软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(11):3576−3595 [doi: 10.13328/j.cnki.jos.006097]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向大数据流的分布式索引构建*

杨良怀[1], 卢晨曦[1], 范玉雷[1], 朱镇洋[1], 潘 建[2]

[1](浙江工业大学 计算机学院,浙江 杭州 310023)

[2](浙江工业大学 之江学院,浙江 绍兴 312030)

通讯作者: 范玉雷, E-mail: fyl815@zjut.edu.cn

摘 要: 大数据流的高效存储与索引是当今数据领域的一大难点.面向带有时间属性的数据流,根据其时间属性,将数据流划分为连续的时间窗口,提出了基于双层 B+树的分布式索引结构 WB-Index.下层 B+树索引基于窗口内流数据构建,索引构建过程结合基于排序的批量构建技术,进一步对时间窗口分片,将数据流接收、分片数据排序以及 B+树构建并行化,提高了构建性能.上层 B+树索引基于各时间窗口构建,结合时间窗口时间戳的递增性和无限性,提出了避免节点分裂的构建方法,减少了 B+树分裂移动开销,提高了空间利用率和更新效率.WB-Index 架构中,将流数据和索引分离,同时利用内存缓存尽可能多的双层 B+索引和热点数据来提高查询性能.理论和实验结果表明,该分布式索引架构能够支持高效的实时数据流写入以及流数据查询,能够很好地应用于具有时间属性的数据流场景.

关键词: 大数据;数据流;分布式索引;B+树

中图法分类号: TP311

中文引用格式: 杨良怀,卢晨曦,范玉雷,朱镇洋,潘建.面向大数据流的分布式索引构建.软件学报,2021,32(11):3576−3595. http://www.jos.org.cn/1000-9825/6097.htm

英文引用格式: Yang LH, Lu CX, Fan YL, Zhu ZY, Pan J. Distributed index construction for big data streams. Ruan Jian Xue Bao/Journal of Software, 2021,32(11):3576−3595 (in Chinese). http://www.jos.org.cn/1000-9825/6097.htm

Distributed Index Construction for Big Data Streams

YANG Liang-Huai[1], LU Chen-Xi[1], FAN Yu-Lei[1], ZHU Zhen-Yang[1], PAN Jian[2]

[1](School of Computer Science and Technology, Zhejiang University of Technology, Hangzhou 310023, China)

[2](Zhijiang College, Zhejiang University of Technology, Shaoxing 312030, China)

Abstract: Efficient storage and indexing of big data streams are challenging issues in the database field. By segmenting the temporal data stream into continuous time windows, a distributed master-slave index structure is proposed based on double-layer B+ tree called WB-Index. Lower B+ tree index is built on stream tuples in each time window. Upper B+ tree index is built on each successive time window. Lower B+ tree index is constructed by combining both batch loading and parallel sorting techniques. The core idea of the construction method is to slice the time window and isolate the parallelable operations from others in the time window. Sorting and data stream receiving between slices work in parallel, while the B+ tree skeleton (a B+ tree without value) construction for the time window and the merge-sorting operation are parallelized as well. These techniques effectively expedite the B+ tree construction. Due to the monotonous increasement of timestamps of time windows, a split-less method for upper B+ tree index construction is adopted to avoid the node splitting and memory movement overhead, and improve the space utilization and update efficiency. In WB-Index, data stream tuples and index are separated, and index and hotspot data are cached as much as possible to improve query efficiency. Finally, theoretic analysis and experiments have both demonstrated that WB-Index can support efficient real-time data stream writing and stream data querying.

Key words: big data; data stream; distributed index; B+ tree

* 基金项目: 国家重点研发计划(2020YFB1707700)
Foundation item: National Key Research and Development Program of China (2020YFB1707700)
收稿时间: 2019-10-29; 修改时间: 2019-12-25; 采用时间: 2020-02-21

© 中国科学院软件研究所 http://www.jos.org.cn

近年来,数据流在各个领域被广泛应用,如 IOT 场景下的感知数据、智慧交通场景下的的监测数据以及互联网应用中的用户行为数据等[1−3].数据流由无限个形如⟨*t*,*v*⟩的流元组构成,其中,*t* 为流元组产生的时间,*v* 为具体的数据值.与静态关系型数据不同,数据流有流速快、实时性强、规模无限、易失等特点[4].针对数据流规模无限的特点,研究者提出了时间窗口概念,通过限定起止时间来限定处理范围,将数据流无限的处理规模切分成小的处理模块.传统的数据库管理系统在存储稳定、有限的静态数据时性能良好,但在数据流场景下存在严重的性能瓶颈.

为了克服这个困难,早期的研究者设计出一系列的数据流管理系统,如 STREAM[5],Aurora[6],TelegraphCQ[7]等.近年来,分布式流计算平台大量涌现,像 Storm[8],Flink[9]等,能提供高吞吐的流式计算.上述数据流处理系统主要用于流式数据的聚合计算、近似计算、连续查询等,在存储层面只保存查询和计算结果,不存储完整数据流.数据流的深度分析、计算场景需要依赖数据流的实时存储,上述系统无法适用.

数据流场景下,存储系统需保证高效的存储性能.目前,有基于抽样存储[10,11]和利用内存加速存储的系统,如 TimesTen[12]、Hekaton[13]、SolidDB[14]以及基于 LSM[15]存储模型的 BigTable[16]等,海量流媒体索引[17]利用内存组织多媒体流数据索引.无限数据规模下,单机无法支撑海量数据存储,从而衍生出分布式存储系统,如 GFS[18]、Cassandra[19]等.数据流实时存储的一个关键是在不影响写入性能的前提下快速实时地构建索引,以实现高效查询.目前,基于数据流实时存储的索引构建主要集中于位图索引构建方法[20,21],但均缺乏通用性.大数据流场景下,索引的更新频率高、存储开销大,具有很大的挑战.研究者根据数据存储形式,结合分布式思想,将索引结构按照一定的组织方式分层、分片维护,但是目前索引结构并未考虑数据流连续、实时的特点.

本文根据现有工作的不足,面向带有时间属性的数据流,提出一种基于 B+树的双层分布式索引,在保证存储性能的同时,支持高效查询,适用于体量大、维度多、速度快(数百万条/s)的大数据流场景.

本文贡献:

- 提出一种适用于数据流场景的分布式索引及其架构.分布式索引分为上下两层 B+树索引,由控制节点、协调节点、存储节点、查询节点、构建节点共同维护,将对流数据的索引、查询、存储分离,在有效地满足数据流特性的同时,也保证了存储和查询的性能;同时,提出了查询节点负载均衡、上层索引冗余机制和三层缓存策略,以保证系统的可用性和稳定性.
- 针对所提分布式索引,提出一种高效的构建方法.对于每个时间窗口数据流构建的下层 B+树索引,采用基于并行排序的批量装载技术进行构建;针对各个时间窗口构建的上层 B+树索引,采取不分裂的构建方法进行构建.上层和下层构建方法的结合,使得分布式索引构建速度快、时延低;
- 通过理论、实验的分析对比,验证分布式索引在数据流场景下的有效性.

1 相关工作

早期的数据流管理系统,如 OpenCQ[22]、STREAM[5]、Aurora[6]、NiagaraCQ[23]、TelegraphCQ[7]等,都属于集中式架构,通过合理的资源分配、调度和并行化设计,在有限的资源下提供连续查询、近似计算.随着数据流的广泛应用,近些年涌现出一批分布式流计算平台,如 Storm[8],Flink[9],Spark Streaming[24]等,将数据流计算抽象成 DAG[25]计算模型,定义多样的算子来进行实时计算,并支持机器学习、图计算等迭代计算场景.总体而言,上述数据流管理平台重点关注于查询优化、实时计算、资源管理等方面,缺乏数据流实时存储能力.

海量数据场景下,集中式存储受限于存储空间,数据存储量和更新性能存在瓶颈.研究者设计了分布式存储系统解决相应问题.GFS[18]是典型的主从分布式存储系统,将数据以块为最小单元存储在不同的从节点上,同时生成多个副本保证可用性.主节点管理数据块元数据,通过数据校验、版本控制等方式保证数据的完整性和一致性.GFS 数据写入速度慢,适用于离线数据存储.Bigtable 是基于 GFS 的分布式 KV 存储系统,底层借鉴 LSM 存储模型[15],将新数据直接写入到内存中,内存存储量达到阈值后再写入磁盘,从而大幅提高数据写入速度.基于分布式 KV 存储系统 HBase 的时序数据库 OpenTSDB[26],将时间戳和元组关键字等进行组装作为 *key* 值,但是查询时只能按照前缀匹配进行查询,若前缀值是一个范围,搜索只能遍历查找.实时 OLAP 系统 Druid[27]采用列

© 中国科学院软件研究所 http://www.jos.org.cn

式压缩存储,根据定义的维度,实时预聚合数据.在减少数据存储量的同时,支持高效的聚合查询.时序数据库Gorilla[28]结合时序数据的热点性,将最近一天产生的数据在内存中缓存来提高查询性能,对于历史数据,则通过压缩、预聚合的方式减少存储量.预聚合的方式导致了其查询场景具有局限性.

索引技术是数据流实时存储的关键.高效的索引能支撑数据的实时存储,并提供高效的查询服务.常见的索引有树形索引和哈希索引[29−31].AVL[32]树和B[33]树都属于平衡树,AVL树的节点只保存两个子节点指针,B树能根据定义的阶数保存多个子节点指针,在大数量场景下树高较小,在外存中广泛使用.B+树对B树优化,使其具有更好的范围查询性能,有时也被用作内存索引.T[34]树和T*[35]树兼具AVL树和B树的特点,常用于内存索引,但比起B+[36]树,T树的查询性能略差.哈希索引通过哈希函数将键值直接映射到具体的存取位置,平均复杂度为O(1),但其不支持范围查询.

