动态数据摘要技术在不同外来元素占比下的混合查询吞吐量, 如图 6(c)所示. 结果表明, 随着外来元素比重的不断增加, 所有动态数据摘要技术的混合查询吞吐量都随之下降, 其下降幅度以 DBF 和 DCF 尤为明显. 这是因为 LDCF 和 JF 仅查询一个 CF 块, 因此其受查询类型的影响仅限上述第 1 方面; 而 DBF 和 DCF 同时受到上述两方面的影响, 且随着容量增大, 第 2 方面影响的权重也逐渐增大.

总的来说，相比于 DBF、DCF 和 LDCF, JF 的积极查询吞吐量为 3.08–26.99, 2.62–22.39 和 1.12–2.64 倍，消极查询吞吐量为 4.36–43.24, 4.47–41.04 和 1.09–2.31 倍，整体混合查询吞吐量为 51.32, 46.25 和 3.44 倍.

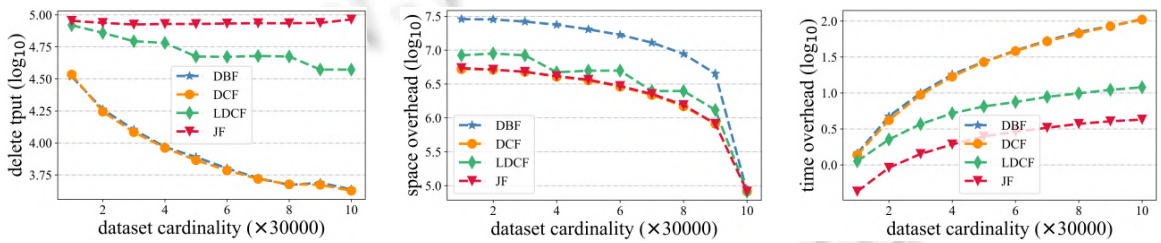


(a) 积极查询(所有查询对象存在) (b) 消极查询(所有查询对象不存在) (c) 混合查询(部分查询对象存在)

图 6 动态数据摘要技术的查询吞吐量比较

4.2.4 元素删除性能与容量缩减性能

该实验评价动态数据摘要技术在删除元素和缩减容量方面的性能. 首先调整输入数据集的大小，将各动态数据摘要技术填充到不同级别的容量；然后删除掉各动态数据摘要技术中所有已存储的元素，记录其在不同容量下的删除吞吐量，结果如图 7(a)所示. 删除吞吐量的测试结果和图 6(a)中积极查询的结果相似，这是因为删除操作的实质，就是在积极查询操作基础上额外执行一次清空匹配指纹的操作. 除了优异的查询性能外，JF 也能维持高速恒定的删除响应速度，而与所表示的数据集大小无关，实现真正常数级别的删除开销.



(a) 删除吞吐量 (b) 空间开销(单位：比特，阈值: 0.8) (c) 时间开销(s)

图 7 动态数据摘要技术的删除与容量缩减性能比较

其次，实验中进一步测试了动态数据摘要技术回收闲置空间的性能. 首先，为各动态数据摘要技术设定回收空间的条件. 具体来说，对于 JF，将空间回收算法中的空间利用率阈值 *Th* 设置为 0.8, DCF 同样满足该设定. 对于 DBF，若空间利用率最低的两个 CBF 块合并后仍不超过额定假阳性阈值，则认为其满足容量缩减的条件. 对于 LDCF，若空间利用率最低且具有同一父节点的两个子 CF 块，其空间利用率之和不超过 0.8，则认为 LDCF 满足容量缩减的条件. 接着，将输入数据集的基数大小固定为 300 000，从已存储的元素中随机挑选一部分元素作为删除数据集，删除数据集的基数大小从 30 000 不断增长到 300 000. 在删除元素过程中，一旦动态数据摘要技术符合空间回收条件，则尝试回收空闲存储单元格，缩减容量. 最后，记录各动态数据摘要技术删除不同大小的删除数据集后所剩的空间开销以及期间所花费的总时间开销，如图 7(b)和图 7(c)所示. 空间开销方面，JF 和 DCF 总是使用最少的空间来表示元素. 此外，随着数据结构表示元素数量的减少，DBF、DCF 和 LDCF 总是能及时回收闲置的空间. 然而, LDCF 在某些阶段，比如横轴取值范围为[1,3], [4,6], [7,8]时，空间开销维持不变，未能及时回收因元素删除后而空闲的空间资源. 造成该现象的根本原因是因为 LDCF 采用二叉树结构维护 CF 块，空间开销呈指数增长，在表示一定数量的元素时，其空间利用率低，空间效率落后. 举个例子，为了表示 8.5 个 CF 块容量大小的数据集，LDCF 需要深度为 4 的树结构，即 16 个 CF 块来表示该数据集，而空间开销随数据集基数线性增长的 DBF、DCF 和 LDCF，则只需要 9 个 CF 块或 CBF 块. 时间开销方面，JF 也总是消耗最少的时间. 且随着删除数据集基数的增大以及需要回收的空间资源增多，所有动态数据摘要技术的时间开销也不断增大.

