一种大数据流内存B+树构建方法

杨良怀 项俊腱 徐 卫 范玉雷

(浙江工业大学计算机科学与技术学院 杭州 310023)

摘 要 面向具有时间维度的大数据流,基于二级B+树索引结构,提出了一种高效的面向时间窗口、采用批量装载技术的内存B+树构建方法。该方法对时间窗口进行分片,通过分离出可以并行处理的操作来加速构建过程,将排序操作与数据流接收并行,B+树骨架的构建与排序并行;采用基于排序的批量装载技术以及优化的构建顺序,能够避免多线程之间不必要的加锁、同步开销,有效提高构建效率。提出的多次微批量排序单次批量装载(MBSortSBLoad)B+树构建方法的构建速度快,能承载的最大流速大。实验验证了所提方法的有效性。

关键词 B+树,数据流,内存索引,大数据

中图法分类号 TP333.3 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2018.03.027

In-memory B+tree Construction Methodology for Big Data Stream

YANG Liang-huai XIANG Jun-jian XU Wei FAN Yu-lei

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

Abstract This paper investigated into the issues of indexing on data stream with time dimension in near real-time. By resorting to 2-tier B+tree index,this paper invented a highly effective in-memory B+tree construction method for scenarios with real-time query requirements,which separates as many parallelizing operations as possible. This paper parallelized the operations of sorting and data receiving by dividing the time-window into equal-duration slice,and parallelized the construction of B+tree skeleton with sorting. This paper avoided unnecessary locking and synchronizing cost by adopting sorting-based bulk loading techniques and optimized constructing sequence. The proposed in-memory B+tree construction algorithm called MBSortSBLoad can build B+tree quickly and accept higher data arriving rates. Extensive experiments demonstrate the effectiveness of the proposed methods.

Keywords B+tree,Data stream,In-memory index,Big data

1 引言

随着物联网技术(IoT)的发展,数据获取变得便捷,数据流的应用越来越广泛。数据流是由流元组构成的一个无限的有序序列。流元组的形式为$\langle t,s\rangle$,其中t指流元组的时间戳,s指数据值。与传统的关系型元组相比,数据流具有以下特点[1]:数据流连续、实时到达;数据流的潜在规模是无限的;系统无法控制数据流到达的速率和元组的顺序;数据流一经处理,除非特意保存,否则难以再次处理。针对数据流潜在规模无限的特点,研究者提出使用时间窗口来限定流元组的处理范围,缓存不断到达的流元组。时间窗口指定了当前处理的数据流元组的起始位置和结束位置。传统的关系型数据库管理系统在处理规模有限的、静态的数据集合时具有良好的性能,但是在处理数据流时存在严重的不足。

在数据流处理系统方面,研究人员做了大量的研究工作,如STREAM系统[2]、TelegraphCQ系统[3]、Aurora系统[4]等。上述流数据库系统的目的在于支持在线数据的静态查询、连续查询、近似计算等,不在磁盘上保存整个数据流,仅保存查询结果。近年来,数据流应用不再局限于在流数据实时处理的速度和查询结果的精确度方面的高要求,而是拓广到了海量流数据的深度分析上。这对数据流的实时存储提出了要求。上述数据流管理系统没有关注数据流的实时存储问题。

数据流实时、连续的特点对数据流的存储速度提出了很高的要求。李建中等人[5]针对数据流的巨大体量,采用抽样方法实现数据流的存储。数据流实时存储另一方面是实时构建索引,实现即时查询操作。Fusco等人[6]对网络数据包流的特殊应用提出了一种实时位图索引,Pu等人[7]针对传感器网络中的异构数据流构建位图索引,但是上述工作都只是对特定的数据流应用提出了有效的索引构建方式。

随着内存容量持续增加、价格降低,将数据库放到内存中以取得高性能已成为现实,如:商用内存数据库系统Times-

到稿日期:2017-01-20 返修日期:2017-04-15 本文受浙江省基金项目(LY14F020017,LQ15F020007),国家基金项目(61070042)资助。