海量数据下,集中式架构无法完整地存储索引结构,也限制了索引的搜索性能.研究者提出了分布式索引,根据存储、查询场景的不同,可分为主从结构和对等结构.CG-Index[37]属于典型的主从结构,其在每个从节点上构建本地B+索引,并根据本地索引的部分节点构建全局索引.查询时,需根据全局索引路由到对应的本地索引查询.文献[38]采用一致性Hash作为全局索引,将数据分块存储到不同的节点,并构建相应分块的B+树索引.由于一致性Hash只能支持点查询,该索引无法应用于水平范围切分的场景;且一致性Hash的分发机制在数据块较大或存在数据热点时,各存储节点无法达到负载均衡.何龙[39]和李斌[40]等人分别提出了面向HDFS和面向HBase的多层索引技术和二级索引结构,主要考虑大数据存储和查询.Cassandra[19]是第2类对等索引结构的典型代表,其利用一致性Hash[41]对数据分区存储,集群内部采用gossip[42]协议广播路由信息,使每个节点都能处理、转发查询请求,并能保证数据的多副本一致性.Aguilera等人[43]提出一种分布式B树,其核心思想是,将B树中的节点拆分存储到不同的存储节点上.然而该索引结构在更新时,若触发树节点分裂,会导致集群节点间的协同更新,性能较差.为了提高更新性能,文献[44]通过将多次更新合并来降低更新频率,但其只能保证数据的最终一致性.文献[45]根据节点分裂日志动态调整节点容量,避免节点频繁分裂.

海量数据场景下,可通过批量装载技术[46]加速索引构建.基于抽样的批量装载方法[47]利用采样数据集构建索引框架,但采样数据存在数据不均的局限性,难以保证索引的平衡性.基于排序的批量装载方法[48−50]先对数据集排序,然后自底向上批量构建索引,该方法会损失一定的实时性.基于缓存的批量装载方法[51,52]的核心思想是:将新写入的记录存储在节点对应的缓存中,等写入量达到阈值后,再批量更新到具体的索引节点上.Jermaine等人[53]提出的Y树索引就采用了这种思想,但其实现的复杂度较高.分布式场景下,Barbuzzi[54]利用MapReduce框架,在数据分区的前提下并行地插入数据,提高了存储效率.Wang等人[55]为了解决现实应用中对新数据和历史数据的实时检索问题,提出一种数据分片策略和基于模板的索引结构,以减少索引调整代价.上述的分布式索引以及构建方法适用于静态数据,在高速数据流场景下存在性能瓶颈.数据流往往是无状态的流水型数据,写入后就不再更新.基于LSM存储模型的分布式索引具有很高的写入性能,但查询需要串行地搜索磁盘和缓存数据,效率较低.本文针对带有时间属性的数据流,设计了主从结构的分布式索引,能够支持高效的实时数据流写入以及流数据查询.

2 分布式索引 WB-Index 结构和架构

2.1 WB-Index索引结构

对于带时间属性的数据流,每条数据流记录被称为流元组,形如$\langle t,d\rangle$,其中,t为流元组生成时间,d为具体的流元组内容.流元组d可用$\langle k,v\rangle$表示,k表示流元组的码值,v表示具体的流元组内容.数据流具有无限性,本文依据时间维度切分数据流,将数据流划分为连续的时间窗口.对于第i个时间窗口W_i,用$\langle t_i,W_i\rangle$表示,其中:t_i表示对应时间窗口的起始时间;W_i则为时间窗口内的流元组集合,形如$W_i=\{\langle t_m,d_m\rangle|m=1,2,...,n\}$,$n$为$W_i$窗口内具体的流元组数量.

对于每个时间窗口W_i,本文利用流元组码值k构建相应的下层索引.下层索引由一棵B+树索引、布隆过滤器、统计信息构成:B+树索引用于检索窗口内流元组;布隆过滤器用于过滤掉不存在码值的查询请求;统计信息

© 中国科学院软件研究所 http://www.jos.org.cn

用以记录窗口流元组码值特征,如最小值、最大值.对于连续时间窗口,形如{W_1,W_2,…,W_i,W_{i+1}},从时间维度构建上层 B+树索引,其 *key* 值为窗口 W_i 的时间戳 t_i,*value* 值为下层索引引用.提供基于时间的查询.图 1 为整个索引结构图.

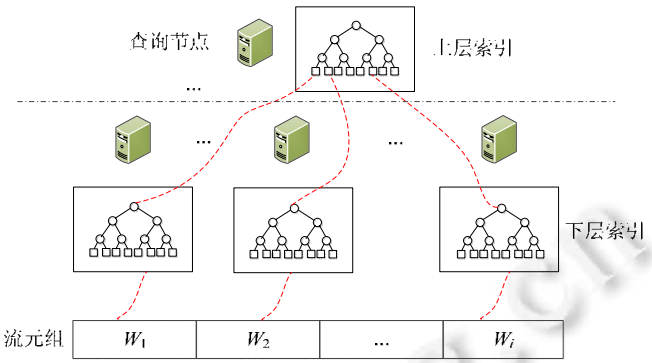

Fig.1 Index structure

图 1 索引结构

WB-Index 索引结构中,上层索引由窗口时间 t_i 及其下层索引的元数据构成,数据量较小,可在单机中存储;下层索引由流元组码值构成,存储开销大,无法在单机中维护,且各窗口对应的下层索引相互独立,因此,WB-Index 根据负载均衡原则,将各个下层索引分发到不同的节点维护,从而减轻索引的更新、存储负载.上层索引 *value* 值中,对下层索引的引用可表示为(t_i,*host*).t_i 表示对应时间窗口时间戳,*host* 表示该时间窗口下层索引所在的主机信息.查询时,与传统的主从索引结构查询类似,首先搜索上层索引,再搜索对应的下层索引获取结果.由于各个下层索引相互独立且存储在不同的节点上,因此下层索引支持并行化搜索.

2.2 WB-Index系统架构

如图 2 所示,WB-Index 集群由多个协调节点、一个构建节点、多个查询节点、多个存储节点和多个控制节点构成,集群中节点间的通信效率直接影响索引的构建和查询性能.系统实现中,将所有主机部署在同一机房以降低数据传输时延.主机间使用自定义 TCP 协议通信,其格式由定长包头和具体消息体构成.传输层采用 Protocol Buffers 数据结构,其能减少数据包大小,并具有良好的解析性能.

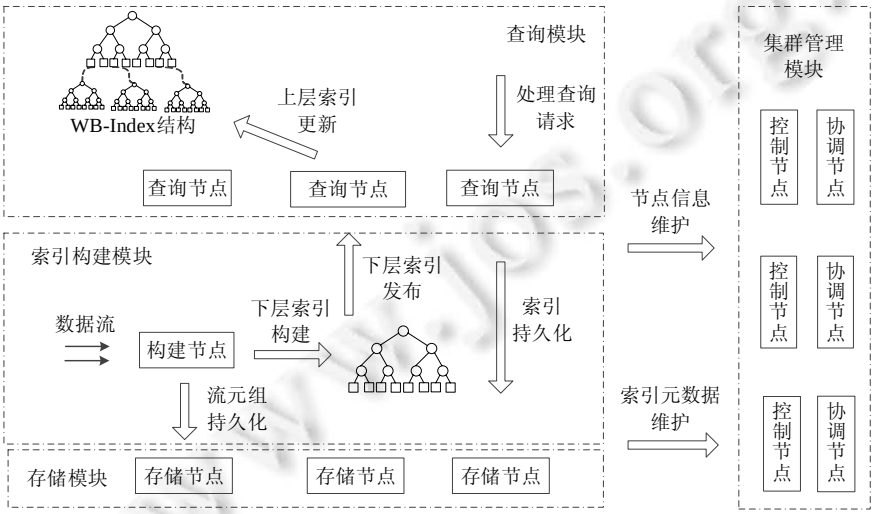

Fig.2 WB-Index architecture

图 2 WB-Index 架构

© 中国科学院软件研究所 http://www.jos.org.cn

分布式索引集群通过上述 5 类节点的相互协同,保证了高效的索引构建、数据存储和查询响应.各类节点的作用如下:

(1) 构建节点实时接收数据流,快速构建下层索引,并选择合适的查询节点发布索引结构;

(2) 查询节点维护双层索引,支持高效查询;

(3) 存储节点是集群中最底层的部分,用于持久化数据流及其索引;

(4) 协调节点保存集群各节点状态数据和索引元数据;

(5) 控制节点是集群的核心,负责合理的调度集群中的各类节点,保证集群的稳定.

构建节点接收到数据流后,将其缓存在当前的时间窗口中,并在窗口结束后触发构建下层索引,完成下层索引构建后,构建节点选择合适的查询节点发布索引,由查询节点提供查询服务.这种机制将索引构建和查询解耦,同时保证了两个模块的高效.查询节点收到下层索引后,以引用的方式将其插入上层索引,并同步更新所有上层索引副本,从而完成整个 WB-Index 的构建.窗口流元组一开始会缓存在构建节点中,由构建节点异步的将流元组持久化到存储节点.同样,受内存空间限制,查询节点中的双层索引结构也将持久化到存储节点.

WB-Index 上下两层索引都支持范围查询,其结构限定查询条件需带有时间属性.上层索引冗余维护在每个查询节点,故相应节点均能处理查询请求.分布式查询过程可分为上层索引搜索、下层索引搜索、流元组加载和数据整合.查询时,首先根据查询条件中的时间属性搜索上层索引,获取其叶节点中的引用值,并定位查询涉及的下层索引,再搜索下层索引加载对应流元组,最后整合查询结果.数据流查询过程中充分利用了内存的缓存作用,把公共查询结果(相同查询条件的流元组)、热点查询结果(热点流元组)和最新数据流元组分别缓存在查询节点和构建节点上,最大限度提升查询性能.

WB-Index 索引系统具有可扩展性,借鉴当前典型的云计算系统(如 Hadoop)来设计.WB-Index 系统可以根据系统性能需要,动态地增加查询节点与存储节点.WB-Index 系统通过采用多副本冗余机制实现故障恢复.另一方面,随着系统的运行,存在“过期”数据的处理问题.不同的应用在这方面有不同的过期数据处理要求.由于所提 WB-Index 索引的上层索引是对窗口时间的索引,过期数据位于上层 B+树索引的左侧,可以方便据此进行数据的生命周期管理:增加查询节点或存储节点扩容,或选择时间区域归档,或销毁数据(删除上层 B+树的某些叶节点).