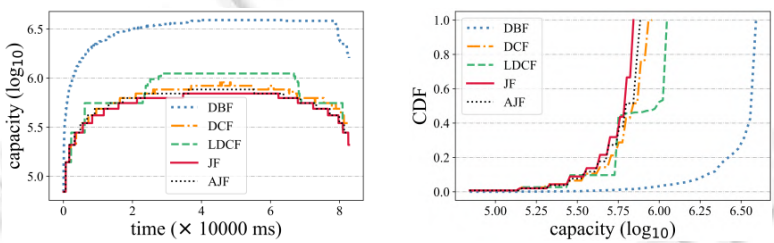
总的来说, 相比于 DBF、DCF 和 LDCF, JF 的删除吞吐量为 2.73–21.22, 2.63–21.75 和 1.08–2.47 倍. 在动态数据摘要技术缩减容量方面, 相比于 DBF 和 LDCF, JF 能及时多回收 81.1%–82.6%和 9.4%–42.8%的空间资源; 相比于 DBF、DCF 和 LDCF, 在删除元素及回收空间过程中, JF 能够节省 70.6%–96.0%, 68.9%–95.9%和 59.6%–64.4%的时间开销.

4.3 综合性能测试结果

为了探究各动态数据摘要技术在真实场景中的实际应用效果, 这里使用两类典型的真实数据集对包括自适应跳跃滤波在内的各动态数据摘要技术展开了测试, 即 WIDE 网络流量数据集和 SNAP 网络结构数据集, 这两个数据集是互联网大数据、社交网络大数据和社会公共领域大数据的典型代表.

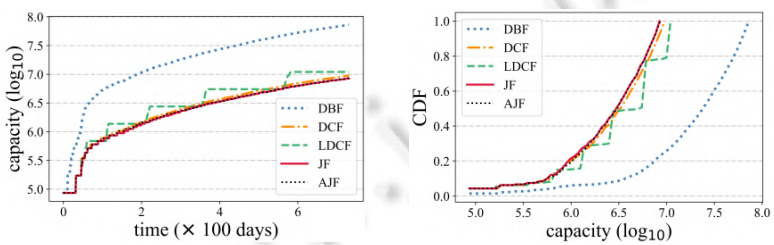
4.3.1 空间开销

实验首先使用动态数据摘要技术表示 WIDE 网络流量数据集, 按时间顺序(以 ms 为单位)插入每毫秒内到来的网络流信息, 同时删除掉该时刻离开的网络流信息, 记录下动态数据摘要技术每毫秒的空间开销, 结果如图 8 所示; 其次, 使用动态数据摘要技术表示 SNAP 网络结构数据集, 同样按时间顺序(以天为单位)插入每天发生的用户交互数据; 接着, 随机挑选当天 80%的数据作为删除数据集, 对动态数据摘要技术执行删除操作, 记录下动态数据摘要技术每天的空间开销, 结果如图 9 所示. 注意: 在上述两个综合实验中, 设置动态数据摘要技术的容量缩减条件与第 4.2.4 节一致, 一旦符合容量缩减条件, 则尝试对动态数据摘要技术进行闲置空间回收.



(a) 空间开销(单位: 比特, 阈值: 0.8) (b) 空间开销累积频率分布图

图 8 WIDE 网络流量数据集测试下动态数据摘要技术的空间开销对比



(a) 空间开销(单位: 比特, 阈值: 0.8) (b) 空间开销累积频率分布图

图 9 SNAP 网络结构数据集测试下动态数据摘要技术的空间开销对比

图 8(a)和图 9(a)以对数形式展示了各动态数据摘要技术在 WIDE 数据集以及 SNAP 数据集测试下的空间开销情况. 无论是表示网络流量还是网络结构数据, JF 总是使用最少的空间资源, 紧接着是 AJF, 其次是 DCF. 这是因为 JF 和 AJF 使用的跳跃一致性哈希算法具有高度的均匀性, 可以将数据元素均匀映射到所有的 CF 块中, 使得所有 CF 块维持着几乎相同的空间利用率. 然而, DCF 总是将元素存储在列表中最后一个 CF 块之中, 这导致数据元素的存储位置和其达到的时序紧密相关. 因此, DCF 中的存储操作总是发生在最后一个 CF 块之中, 而删除操作总是发生在列表中前列的 CF 块中, 这直接导致 DCF 中部分 CF 块的空间利用率存在着显著差异. 这些差异造成了 DCF 在表示动态数据集时具有比 JF 更多的空间开销. AJF 在扩容时可能会一次性增加多个 CF 块以减少潜在的指纹重新映射操作, 其扩容粒度可能大于 JF, 因此也具有比 JF 稍大的空间开销. LDCF