杨良怀(1967-),男,博士,教授,主要研究方向为数据库系统,E-mail:yanglh@zjut.edu.cn;**项俊腱**(1992-),男,硕士,主要研究方向为数据库系统;**徐 卫**(1966-),男,硕士,高级工程师,主要研究方向为信息系统,E-mail:xw@zjut.edu.cn(通信作者);**范玉雷**(1984-),男,博士,讲师,主要研究方向为数据库系统。

Ten,SolidDB,Hekaton 等。本文针对现有工作的不足,基于二级 B+树索引结构,提出了面向时间窗口的批量构建内存 B+树索引方法,近实时构建 B+树索引,并提供近实时查询。将一个时间窗口内的流元组构建成一棵内层 B+树;将该内层 B+树索引整体作为一个“value”值,以时间窗口的时间作为“key”值,形成〈key,value〉元组,用于构建外层 B+树。B+树索引能够实现等值查询和范围查询,而且相比于其他树形索引,在相同的数据规模下,B+树的层高更小。

本文贡献如下:

1)提出了多次微批量排序单次批量装载的 B+树构建方法。该方法采用创新的批量装载方式,提出先构建 B+树骨架再赋码,从而完成 B+树构建,构建速度快、时延低、能提供近实时查询。

2)量化分析了多次微批量排序单次批量装载的 B+树构建方法,在时间窗口固定、流元组匀速到达的理想情况下,分析到达速率与分片数之间的关系。

2 相关工作

文献[1]综述了数据流管理系统(DSMS)的构建方法,介绍了现有的一些数据流管理系统,如 OpenCQ,NiagaraCQ,Aurora 等。现有的这些数据流管理系统着重于连续查询、查询优化、语义分析、近似计算、资源管理等方面,系统吞吐量无法应对海量的大数据流,缺乏有效的数据流实时存储技术、数据流处理算法[8]。

合适的索引能够在大数据集中快速定位目标数据对象,如 AVL 树、B 树及其变种 B+树、T 树、T+树、哈希索引等索引结构。AVL 树是一种平衡二叉查找树,树内节点保存一个数据项和两个指针,该索引结构的存储空间利用率低;当数据量大时,AVL 树的层高较大,查找开销变大。B 树是一种适用于外查找的平衡多叉树,与 AVL 树索引相比,在相同的数据规模下,B 树具有层高较小的特点,适用于外存索引,减少了单次查找磁盘 I/O 的次数。B+树是 B 树的优化变种,具有优化的范围查找的功能。B+树也被许多研究人员用于内存索引,在大数据量的情况下,B+树层高小的特点能够减少数据访问的层数,减少查询处理的开销。

内存数据库技术被应用于数据流的处理,满足数据流应用要求实时处理的需求。T 树和 T^* 树是专门为内存数据库设计的内存索引结构,不仅具有 AVL 树的二叉查找的功能,同时还具有 B 树的优良更新和存储的特点。T^* 树是 T 树的优化,用以优化范围查询的功能。Lu 等人[9]对 B+树和 T 树进行了性能分析,结果表明 B+树的性能优于 T 树,其原因是数据量相同的情况下,T 树的层高更高,数据访问的时间更长;在并发情形下,B+树的性能更胜于 T 树,原因是 T 树需要加更多的锁。哈希索引则是通过哈希函数映射键值到数组来存取数据对象的一种方法,平均存取开销为 O(1),它适用于等值查找,不适合范围查找。

为了加快索引的构建速度,研究者提出了批量装载的技术。Bercken 等人[10]将现有的批量装载技术分为 3 类,分别为基于排序的批量装载技术[11]、基于样本的批量装载技术[12]、基于缓存的批量装载技术[13]。基于排序的批量装载技术的核心思想是先在内存中将数据进行排序,然后以自底向上的方式构建树形索引结构,如 B+树索引结构。在 B+树索引结构中,除了最右端的节点,每一层的节点都能够被填满数据。基于样本的批量装载技术的核心思想是先对数据集进行采样,之后根据采集的样本生成多个代表集,然后根据代表集生成相应的索引结构,最后,将剩下的数据被分配到某一个合适的代表集中。基于样本的批量装载技术的不足之处在于:有些装载好的索引节点中存在数据分布不均匀的现象,会导致索引结构的不平衡。