2.3 查询节点负载均衡

构建节点将构建好的下层索引分发到查询节点,由查询节点处理查询请求.数据流速具有波动性,每个窗口中,流元组数据量有较大差异;另外,流元组具有热点性,热点数据的访问频次远高于其他数据,需将热点数据尽可能地分散到不同的查询节点上,避免查询请求堆积到个别查询节点,从而保证查询效率.因此在分发下层索引时,需保持各节点的负载均衡.传统的轮询、哈希负载均衡法没有结合数据流的特点,无法适用于 WB-Index.本文结合数据流的波动性、热点性以及集群节点间的性能差异,基于加权轮询算法,通过实时采集节点的性能指标、查询频次等信息,动态调整查询节点分发权重,实现较为精细的负载均衡.

图 3 描述了完整的负载均衡过程,其中涉及负载参数见表 1,具体步骤如下.

(1) 查询节点实时采集、计算节点负载信息,定时推送至协调节点,由协调节点管理节点负载信息;

(2) 控制节点监听协调节点中节点负载信息变化,调整各查询节点权重;

(3) 发布索引时,根据各查询节点权重,采用加权轮询算法,选取合适的节点发布.

Table 1 Node payload parameters

表 1 节点负载参数表

负载参数	描述
N_q	单位时间内下层索引查询频次
N_c	单位时间内处理器平均负载
N_m	当前空闲内存大小

© 中国科学院软件研究所 http://www.jos.org.cn

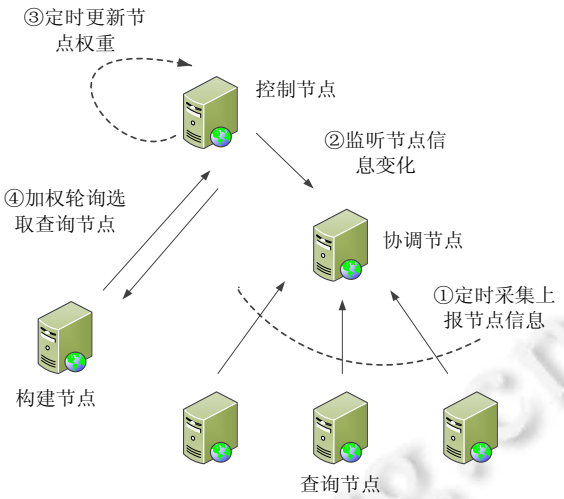

Fig.3 Load balancing method

图 3 负载均衡方法

其中,负载信息的采集、上报以及节点权重的更新频率应适中:较高的频率虽能使得负载管理更加精细,但会额外增加集群负载;较小的频率则会导致负载信息更新不及时,导致负载均衡不准确.负载均衡过程中涉及的节点权重计算方式如下.

(1) 上述的节点负载参数量级不同,为了便于加权处理,首先需进行归一化预处理:

$$x_{norm}=\frac{x-\min(x)}{\max(x)-\min(x)} \tag{1}$$

(2) 针对每个节点 n,根据预设的参数权重 k_i,计算出最终的权重值,如公式(2)所示.

$$w(n)=(k_1,k_2,k_3)\begin{pmatrix}N_q\\N_c\\N_m\end{pmatrix},\ \sum_{i=1}^{3}k_i=10 \tag{2}$$

其中,负载参数权重 k_i 可根据集群状况动态调整.例如:在内存受限的情况下,可增加 N_m 权重;在查询量大的情况下,可增加 N_q 权重.

2.4 上层索引冗余机制

WB-Index 属于主从结构,上层索引可以看成全局索引,所有的查询请求都会先搜索上层索引.若集群中只维护一份上层索引,会导致查询性能存在瓶颈.因此,本文将上层索引同步至多个查询节点上,并将索引副本元数据存储在协调节点.查询时,多个查询节点共同提供上层索引的查询服务,从而提高整体的查询性能.上层索引更新频率低、数据量小,冗余机制不会增加过多的存储开销.另外,索引副本间要保证数据一致,每当上层索引更新后,需要同步更新所有副本.在副本更新过程中,对于已经更新成功的副本是可用的,正在更新的副本则是不可用的.若一个副本在更新过程中更新失败,首先会触发多次重试,若超过重试次数后仍无法更新相应副本,则先会将协调节点上相应的元数据置为失效状态,确保不一致的索引数据对查询不可见;然后会记录失败日志,定时触发同步补偿,确保副本数据最终一致.

2.5 流元组缓存方案

WB-Index 查询涉及多次磁盘、网络 IO 操作,减少相应的 IO 操作频率能有效的提高查询效率.因此,结合数据流特点及其查询的热点性,本文设计了 3 层缓存来提高分布式查询的整体效率.缓存架构如图 4 所示,第 1 层缓存基于公共查询结果,第 2 层缓存基于热点流元组,第 3 层缓存基于较新流元组.第 1 层和第 2 层缓存主要是针对查询而设计的缓存,由于每个查询节点上下层索引不一样,所以每个查询节点上缓存的数据可能不一样.

© 中国科学院软件研究所 http://www.jos.org.cn

(1) 第 1 层缓存,用于缓存公共查询结果.本文中数据流存储的最小单元一个时间窗口对应的数据量,数据流属于流水型数据,存储后就不再更新.所以对于包含相同查询条件的查询请求,查询结果是固定的.故可将其结果缓存,减少不必要的重复查询开销.

(2) 第 2 层缓存,用于缓存热点数据.数据流存在热点性,热点数据会被频繁访问.本文将一次查询加载到的流元组缓存到查询节点,后续查询涉及相关元组即可从内存加载.缓存过程中,会将其所在下层索引叶节点对应的全部元组一并缓存,以保持适中的缓存粒度.第 2 层缓存由对应的下层索引维护,在具有数据热点的场景下,能有效减少跨节点数据加载产生的 IO 开销.

(3) 第 3 层缓存,用于缓存较新数据流.数据流的时效性强,新数据属于分析、处理过程中的热点.本文针对较新的数据流,在完成下层索引构建后,直接将其缓存在构建节点,相应的查询请求从构建节点的内存中加载数据,减少磁盘 IO,提高数据加载性能.

受内存空间限制,每层缓存均采用 LRU 淘汰算法.3 层缓存方案结合了流数据和索引结构的特点,层次分明,能有效提高查询效率.

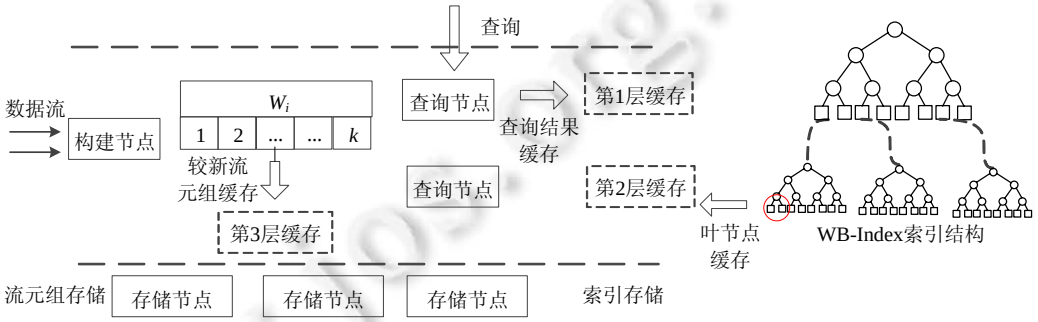

Fig.4 Three-tier stream tuple caches

图 4 3 层流元组缓存

3 分布式索引构建

图 5 描述了分布式索引的构建流程,图中 $W_1,W_2,\ldots,W_i$ 表示起始时间为 t_i、时长为 T_W 的时间窗口.如图中的 a,b,c 所示,一个窗口对应的索引构建过程包括下层索引构建、下层索引发布以及上层索引更新,其总耗时为 t_{build}.基于时间窗口的数据流处理模式需要保证窗口 W_i 结束前,完成窗口 W_{i-1} 的处理任务,即一个窗口任务的处理耗时要小于窗口时长;反之,则表明数据流的流速大于处理速率,处理任务将会积压.因此,为了保证分布式索引平稳构建,需保证 $t_{build}<T_W$.

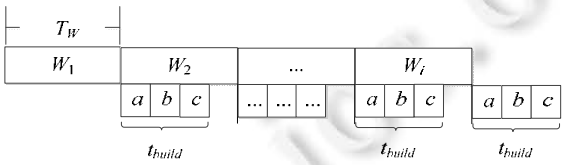

Fig.5 Process of building WB-Index

图 5 WB-Index 构建流程

3.1 下层索引构建

文献[56–58]针对数据流提出了一种多次微批量排序单次批量装载的 B+树构建方法,本文在此基础上将批量排序并行化,并在排序过程中预先装载 B+树,进一步提高构建性能.下层索引构建过程如图 6 所示,包括排序(S_1)、B+树框架搭建(S_2)、B+树预装载(S_3)、B+树赋码值(S_4)这 4 个步骤.其中,B+树框架搭建(S_2)和赋值(S_4)与文献[56]相同,故本部分不再赘述,以下仅描述优化后的排序(S_1)和预装载(S_3)部分.