的空间开销整体上要大于 JF 甚至是 DCF, 这是因为 LDCF 需要维持树状结构的 CF 块列表, 其空间开销呈指数级别增长. 该现象在图 9(a)中尤其明显, SNAP 数据集的基数随时间不断增加, 同时, LDCF 的空间开销曲线在对数尺度下每呈线性增长后就维持一段恒定水平, 这与第 4.2 节中单一指标性能测试的结果一致. 由于设计机制落后, DBF 的空间开销要远远高于其他动态数据摘要技术.

图 8(b)和图 9(b)分别绘制了各动态数据摘要技术在 WIDE 数据集以及 SNAP 数据集测试下, 空间开销对数结果的累积频率分布图, 从另一个角度展示了各动态数据摘要技术的空间开销对比情况. 可以看出, JF 总是能够以相同的空间开销来表示比其他动态数据摘要技术更多的元素, 或者以更少的空间资源来表示相同数量的元素.

总的来说, 相比于 DBF、LDCF、DCF 和 AJCF, 在表示 WIDE 网络流量数据集时, JF 整体上能够节省 83.0%, 31.1%, 14.3%和 6.5%的空间开销; 在表示 SNAP 网络结构数据集时, JF 整体上能够节省 87.9%, 28.5%, 9.5%和 0.7%的空间开销.

4.3.2 时间开销

除了测试各动态数据摘要技术表示真实动态数据集的空间开销性能以外, 本文还测试了各动态数据摘要技术存储、查询和删除真实数据集的综合时间开销. 具体来说, 在 4.3.1 节实验的基础上引入了查询操作, 使用所有插入数据作为元素内容, 并给定随机的时间戳来构造查询数据集, 按时间顺序对动态数据摘要技术执行查询操作, 以此模拟实际应用中动态数据摘要技术在表示动态数据期间所面临的查询任务. 最终记录下实验总共花费的时间, 结果见表 4.

表 4 各动态数据摘要技术表示真实动态数据集的整体时间开销(s)

	DBF	DCF	LDCF	AJF	JF
WIDE	381.7	336.07	78.94	62.42	62.46
SNAP	1 416.49	1 372.61	133.47	91.06	93.30

从表 4 的结果来看, DBF 和 DCF 处理真实动态数据集的整体时间开销要远高于其他 3 类动态数据摘要技术, 尽管 DBF 和 DCF 具有较高的插入吞吐量. 这说明在表示真实动态数据集的过程中, 尤其是当数据基数较大时, 查询操作和删除操作对整体时间开销的影响占据了主要地位. AJF 的时间开销要低于 JF, 这符合理论期望, 因为 AJF 基于数据流突发系数适应性地调整 CF 块数量, 避免了某些不必要且耗时的数据重新映射操作. 因此, AJF 能够提升 JF 处理动态数据集的速度, 但是提升程度和数据流突发大小有密切联系. 数据流突发系数越大, 提升程度越高; 否则, 越小. 通过对比分析 AJF 在 WIDE 和 SNAP 数据集下的提升程度, 可以验证该结论. 可以发现, 由于 SNAP 数据集具有比 WIDE 更大的突发系数, 因此对比 JF, AJF 在 SNAP 数据集下的提升程度也更加明显.

总的来说, 相比于 DBF、DCF 和 LDCF, 在处理 WIDE 网络流量数据集时, JF 整体上能够节省 83.6%, 81.4%和 20.9%的时间开销; 在处理 SNAP 网络结构数据集时, JF 整体上能够节省 93.4%, 93.2%和 30.1%的时间开销.