基于缓存的批量装载技术[13]是针对 R 树提出的基于缓存的批量更新机制,其核心思想是为树中除根节点以外的每一个中间节点分配缓存,每一条记录的插入并不是直接插入到目标位置中,而是先添加到根节点下一层的中间节点缓存中,等到缓存中的记录数超过设定的阈值后,缓存中的一部分记录被转移到下一层缓存中,在经过多次逐层转移后,记录才被添加到相应的叶节点上。Jermaine 等人[14]在 B+树的基础上,结合缓存的思想,提出了 Y-tree 的索引结构,实现了B+树上的批量插入。Y-tree 的每一个中间节点根据索引键的范围都设置有多个缓存。当一条记录从根节点插入到 Y-tree 中时,先将其存储在根节点相应的缓存中。当该缓存中的记录数超过设定的阈值时,批量地将缓存中的记录推送到下一层中间节点,依次逐层进行,直到记录到达相应的叶子节点,完成插入工作。基于缓存的批量装载技术的不足之处在于实现的复杂度较高。

针对流数据索引问题,Kholgh 等人[15]对位图索引、基于滑动窗口的索引、小波索引、时间索引和多粒度索引 5 种数据流索引模型进行了比较和分析。位图索引适用于传感器网络等应用中,被用来处理异构数据流,通过构建数据流的概要结构来提高空间利用率。基于滑动窗口的索引适用于用户对数据的感兴趣程度随时间递减的应用中,如搜索引擎和金融应用只对最近某段时间范围内的数据流感兴趣,并且对该时间范围有严格的要求。该模型可以减少存储开销,但是在磁盘存储和在线更新上存在一定的问题。小波索引实现规定时间范围内旧数据的快速移除和新数据的快速添加。首先将数据划分为多个范围,然后为每一个范围内的数据构建一个索引,以保证后续新数据的更新只会影响一个索引,从而加快索引的更新速度。Shivakumar 等人[16]提出了 6 种小波索引的构建和维持算法,也为各个算法分别提供了 3 种合适的更新方法。时间索引通过在查询中周期地执行检查点来构建索引,适用于已经按照有效的时间属性排好序的应用和数据流短时的管理系统。多粒度索引对构成查询的部分强加内部限制,进行特征提取,对高解析度的特征和低解析度的特征进行计算。相比于上述的数据流模型,多粒度索引具有最好的时间和空间有效性。

Fusco 等人[6]对网络数据包流的特殊应用提出了一种实时位图索引。该方法先采用联机局部敏感散列对记录进行重新排序,将内容相近的记录放到一起,然后采用窗口的技术对重排序的记录进行划分;记录按列进行存储,实时地创建位图索引,然后在创建位图索引时,再尝试对新产生的和已经存在的位图索引进行归并,进一步压缩位图索引。Pu 等人[7]针对

传感器网络中异构数据流索引和归档，提出了 ArQSS 数据流索引系统。ArQSS 采用基于位图的索引，弥补了关系型数据库中表存储和将记录序列化存储到平面文件这两种方式在空间利用不足和查询性能不高方面的缺点。通过使用控制反馈和在线的结构化聚集机制，ArQSS 系统能够自动地调整系统参数来自适应地提高系统性能。

在上述的工作中，位图索引只是以概要结构的形式保存数据流，并没有实现数据流的完整存储；小波索引虽然在数据流更新方面提出了合适的算法，但此类更新算法舍弃了较老时间范围的数据流以插入时间较新的流数据，适用于近似计算。本文选用基于排序的批量装载技术，提出先构建 B+树骨架再赋码来完成 B+树的构建，该方法能够完整地保存数据流，实时地构建索引，并提供近实时查询。