© 中国科学院软件研究所 http://www.jos.org.cn

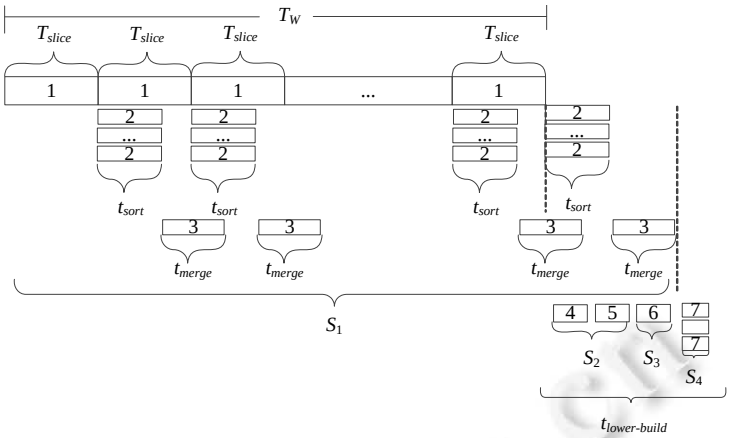

Fig.6 Construction scheme of the lower B+ tree

图 6 下层 B+树构建方法

(1) 排序

为了提升窗口数据排序性能,本文进一步切分窗口,将生成的连续分片作为最小排序单元(编号 1 所示).数据流接收过程中,每接收到完整的分片数据,就并行地触发分片内数据预排序(编号 2 所示).预排序完成后,由专门的归并线程对有序分片作二路归并排序(编号 3 所示).当接收到完整数据流后,等最后一个分片排序结束后,就完成了对整个窗口数据的排序.如图 6 所示,整个排序过程与数据流接收高度并行,理想情况下,排序时延迟仅为最后一个分片的排序和归并时间之和.

当分片内数据量较大时,可采用并行排序法加速排序,从而避免排序任务积压.并行排序的核心思想是:对数据分段,先并行地对各分段排序,最后通过多路归并方法完成全局排序.在数据量为 m、并行度为 n 的情况下,其时间复杂度为

$$O((m/n)\log(m/n)+m\log m) \tag{3}$$

传统的快速排序时间复杂度为 $O(m\log m)$,比较可得:在数据量大、机器 CPU 资源充裕的前提下,并行排序能提高排序性能;相反,直接使用传统串行排序性能更佳.

(2) B+树预装载

在 B+树框架搭建(S_2)阶段,B+树内部节点间的关系已完成关联,但是缺少叶节点与流元组之间的关联.流元组排序阶段最后涉及到归并排序,有序流元组在连续数组空间中存储,故可以在窗口结束时,根据窗口数据量预先创建数组,并根据码值在数组中的偏移值预装载 B+树叶节点中指向流元组的指针,建立叶节点与流元组之间的关联,并预计算内部节点码值对应的数组偏移量,赋值阶段(S_4)能直接根据偏移赋值.预装载完成后,B+树叶节点与流元组之间的关联也装载完毕.对于 B+树节点的码值,需在数据排序完成后进行,这是最终赋值(S_4)要完成的任务.

3.2 下层索引发布

下层索引构建过程包含很多排序、计算任务,会占用大量的机器资源,而索引的查询过程也需要较多的计算资源.因此,构建节点无法在构建索引的同时提供高效的查询服务,需将下层索引发布到查询节点,由查询节点提供查询服务.发布过程中,利用第 2.3 节描述的负载均衡方法保证各查询节点负载均衡.查询节点接收到索引数据后,需将内节点以及叶节点与流元组之间的偏移量关系转换成相应的引用关系,从而恢复下层 B+树索引.转换过程需要处理每个 B+树节点,时间复杂度为 $O(n)$.但节点间的不存在转换依赖,因此整个转换过程可以并行化.

3.3 上层索引更新

查询节点接收并恢复窗口 W_i 对应的下层索引后,将窗口 W_i 的起始时间 t_i 和下层索引的引用(t_i,*host*)构建一

© 中国科学院软件研究所 http://www.jos.org.cn

条记录,并更新到上层索引中,从而完成 W_i 窗口的索引构建.B+树是平衡树,更新过程会动态调整节点以保持平衡性.上层索引中,其 *key* 值是窗口时间,具有线性递增的特点.若采用传统的更新方法,更新过程会导致大量的节点分裂,且分裂后的节点由于 *key* 值递增的特点,后续不会再有数据写入.本文结合当前场景,针对传统更新方法存在的存储资源浪费、性能不稳定等问题,提出一种避免节点分裂的更新方法,在保持 B+树相对平衡的前提下,提高更新性能.

上层索引更新过程中,在 B+树中维护全局插入点,如图 7 所示.每次更新只需将新数据顺序的写入叶节点,且更新过程不触发节点分裂,每个节点空间都能被完全利用.若全局插入点对应的叶节点空间饱和,则向上递归查找未满父节点,再在父节点下递归创建子节点用于后续的数据写入.若所有父节点已满,则需要增加树高,创建新的根节点作为父节点.上层索引每隔一个时间窗口更新一次,当叶节点空间饱和后,可预先创建叶节点,用于后续更新写入.

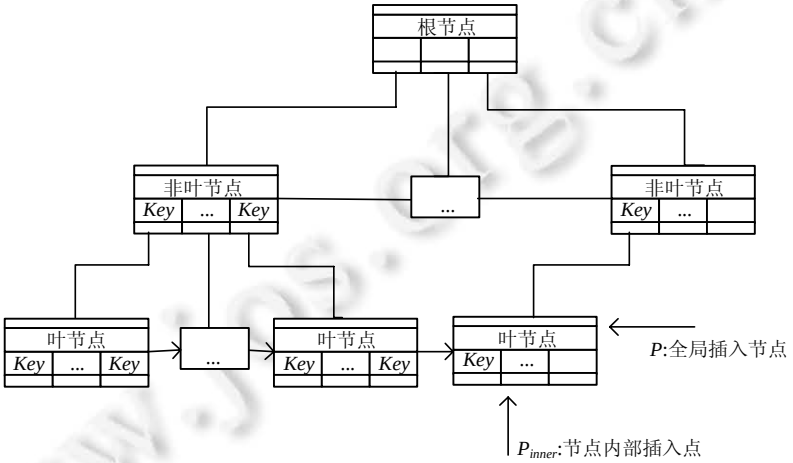

Fig.7 Upper B+ tree updates

图 7 上层 B+树更新

上层索引详细更新过程如算法 1 所述.

算法 1. 上层索引更新.

输入:*key*:待更新窗口起始时间;*value*:待更新窗口对应的下层索引引用.

输出:*root*:B+树根节点.

符号说明:*B* 表示 B+树阶数;*P* 表示上层索引插入节点;P_{index} 表示节点内部插入偏移值;n_{height} 表示上层 B+树树高.

1. 若上层索引为空,则初始化索引:
   1.1. 申请并初始化空间为 *B* 的节点 N_{init}.并将其作为根节点:$root \leftarrow N_{init}$.
   1.2. 初始化全局插入点和节点内部插入偏移值:$P \leftarrow node, P_{index} \leftarrow 0$.
   1.3. 更新树高:$n_{height} \leftarrow 1$;
2. 若上层索引不为空,则更新上层索引:
   2.1. 根据全局插入点 *P*,直接插入到相应叶节点,更新叶节点对应的 P_{index};
   2.2. 若更新后 $P_{index}=1$,表示对应的叶子节点是新节点,则需递归的对其父节点赋键值,过程如下:

```
assignKey(key,curNode){
    若当前节点 curNode 为根节点 Root 或 curNode 已满则返回;
    否则
        将键值 key 添加到 curNode;
```

© 中国科学院软件研究所 http://www.jos.org.cn

curNode←*curNode.parent*;
assignKey(*key*,*curNode*);
}

2.3. 若更新后的 P_{index}=*B*,则表示节点已满,异步调用 *expandSpace*(*P*),开辟新的节点空间;

2.4. 完成更新.

异步任务:*expandSpace*(*P*) //*P*:当前已满节点

1. 以 *P* 为起点,递归向上搜索首个未满父节点,暂存其引用值 P_{expand},过程如下:

findNotFullParent(*P*){
若 *P* 为根节点且已满,则
创建新节点作为新根节点;
P 的键值传播到新根节点,并把 *P* 作为其首个子节点;
返回 *root* 节点;
若 *P* 未满,则 P_{insert}←*P* 并返回 *P*;
若 *P* 已满,则递归查找,返回 *findNotFullParent*(*P.parent*)的结果
}

2. 以 P_{expand} 为起点,向下递归创建子节点.从而保持索引相对平衡,过程如下:

createChildNode(P_{insert}){
若 P_{insert} 已为叶节点,则结束;
否则执行:
创建容量为 *B* 的新节点 *tmp*;
设置新节点树高:当前树高−1;
把 *tmp* 添加到 P_{insert} 的子节点中;
递归调用 *createChildNode*(*tmp*),向下创建节点
}

上层索引更新过程中,若插入点为非空节点,索引更新时间复杂度为 O(1);若插入点是空节点,需额外初始化所有父节点 *key* 值,时间复杂度为 O(n_{height}).B+树树高很低,O(n_{height})可近似为 O(1).因此可得:本文所提更新方法在数据流场景下能保持稳定高效的更新效率,远优于传统的 B+树更新方法.

随着系统的运行,上层索引不断增大.对于历史数据,查询次数少,可对其归档存储.本文采用分段存储的策略,每过一个时间周期,就生成一个全新的上层索引,用于后续的更新写入,如图 8 所示;并将上层索引和周期信息的对应关系存储在协调节点上,查询时,可利用对应关系选取相应的上层索引.此外,对于老旧的索引分段,可将其存储在磁盘中,减少内存存储开销.

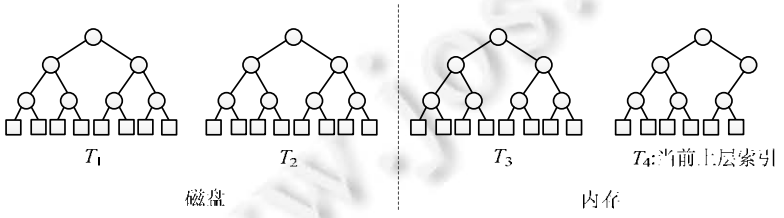

Fig.8 Upper B+ tree index segmentation

图 8 上层 B+树索引分段

4 索引构建性能评估

本节对 WB-Index 索引构建性能做量化评估,表 2 为涉及的相关参数,其中,涉及时间相关的参数单位为 s.