5 总 结

本文以实现随数据基数线性增长的空间开销和数据处理分析的常数级别时间开销为研究目标, 提出了面向大数据治理中动态大数据处理分析的跳跃滤波. 本文展示了跳跃滤波如何实现数据存储、查询和删除的具体过程, 进一步提出了可减少指纹重新映射次数的自适应变种. 跳跃滤波的核心是, 根据元素的指纹和 CF 块的数量, 通过跳跃一致性哈希计算出元素的候选 CF 块. 该设计将查询对象数量降到了 O(1), 同时也确保其使用的 CF 块数量随数据基数线性增长. 综合实验结果表明, 无论是单一性能还是综合性能测试结果, 除了插入吞吐量以外, 跳跃滤波在查询吞吐量、删除吞吐量、空间开销以及时间开销方面, 其性能都要优于其他动态数据摘要技术.

References:

[1] Soares S. Big Data Governance: An Emerging Imperative. Boise: MC Press, 2012. 3−286.

[2] LüWF, Zhe ng ZM, Tong YX, Zhang RS, Wei SY, Li WH. Intelligent system for distributed social governance based on big data. Ruan Jian Xue Bao/Journal of Software, 2022, 33(3): 931−949 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6455.htm [doi: 10.13328/j.cnki.jos.006455]

[3] Zhang MW, Huang JJ, Han L. Range-based multi-keyword searchable scheme with privacy protection in e-healthcare cloud systems. Ruan Jian Xue Bao/Journal of Software, 2021, 32(10): 3266−3282 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6086.htm [doi: 10.13328/j.cnki.jos.006086]

[4] Bloom BH. Space/Time trade-offs in Hash coding with allowable errors. Communications of the ACM, 1970, 13(7): 422−426.

[5] Bender MA, Farach-Colton M, Johnson R, Kraner R, Kuszmaul BC, Medjedovic D, Montes P, Shetty P, Spillane RP, Zadok Z. Don't thrash: How to cache your hash on flash. Proc. of the VLDB Endowment, 2012, 5(11): 1627−1637.

[6] Fan B, Andersen D, Kaminsky M, Mitzenmacher M. Cuckoo filter: Practically better than bloom. In: Proc. of the CoNEXT. 2014. 75−88.

[7] Ozisik AP, Andresen G, Levine BN, Tapp D, Bissias G, Katkuri S. Graphene: Efficient interactive set reconciliation applied to blockchain propagation. In: Proc. of the SIGCOMM. 2019. 303−317.

[8] Maggs BM, Sitaraman RK. Algorithmic nuggets in content delivery. ACM SIGCOMM Computer Communication Review, 2015, 45(3): 52−66.

[9] Li SS, Luo LL, Guo DK. Sketch for traffic measurement: design optimization application and implementation. arXiv:2012.07214, 2020.

[10] Yang T, Jiang J, Liu P, Huang Q, Gong JZ, Zhou Y, Miao R, Li XM, Uhlig S. Elastic sketch: Adaptive and fast network-wide measurements. In: Proc. of the SIGCOMM. 2018. 561−575.

[11] Mun JH, Lim H. New approach for efficient IP address lookup using a bloom filter in trie-based algorithms. IEEE Trans. on Computers, 2015, 65(5): 1558−1565.

[12] Druschel P, Rowstron A. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In: Proc. of the SOSP. 2001. 188−201.

[13] Luo LL, Guo DK, Zhao YW, Rottenstreich O, Ma RTB, Luo XS. MCFsyn: A multi-party set reconciliation protocol with the marked cuckoo filter. IEEE Trans. on Parallel and Distributed Systems, 2021, 32(11): 2705−2718.

[14] Liu LT, Shen YL, Yan YB, Yang T, Shahzad M, Cui B, Xie GG. Sf-Sketch: A two-stage sketch for data streams. IEEE Trans. on Parallel and Distributed Systems, 2020, 31(10): 2263−2276.

[15] Tong D, Prasanna VK. Sketch acceleration on FPGA and its applications in network anomaly detection. IEEE Trans. on Parallel and Distributed Systems, 2017, 29(4): 929−942.

[16] Zhong Z, Yan S, Li ZK, Tan DC, Yang T, Cui B. BurstSketch: Finding bursts in data streams. In: Proc. of the SIGMOD. 2021. 2375−2383.

[17] Paul D, Peng YQ, Li FF. Bursty event detection throughout histories. In: Proc. of the ICDE. 2019. 1370−1381.

[18] Yang T, Zhang HW, Li JY, Gong JZ, Uhlig S, Chen SG, Li XM. HeavyKeeper: An accurate algorithm for finding top-*k* elephant flows. IEEE/ACM Trans. on Networking, 2019, 27(5): 1845−1858.

[19] Guo DK, Wu J, Chen HH, Yuan Y, Luo XS. The dynamic Bloom filters. IEEE Trans. on Knowledge and Data Engineering, 2010, 22(1): 120−133.