© 中国科学院软件研究所 http://www.jos.org.cn

Table 2 Parameters used in WB-Index construction

表 2 WB-Index 构建中用到的参数列表

参数	描述
T_W	单个时间窗口长度
T_M	一次内存访问时间
T_S	时间窗口内单个分片的时长
N_S	时间窗口分片数量
N_B	集群网络带宽(Mb/s)
N_W	时间窗口内的元组数量
N_m	下层 B+树阶数
$P_{assignKey}$	下层 B+树赋值并行度
$P_{recover}$	查询节点连接 B+树节点恢复下层 B+树的并行度
$t_{buildLower}$	下层索引构建时间
$t_{releaseLower}$	下层索引发布时间
$t_{updateUpper}$	上层索引更新时间
t_{sort}	构建下层 B+树时分片排序时间
t_{merge}	构建下层 B+树时最终归并排序时间
t_{para}	构建下层 B+树时计算 B+树参数时间
t_{skltn}	构建下层 B+树时 B+树骨架构建时间
$t_{preLoad}$	构建下层 B+树时预装载时间
$t_{assignKey}$	构建下层 B+树时赋值时间
$t_{serialize}$	序列化一个 *Node* 所需时间
$t_{deserialize}$	反序列化一个 *Node* 所需时间
$t_{transfer}$	下层索引传输开销
$t_{recover}$	查询节点连接 B+树节点恢复下层 B+树的耗时
n_{height}	上层索引树高
t_{delay}	对于一个时间窗口分布式索引的总构建时间

理论分析中,流元组码值按整型数据评估,并忽略构建过程中的 CPU 计算时间开销.

4.1 下层索引构建性能评估

$t_{buildLower}$ 表示从接收一个完整的窗口数据流到完成对应的下层索引构建的时间开销,主要包括排序和树装载开销.分片排序阶段,若分片排序时长小于分片时长($T_S<t_{sort}$),则分片排序延迟 $t_{sortDelay}$ 为最后一个分片排序耗时,即

$$t_{sortDelay}=t_{sort} \tag{4}$$

若分片排序时长大于分片时长($T_S>t_{sort}$),此时,数据流速大于排序速度,将会产生任务积压,增加额外的排序延迟,可表示为

$$t_{sortDelay}=t_{sort}+(N_S-1)\times t_{sort}-\frac{T_W(N_S-1)}{N_S} \tag{5}$$

归并排序中,分片间的归并任务串行执行,总耗时为

$$t_{merge}=2N_ST_MN_W \tag{6}$$

归并排序并行于数据流接收和分片排序,当前 $N_{slice}-1$ 个分片的归并时间开销小于 $T_W(N_S-2)/N_S$,则总的归并时延 $t_{mergeDelay}$ 由最后分片的归并时间决定,即

$$t_{mergeDelay}=4T_MN_W \tag{7}$$

若数据流流速大,前 $N_{Se}-1$ 个分片的归并时间开销大于 $T_W(N_S-2)/N_S$,则

$$t_{mergeDelay}=2T_MN_SN_W-\frac{T_W(N_S-2)}{N_S} \tag{8}$$

一个窗口数据的排序时延由分片排序时延和归并时延构成,结合公式(5)~公式(8)可得:随着分片数 T_S 增加,分片排序耗时降低,但对应的归并排序耗时会随之增加.如果将前 N_S-1 个分片的归并耗时控制在 $T_W(N_S-2)/N_S$ 内,分片数 T_S 的增加不影响归并时延.由此可得:存在一个最优分片数 N_S,使得窗口元组排序性能最佳.系统运行时,可根据数据流速动态调整分片数,以保持较好的排序性能.

© 中国科学院软件研究所 http://www.jos.org.cn

下层索引构建过程中,构建树型结构的时延可表示为 $t_{para}+t_{skltn}+t_{preLoad}+t_{assignKey}$,其中,$t_{para}$ 和 t_{skltn} 的计算公式参考文献[56].下层索引预装载环节,需计算所有节点 *key* 值偏移,则

$$t_{preLoad}=T_M\left(\frac{N_W}{m^2}\times m+\frac{N_W}{m}\times m\right)=\frac{(m+1)T_M N_W}{m} \tag{9}$$

最后的赋值环节,在 $P_{assignKey}$ 个并行度下,时间开销为

$$t_{assignKey}=2T_M\left(\frac{N_W}{m^2}\times m+\frac{N_W}{m}\times m\right)\times\frac{1}{P_{assignKey}}=\frac{2(m+1)T_M N_W}{mP_{assignKey}} \tag{10}$$

结合公式(5)~公式(10)可得:排序时延远大于下层索引框架构建时延(t_{para} 和 t_{skltn} 的计算公式参考文献[56]),且 $t_{para},t_{skltn},t_{preLoad}$ 与排序并行,因此总的下层索引构建开销为

$$t_{buildLower}=t_{sortDelay}+t_{mergeDelay}+t_{assignKey}\approx t_{sortDelay}+t_{mergeDelay} \tag{11}$$

其中,赋值阶段的时间开销远小于排序阶段,排序性能直接影响下层索引构建性能,故分片数 N_S 直接决定了下层索引的构建性能.

4.2 下层索引发布性能评估

下层索引发布的时间开销主要来自节点间的数据传输和树结构的恢复开销,可表示为

$$t_{releaseLower}=t_{serialize}+t_{send}+t_{deserialize}+t_{recover} \tag{12}$$

索引发布阶段,发送方需先将其结构数据序列化.根据序列化方式的不同,过程中会写入标识字符,以便于反序列化相应结构.为了便于理论评估,分析过程中忽略这些字符,并将反序列化和序列化两者的时间开销近似等价,即

$$t_{deserialize}\approx t_{serialize}=5T_M\left(\frac{N_W}{m}\times m+\frac{N_W}{m}\times 2+\frac{N_W}{m^2}+\frac{N_W}{m^2}\times m+\frac{N_W}{m^2}\times m\right)=\frac{5T_M N_W(m^2+4m+1)}{m^2} \tag{13}$$

下层索引传输阶段,若每个树节点序列化后的大小为 S(通过预实验得,子节点数 m=3000 时,S=17KB),则数据传输开销为

$$t_{send}=\frac{S\left(\frac{N_W}{m}+\frac{N_W}{m^2}\right)}{8\times 10^6 N_B} \tag{14}$$

查询节点收到索引节点后,并行地将内节点以及叶节点与流元组之间的偏移量关系转换成相应的引用关系,从而恢复下层索引,耗时为

$$t_{recover}=2T_M\left(\frac{N_W}{m^2}\times(m+1)+\frac{N_W}{m}\times(m+2)\right)\times\frac{1}{P_{recover}}=\frac{2T_M N_W(m^2+3m+1)}{m^2P_{recover}} \tag{15}$$

结合公式(12)~公式(15)可得:索引发布的时间开销主要来自数据传输,节点间的带宽直接决定了分布式索引的构建效率.

4.3 上层索引更新性能评估

下层索引构建完成后,需要更新上层索引.上层索引更新在最坏的情况下,时间复杂度为 $O(n_{height})$,时间开销为

$$t_{updateUpper}=\log(n_{height})T_M \tag{16}$$

由公式(16)可得,上层索引的更新开销相比下层索引构建和发布开销可忽略不计.

4.4 WB-Index整体性能分析

由以上分析可知,WB-Index 构建性能主要取决于下层 B+树索引的构建和发布性能.一个时间窗口的分布式索引构建总时间开销 $t_{delay}=t_{buildLower}+t_{releaseLower}+t_{updateUpper}$.为了确保 WB-Index 能够平稳的构建,有如下限制:

$$t_{delay}<T_W \tag{17}$$

若不满足该限制,流数据将会不断堆积,影响索引构建,对应的查询结果也失去准确性,整个分布式索引将

© 中国科学院软件研究所 http://www.jos.org.cn

会不可用.对于时长为 T_w 的窗口 W_i,其中包含的元组数 N_w 与 t_{delay} 正相关,因此窗口内包含的元组数存在上限 $N_{\max}$,对应的数据流速 $N_{\max}/T_W$ 即为索引所能承受的最大流速.同理,下层索引发布时长 $t_{releaseLower}$ 与网络传输带宽 N_B 相关,也会影响整体构建时延,因此也有带宽下限约束.带宽与数据流流速应成正相关关系,其联系由公式(17)决定.

5 实验评价

5.1 实验环境

本次实验选用阿里云平台,集群环境由 20 个 ECS 实例节点组成.分布式索引选用 JAVA 语言实现.表 3 为 ECS 实例详细配置.为了便于描述,用 ECS01~ECS20 对节点编号,各节点具体安排见表 4,其中,将负载较小的节点类型部署在一起,以最大化资源利用率.

Table 3 ECS instance's software/hardware configuration

表 3 ECS 实例软硬件配置

类别	配置
系统	Ubuntu 16.04,64bit
CPU	4 cores Intel Xeon Platinum 8163@2.50GHz
内存	32GB
网络	1Gbps
磁盘	100GB

Table 4 Types of nodes in WB-Index cluster

表 4 WB-Index 集群节点构成

编号	类型
ECS01	构建节点
ECS02,ECS03	查询节点,协调节点
ECS04	控制节点,协调节点,查询节点
ECS05~ECS14	查询节点
ECS15~ECS20	存储节点

实验使用 TPC-H 生成的 1 亿条 Line Item 数据集,数据格式为⟨*K*,*V*⟩,其中,*V* 的大小为 100B.实验中,实时从内存加载数据来模拟接收一个数据流,并通过改变加载量来模拟不同的数据流速.WB-Index 中,B+树节点阶数 *m*=3000.

5.2 下层索引构建性能评估

由第 4.4 节的理论分析可得,下层索引构建性能是影响分布式索引构建性能的重要因素.下层索引构建中的分片排序阶段,通过并行排序法加速排序.通过预实验得到:

- 当前机器配置下,分片元组数量小于 150 万时,传统快速排序性能更好.
- 超过该数据量时,用 2 个线程并行排序性能较好.实验中,按照此规则选用合适的排序方式.

由第 4.1 节的理论分析可得,分片数 N_{slice} 直接决定了下层索引的构建性能.本实验在 20s 的时间窗口内,模拟 100 万/s 的数据流,评估分片数对下层索引的影响.

图 9 为具体实验结果,从中可得:

- 随着分片数的增加,下层索引构建性能先升后降;
- 分片数量达到 25 个时,构建性能趋于稳定,并在 35 个时性能最优.

这与第 4.1 节的分析结论相符.实验对比了下层索引构建中各阶段耗时,由图 9 可得,排序是最主要的时间开销.因此,将构建节点部署在高性能主机上,能有效提高下层索引构建效率.

数据流流速存在波动性,本实验在 20s 的时间窗口内,通过模拟不同数据流速,评估下层索引构建的稳定性,实验设置窗口分片数 N_{slice}=35.

© 中国科学院软件研究所 http://www.jos.org.cn

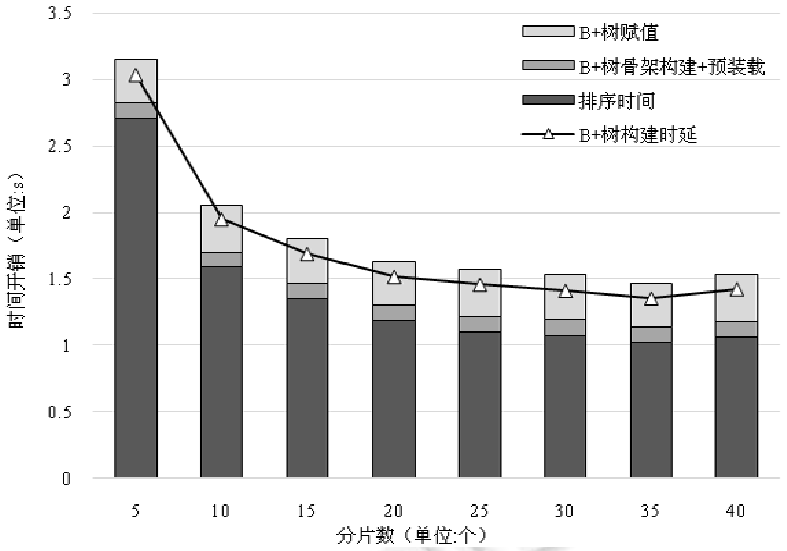

Fig.9 Construction delay of the lower B+ tree vs. the number of slices

图 9 下层索引构建延迟与分片数的关系

如图 10 为具体实验结果,从中可得:下层索引构建效率与数据流速呈线性关系,充分证明构建方法的稳定性.当数据流速达到 250 万/s,仅存在 4.2s 的构建延迟.这也证明了索引构建性能好,能支持高速数据流写入.

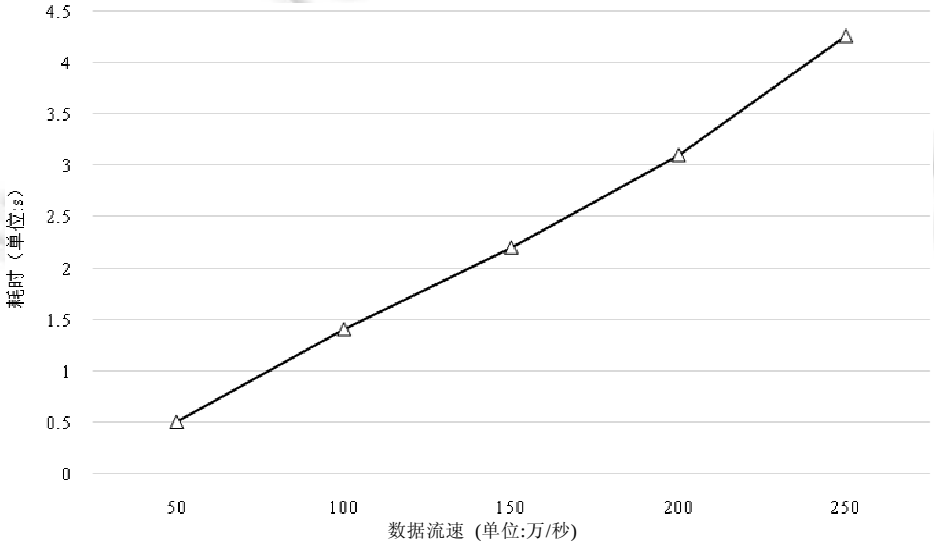

Fig.10 Construction delay of the lower B+ tree vs. the varying stream rates

图 10 不同数据流速下,下层索引的构建延迟

5.3 上层索引构建性能评估

针对上层索引,本文根据场景中 *key* 值递增的特点,在更新过程中避免节点分裂,保证存储效率.本实验评估不同上层索引规模下的更新性能.

图 11 为具体实验结果,从中可得:随着更新次数的指数增长,上层索引规模不断增大,但每次索引的更新时间基本保持稳定,千万次更新总耗时仅为 10s 左右.这符合第 4.3 节的结论,上层索引更新耗时可忽略,在数据流场景下不存在性能瓶颈.

© 中国科学院软件研究所 http://www.jos.org.cn

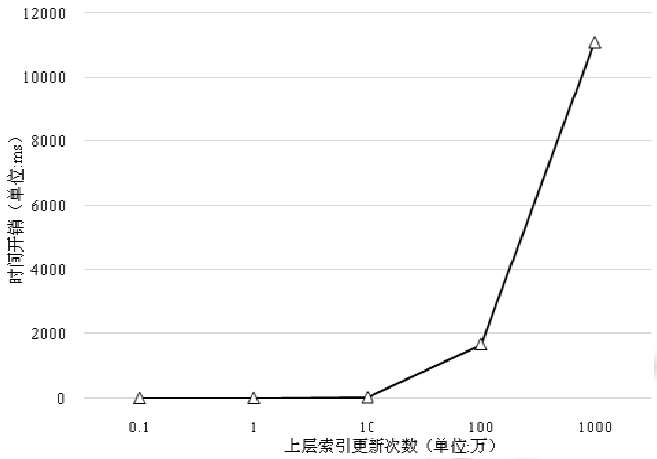

Fig.11 Update time vs. the number of the upper B+ tree updates

图 11 上层索引更新次数与更新时间的关系

5.4 WB-Index索引构建性能评估

本实验在 20s 的时间窗口,通过模拟不同数据流速来评估 WB-Index 构建性能,并对比其包含的下层索引构建、下层索引发布、上层索引更新这 3 个阶段的耗时.实验设置窗口分片数 N_{slice}=35.

实验结果如图 12 所示,从中可得:WB-Index 索引构建效率与数据流速呈线性关系,下层索引发布过程由于存储网络 IO,耗时较长.因此,提高节点间带宽能有效提高 WB-Index 构建性能.

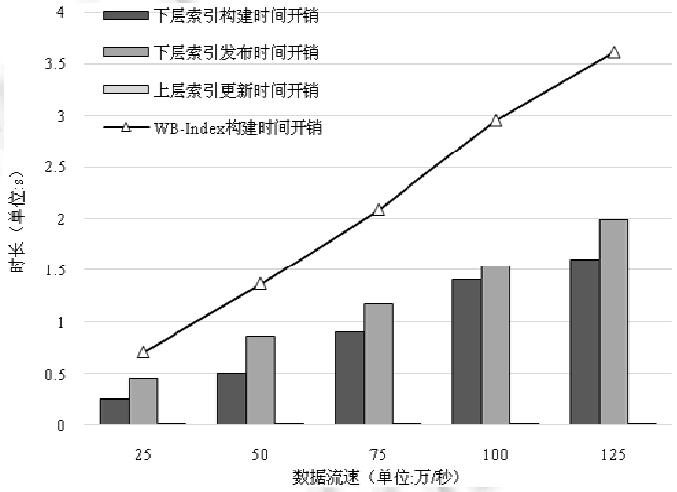

Fig.12 Construction delay of WB-Index vs. the varying stream rates

图 12 数据流流速与 WB-Index 构建时延的关系

如上文所述,CG-Index 和 LSM-树模型均能应用于分布式场景,属于分布式索引结构.本实验模拟不同的数据流速,对比 LSM-树、CG-Index 和 WB-Index 这三者的构建效率.为了保证实验具有可比性,模拟的数据流持续时间为 100s,CG-Index 中分配 10 个节点用于存储,设置 LSM-树模型中触发转储的内存阈值为 200 万条数据.WB-Index 窗口时长 Tw=20,分片数 N_{slice}=35.

图 13 为具体实验结果,从中可得:相比于 LSM-树和 CG-Index,WB-Index 的构建性能更优;在大流速的场景,其性能优势更加明显.CG-Index 将数据流存储在多个节点上,并在对应节点构建 Local Index.但随着数据量的不断增长,Local Index 的更新开销会快速增加,在大规模的数据场景下,存在性能瓶颈.LSM-树模型实时写入数据,

© 中国科学院软件研究所 http://www.jos.org.cn

并根据设置的内存阈值定期将数据转储到磁盘,具有较为稳定的构建性能.相比之下,WB-Index 在构建过程中会缓存当前窗口的数据流,损失一定的实时性,但大幅度提高了构建效率,更适用于与大流量、高流速的数据流场景.查询方面,CG-Index 主要用于二级索引,元组数据量较小,通过缓存节点来提高查询性能.而在 LSM-树中,若对应元组较大,会较为频繁地触发磁盘转储,查询时,需依次搜索磁盘中的索引结构,直到获取查询结果,性能较差.WB-Index 下层索引叶节点只记录具体流元组偏移,这保证其索引结构较小,可将其大量缓存来提高查询性能.此外,WB-Index 能够支持并行查找,整体查询性能优于 LSM-树.