[20] Guo DK, Li M. Set reconciliation via counting Bloom filters. IEEE Trans. on Knowledge and Data Engineering, 2013, 25(10): 2367−2380.

[21] Chen HH, LiaoY L, Jin H, Wu J. The dynamic cuckoo filter. In: Proc. of the ICNP. 2017. 1−10.

[22] Luo LL, Guo DK, Rottenstreich O, Ma RTB, Luo XS, Ren BB. The consistent cuckoo filter. In: Proc. of the INFOCOM. 2019. 712−720.

[23] Karger D, Lehman E, Leighton F, Panigrahy R, Levine M, Lewin D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In: Proc. of the STOC. 1997. 654−663.

[24] Zhang F, Chen HH, Jin H, Reviriego P. The logarithmic dynamic Cuckoo filter. In: Proc. of the ICDE. 2021. 948−959.

[25] Lamping J, Veach E. A fast, minimal memory, consistent hash algorithm. arXiv:1406.2294, 2014.

[26] Luo LL, Guo DK, Ma RTB, Rottenstreich O, Luo XS. Optimizing Bloom filter: Challenges, solutions, and comparisons. IEEE Communications Surveys & Tutorials, 2018, 21(2): 1912−1949.

[27] Tarkoma S, Rothenberg CE, Lagerspetz E. Theory and practice of Bloom filters for distributed systems. IEEE Communications Surveys & Tutorials, 2011, 14(1): 131−155.

[28] Geravand S, Ahmadi M. Bloom filter applications in network security: A state-of-the-art survey. Computer Networks, 2013, 57(18): 4047−4064.

[29] Broder A, Mitzenmacher M. Network applications of Bloom filters: A survey. Internet Mathematics, 2004, 1(4): 485−509.

[30] Fan B, Andersen DG, Kaminsky M. MemC3: Compact and concurrent MemCache with dumber caching and smarter Hashing. In: Proc. of the USENIX NSDI. 2013. 371−384.

[31] Reviriego P, Pontarelli S. Perfect Cuckoo filters. In: Proc. of the CoNEXT. 2021. 205−211.

[32] Mitzenmacher M, Pontarelli S, Reviriego P. Adaptive Cuckoo filters. In: Proc. of the ALENEX. 2018. 36−47.

[33] Eppstein D. Cuckoo filter: Simplification and analysis. In: Proc. of the SWAT, Vol.53. 2016. 8:1−8:12.

[34] Fu PT, Luo LL, Li SS, Guo DK, Cheng GY, Zhou Y. The vertical Cuckoo filters: A family of insertion-friendly sketches for online applications. In: Proc. of the ICDCS. 2021. 57−67.

[35] Wang MM, Zhou MX, Shi SQ, Qian C. Vacuum filters: More space-efficient and faster replacement for Bloom and Cuckoo filters. Proc. of the VLDB Endowment, 2019, 13(2): 197−210.

[36] https://github.com/fptjy/JumpFilter

[37] https://mawi.wide.ad.jp/

[38] http://snap.stanford.edu/data/

附中文参考文献:

[2] 吕卫锋, 郑志明, 童咏昕, 张瑞升, 魏淑越, 李卫华. 基于大数据的分布式社会治理智能系统. 软件学报, 2022, 33(3): 931−949. http://www.jos.org.cn/1000-9825/6455.htm [doi: 10.13328/j.cnki.jos.006455]

[3] 张明武, 黄嘉骏, 韩亮. 医疗大数据隐私保护多关键词范围搜索方案. 软件学报, 2021, 32(10): 3266−3282. http://www.jos.org.cn/1000-9825/6086.htm [doi: 10.13328/j.cnki.jos.006086]

符鹏涛(1998－), 男, 硕士, CCF 学生会员, 主要研究领域为数据摘要技术, 网络测量.

赵翔(1986－), 男, 博士, 教授, CCF 高级会员, 主要研究领域为知识图谱, 先进数据分析.

罗来龙(1991－), 男, 博士, 副研究员, CCF 高级会员, 主要研究领域为数据摘要技术, 分布式网络系统.

李尚森(1997－), 男, 博士生, 主要研究领域为网络测量, 软件定义网络, 数据摘要技术.

郭得科(1980－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为网络计算与系统, 分布式计算与系统, 网络空间安全,边缘计算, 软件定义网络, 移动计算, 网络大数据.

王怀民(1962－), 男, 博士, 教授, 博士生导师, 中国科学院院士, CCF 会士, 主要研究领域为分布计算, 软件技术, 云际计算, 群体智能.