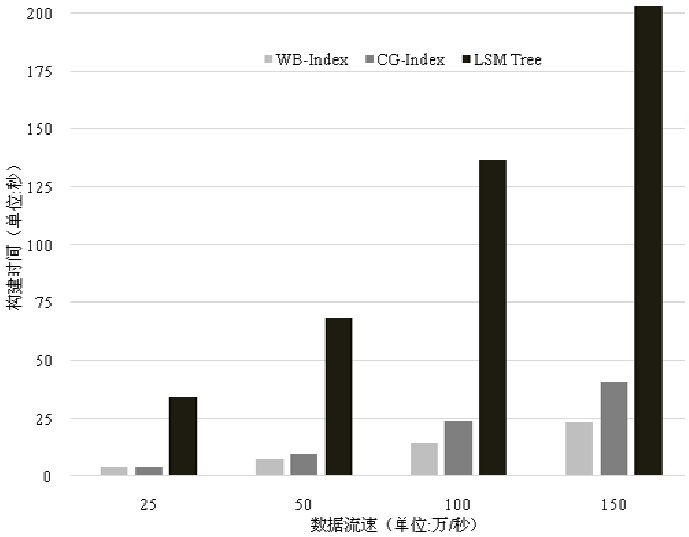

Fig.13 Construction delay comparisons between WB-Index, CG-Index and LSM tree

图 13 WB-Index、CG-Index 与 LSM 树索引的构建延迟对比

由第 4.4 节分析得出:为了保证索引平稳构建,需保证构建时延小于窗口时长.本实验在窗口时长 T_w=5s,分片数 N_{slice}=35 时,评估 WB-Index 所能支撑的数据流速上限.实验中,假设单个窗口内数据流速稳定.

图 14 为具体实验结果:当数据流速达到 360 万/s 时,其构建时延等于窗口时长.因此,360 万/s 即是当前集群环境下 WB-Index 所能承受的最大数据流速.

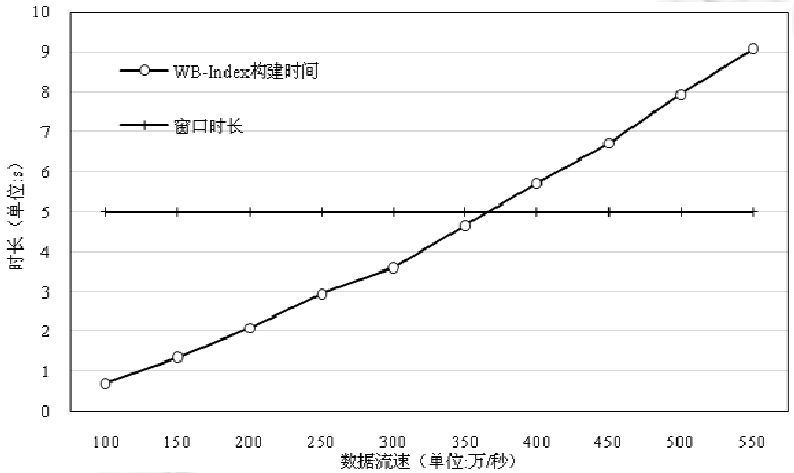

Fig.14 WB-Index construction delay vs. the varying stream rates

图 14 WB-Index 构建时延与数据流速的关系

© 中国科学院软件研究所 http://www.jos.org.cn

5.5 WB-Index查询性能

WB-Index 中的上下两层索引均能支持范围查询,本实验针对基本的取值查询,通过改变窗口和元组查询范围评估查询总体性能.实验设置窗口元组数量 N_W=100 万,并预先将索引和流数据缓存在查询节点和构建节点.

图 15 为具体结果,s 表示单个窗口内流元组的查询范围.从中可得,查询耗时基本小于 1s.实验中,最大查询范围对应的查询结果包含 60 万个元组,大小为 60MB,传输开销大,也影响了相应的查询性能.

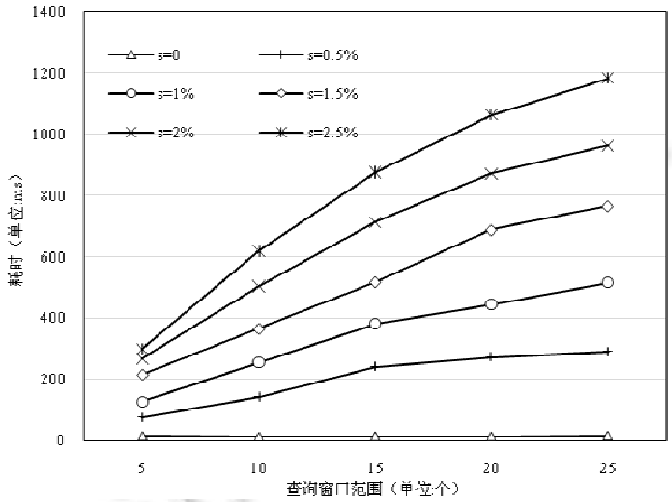

Fig.15 Query time vs. query scope on WB-Index

图 15 WB-Index 查询时间与查询范围的关系

WB-Index 查询过程包括上层索引搜索、下层索引搜索、流元组加载和数据整合.本实验限定 1%的窗口流元组查询范围,通过改变窗口查询范围,对比分析各查询环节耗时.

图 16 为具体实验结果,从中可得:当索引结构缓存在内存时,上下两层索引的搜索开销很小.下层索引支持并行搜索,查询窗口范围的增加对索引搜索性能影响较小.数据加载和整合阶段相比索引搜索阶段耗时较大,数据加载阶段需从存储节点或构建节点加载流元组,网络、磁盘 IO 导致其开销较大,若将相应元组直接缓存到查询节点,即可避免 IO 开销而减少耗时,这也表明了缓存热点流元组的必要性.本实验中,查询结果数据量大,从而导致最后的整合阶段耗时较长.

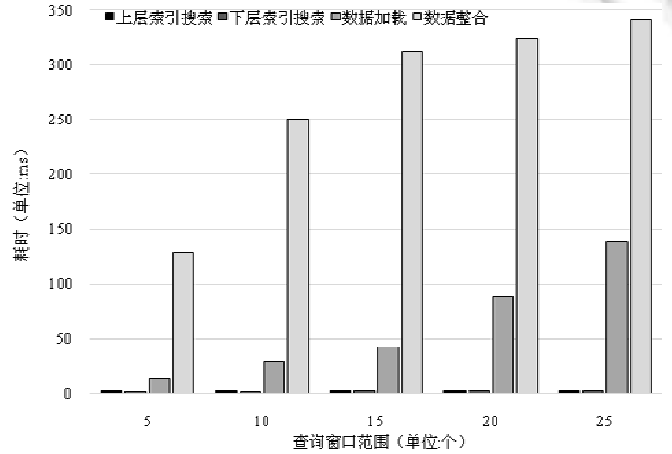

Fig.16 Time comparisons between each query stage on WB-Index

图 16 WB-Index 各查询阶段耗时对比

© 中国科学院软件研究所 http://www.jos.org.cn

6 结束语

本文面向完整保存数据流的应用,结合数据流的特点,提出双层分布式索引:WB-Index.在保证索引构建效率的同时,通过高效缓存保障查询效率;通过负载均衡策略和上层索引冗余机制,保证索引系统的可用性和稳定性.针对分布式索引构建:采用时间窗口的方式切分数据流,基于窗口内流元组构建下层索引,采用高度并行化、批量化的方式加速构建性能;上层索引基于所有时间窗口构建,考虑到时间窗口时间戳的递增性,提出避免分裂的 B+树插入策略,减少 B+树分裂移动开销,提高空间利用率,并提出预分配策略提高数据插入的性能.通过实验验证分布式索引的构建性能,相比起 CG-index 和 LSM 存储模型均有很大的性能优势,适用于高速数据流场景.今后将进一步研究双层分布式索引的持久化方法.

致谢 感谢阿里巴巴数据库事业部为本文的研究提供了实验所需的软硬件资源,感谢彭立勋先生为本研究提供诸多讨论和帮助.

References:

[1] Golab L, Özsu MT. Issues in data stream management. ACM SIGMOD Record, 2003,32(2):5−14.

[2] Xie J, Yang J. A survey of join processing in data streams. In: Data Streams: Models and Algorithms. 2007. 209−236.

[3] Acharya S, Gibbons PB, Poosala V, Ramaswamy S. Join synopses for approximate query answering. ACM SIGMOD Record, 1999, 28(2):275−286.

[4] Babcock B, Babu S, Datar M, Motwani R, Widom J. Models and issues in data streams. In: Proc. of the PODS Conf. 2002. 1−16.

[5] Arasu A, Babcock B, Babu S, Cieslewicz J, Datar M, Ito K, Motwani R, Srivastava U, Widom J. Stream: The Stanford data stream management system. In: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. 2003.

[6] Carney D, Cetinternel U, Cherniack M, Convey C, Lee S, Seidman G, Stonebraker M, Tatbul N, Zdonik S. Monitoring streams: A new class of dbms applications. In: Proc. of the VLDB Conf. 2002. 215−226.

[7] Krishnamurthy S, Chandrasekaran S, Cooper O, Deshpande A, Franklin MJ, Hellerstein JM, Hong W, Madden S, Reiss F, Shah MA. TelegraphCQ: An architectural status report. Data Engineering Bulletin, 2003,26(1):11−18.

[8] Toshniwal A, Taneja S, Shukla A, Ramasamy K, Kulkarni S, Jackson J, Gade K, Fu MS, Donham J, Bhagat N, Mittal S, Ryaboy D. Storm@twitter. In: Proc. of the VLDB Int'l Conf. on Management of Data. 2014. 147−156.

[9] Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K. Apache flink: Stream and batch processing in a single engine. Data Engineering Bulletin, 2015,36(4):28−38.

[10] Zhang DD, Li JZ, Wang WP, Guo LJ. Algorithms for storing and aggregating historical streaming data. Ruan Jian Xue Bao/Journal of Software, 2005,16(12):2089−2098 (in Chinese with English abstract). http://jos.org.cn/jos/article/abstract/20051207?st=article_issue [doi: 10.1360/jos162089]

[11] Ge JW, Gong PQ, Liu ZH. Method of storing and indexing historical streaming data. Application Research of Computers, 2007, 24(6):104−106 (in Chinese with English abstract).

[12] Lahiri T, Neimat MA, Folkman S. Oracle TimesTen: An in-memory database for enterprise applications. Data Engineering Bulletin, 2013,36(2):6−13.

[13] Larson PÅ, Zwilling M, Farlee K. The Hekaton memory-optimized OLTP engine. Data Engineering Bulletin, 2013,36(2):34−40.

[14] Lindström J, Raatikka V, Ruuth J, Soini P, Vakkila K. IBM solidDB: In-memory database optimized for extreme speed and availability. Data Engineering Bulletin, 2013,36(2):14−20.

[15] O'Neil P, Cheng E, Gawlick D, O'Neil E. The log-structured merge-tree (LSM-tree). Acta Informatica, 1996,33(4):351−385.

[16] Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Gruber RE, Chandra T, Fikes A, Gruber RE. Bigtable: A distributed storage system for structured data. ACM Trans. on Computer Systems, 2008,26(2):1−26.

[17] Antaris S, Rafailidis D. In-memory stream indexing of massive and fast incoming multimedia content. IEEE Trans. on Big Data, 2018, 4(1):40−54.

[18] Ghemawat S, Gobioff H, Leung ST. The Google file system. In: Proc. of the ACM Symp. on Operating Systems Principles. ACM, 2003. 20−43.

[19] Lakshman A, Malik P. Cassandra: A decentralized structured storage system. ACM SIGOPS Operating Systems Review, 2010, 44(2):35−40.

© 中国科学院软件研究所 http://www.jos.org.cn

[20] Fusco F, Vlachos M, Stoecklin MP. Real-time creation of bitmap indexes on streaming network data. The VLDB Journal, 2012, 21(3):287−307.

[21] Pu KQ, Zhu Y. Efficient indexing of heterogeneous data streams with automatic performance configurations. In: Proc. of the Scientific and Statistical Database Management. IEEE, 2007. 34−34.

[22] Liu L, Pu C, Tang W. Continual queries for Internet scale event-driven information delivery. IEEE Trans. on Knowledge and Data Engineering, 1999,11(4):583−590.

[23] Chen J, DeWitt DJ, Tian F, Wang Y. NiagraCQ: A scalable continuous query system for internet databases. In: Proc. of the SIGMOD Conf. 2000. 379−390.

[24] Lin JC, Lee MC, Yu IC, Johnsen EB. Modeling and simulation of spark streaming. In: Proc. of the Int'l Conf. on Advanced Information Networking and Applications. 2018. 407−413.

[25] Yan M, Wang L, Zomaya AY, Chen D, Ranjan R. Task-tree based large-scale mosaicking for massive remote sensed imageries with dynamic DAG scheduling. IEEE Trans. on Parallel & Distributed Systems, 2014,25(8):2126−2137.

[26] Suna T. Opentsdb. 2017. http://opentsdb.net/

[27] Yang F, Tschetter E, Léauté X, Ray N, Merlino G, Ganguli D. Druid: A real-time analytical data store. In: Proc. of the ACM SIGMOD Conf. 2014. 157−168.

[28] Pelkonen T, Franklin S, Teller J, Cavallaro P, Huang Q, Meza J, Veeraraghavan K. Gorilla: A fast, scalable, in-memory time series database. Proc. of the VLDB Endowment, 2015,8(12):1816−1827.

[29] Knuth DE. The Art of Computer Programming. Pearson Education, 2005.

[30] Fagin R, Nievergelt J, Pippenger N, Strong HR. Extendible hashing: A fast access method for dynamic files. ACM Trans. on Database Systems, 1979,4(3):315−344.

[31] Litwin W. Linear Hashing: A new tool for file and table addressing. In: Proc. of the VLDB Conf. 1980. 1−3.

[32] Aho AV, Hopcroft JE, Ullman JD. The Design and Analysis of Computer Algorithms. Pearson Education India, 1974.

[33] Comer D. The ubiquitous B-tree. ACM Computing Surveys, 1979,11(2):121−137.

[34] Lehman TJ, Carey MJ. A study of index structures for main memory database management systems. In: Proc. of the VLDB Conf. 1986. 294−303.

[35] Choi KR, Kim KC. T*-tree: A main memory database index structure for real time applications. In: Proc. of the Real-time Computing Systems and Applications. 1996. 81−88.

[36] Lu H, Ng YY, Tian Z. T-tree or B-tree: Main memory database index structure revisited. In: Proc. of the Australian Database Conf. 2000. 65−73.

[37] Wu S, Jiang D, Ooi BC, Wu KL. Efficient B-tree based indexing for cloud data processing. Proc. of the VLDB Endowment, 2010, 3(1-2):1207−1218.

[38] Li X, Ren C, Yue M. A distributed real-time database index algorithm based on B+ tree and consistent hashing. Procedia Engineering, 2011,24:171−176.

[39] He L, Chen JC, Du XY. Multi-layered index for HDFS-based system. Ruan Jian Xue Bao/Journal of Software, 2017,28(3):502−513 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5161.htm [doi: 10.13328/j.cnki.jos.005161]

[40] Li B, Guo JW, Peng Q. Design of HBase secondary indexes for big data storage. Computing Technology and Automation, 2019, 38(2):124−129 (in Chinese with English abstract).

[41] Karger DR, Lehman E, Leighton T, 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 Symp. on the Theory of Computing. 1997. 654−663.

[42] Van RR, Dumitriu D, Gough V, Thomas C. Efficient reconciliation and flow control for anti-entropy protocols. In: Proc. of the 2nd Workshop on Large-scale Distributed Systems and Middleware. ACM, 2008. Article No.6.

[43] Aguilera MK, Golab W, Shah MA. A practical scalable distributed B-tree. In: Proc. of the VLDB Conf. 2008. 598−609.

[44] Ahmed M, Singh SS, Lee MJ. Lazy updates to indexes in a database. U.S. Patent 20090089334, 2009.

[45] Huang B, Peng YX. An efficient distributed B-tree index method in cloud computing. Open Cybernetics & Systemics Journal, 2014, 8:302−308.

[46] Bercken J, Seeger B. An evaluation of generic bulk loading techniques. In: Proc. of the VLDB Conf. 2001. 461−470.

[47] Lo ML, Ravishankar CV. The design and implementation of seeded trees: An efficient method for spatial joins. IEEE Trans. on Knowledge and Data Engineering, 1998,10(1):136−152.

[48] Ciaccia P, Patella M. Bulk loading the M-tree. In: Proc. of the Australian Database Conf. 1998. 15−26.

© 中国科学院软件研究所 http://www.jos.org.cn

[49] Kotidis Y, Roussopoulos N. An alternative storage organization for ROLAP aggregate views based on cubetrees. In: Proc. of the 1998 ACM SIGMOD Conf. 1998. 249−258.

[50] Kondylakis H, Dayan N, Zoumpatianos K, Palpanas T. Coconut: Sortable summarizations for scalable indexes over static and streaming data series. The VLDB Journal, 2019,28(6):847−869

[51] Bercken J, Seeger B, Widmayer P. A generic approach to bulk loading multidimensional index structures. In: Proc. of the VLDB Conf. 1997. 406−415.

[52] Arge L, Hinrichs KH, Vahrenhold J, Vitter JS. Efficient bulk operations on dynamic R-trees. Algorithmica, 2002,33(1):104−128.

[53] Jermaine C, Datta A, Omiecinski E. A novel index supporting high volume data warehouse insertion. In: Proc. of the VLDB Conf. 1999. 235−246.

[54] Barbuzzi A, Michiardi P, Biersack E, Boggia G. Parallel bulk insertion for large-scale analytics applications. In: Proc of the Int'l Workshop on Large Scale Distributed Systems and Middleware. 2010. 27−31.

[55] Wang L, Cai R, Fu ZJ, He J, Lu Z, Winslett M, Zhang Z. Waterwheel: Realtime indexing and temporal range query processing over massive data streams. In: Proc. of the ICDE Conf. 2018. 269−280.

[56] Xiang JJ. Distributed in-memory hybrid index for big data stream [MS. Thesis]. Hangzhou: Zhejiang University of Technology, 2017 (in Chinese with English abstract).

[57] Yang LH, Xiang JJ, Xu W, Fan YL. In-memory B+tree construction methodology for big data stream. Computer Science, 2018, 45(3):173−179,214 (in Chinese with English abstract).

[58] Lu CX. A distributed B+ tree for big data stream [MS. Thesis]. Hangzhou: Zhejiang University of Technology, 2019 (in Chinese with English abstract).

附中文参考文献:

[10] 张冬冬,李建中,王伟平,郭龙江.数据流历史数据的存储与聚集查询处理算法.软件学报,2005,16(12):2089−2098. http://jos.org.cn/jos/article/abstract/20051207?st=article_issue [doi: 10.1360/jos162089]

[11] 葛君伟,公丕强,刘兆宏.一种存储和索引历史数据流数据的方法.计算机应用研究,2007,24(6):104−106.

[39] 何龙,陈晋川,杜小勇.一种面向 HDFS 的多层索引技术.软件学报,2017,28(3):502−513. http://www.jos.org.cn/1000-9825/5161.htm [doi: 10.13328/j.cnki.jos.005161]

[40] 李斌,郭景维,彭骞.面向大数据存储的 HBase 二级索引设计.计算技术与自动化,2019,38(2):124−129.

[56] 项俊健.面向流数据的分布式混合内存索引[硕士学位论文].杭州:浙江工业大学,2017.

[57] 杨良怀,项俊腱,徐卫,范玉雷.一种大数据流内存 B+树构建方法.计算机科学,2018,45(3):173−179,214.

[58] 卢晨曦.面向大数据流的分布式 B+树索引构建[硕士学位论文].杭州:浙江工业大学,2019.

杨良怀(1967－),男,博士,教授,CCF 专业会员,主要研究领域为数据库系统,大数据处理.

卢晨曦(1994－),男,硕士,主要研究领域为数据流存储.

范玉雷(1984－),男,博士,讲师,CCF 专业会员,主要研究领域为数据库系统,数据流,数据挖掘.

朱镇洋(1994－),男,硕士,主要研究领域为自然语言处理.

潘建(1976－),男,博士,副教授,CCF 专业会员,主要研究领域为智能信息处理,物联网技术.

© 中国科学院软件研究所 http://www.jos.org.cn