3　数据流索引

本文所建 B+树结构如图 1 所示，其节点 BTNode 包含关键字数组、指向父节点指针、(若为内部节点)指向子节点指针数组以及(若为叶节点)指向实际流元组的指针数组。

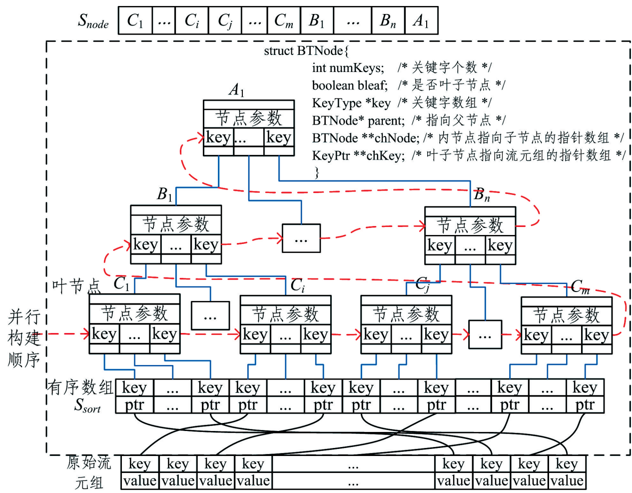

图 1　B+树结构及其骨架示意图

Fig. 1　Structure of B+tree and its skeleton

3.1　问题陈述

数据流流元组不断地到达，具有潜在规模无限、连续、实时的特点。面向时间窗口的批量构建内存 B+树索引的方法需要针对数据流特点，提高吞吐、降低时延，快速完成 B+树索引构建，并提供近实时查询。

图 2 中标记为 $W_1, W_2, \cdots, W_k$ 的矩形表示接收流元组的时间窗口，T_w 是窗口时长；标记为 $B_1, B_2, \cdots, B_k$ 的矩形表示构建 B+树索引的时区，其中构建时延表示每个时间窗口流元组缓存完成后完成 B+树构建需要的时间，用 T_{Delay} 表示。为提供近实时查询，T_{Delay} 越短越好。构建间隔表示当前时间窗口 B+树索引构建完成，下一时间窗口开始构建的时间间隔(也是 B+树可用时延)，用 T_{wait} 表示。若 T_{wait} 大于或等于 0，则表示下一时间窗口到达后，无需等待，可直接开始构建，则 B+树持续构建，系统达到稳定状态。若 T_{wait} 小于 0，则表示下一时间窗口到达后，需等待当前时间窗口完成构建后再进行构建；随着时间窗口数据流元组的持续缓存，等待时间不断累加，构建过程出现堵塞，导致系统崩溃。因此构建方法需要保证 T_{wait} 大于或等于 0 才能提高系统高吞吐，从而保证系统稳定运行。

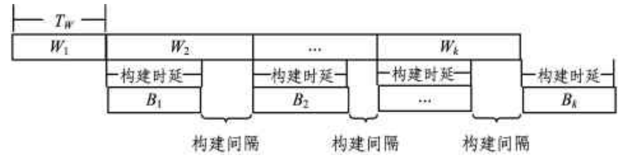

图 2　时间窗口接收数据流元组构建 B+树索引的模型

Fig. 2　Time-window based B+tree construction model by accepting data stream tuples

3.2　解决方案

本文采用基于排序的批量装载技术构建 B+树，其构建过程分为排序与批量装载两部分。批量装载构建 B+树需要流元组排序完成后，再自底向上并行装载，其中排序的时间复杂度最小为 $O(n\log n)$，其中 n 为排序元组数。在并行装载过程中，线程之间可能需要同步，降低了构建性能。为此，要尽可能最小化构建时延 T_{Delay}，同时最大化构建间隔 T_{wait}。本文结合应用场景，利用 B+树的结构特点来优化这两项指标，主要的优化思想是重叠运算与避免不必要的封锁。

所提方案是基于多次微批量排序单次批量装载的 B+树构建方法。该方法采用多次分片排序、一次归并的方法完成窗口流元组排序；在流元组全部到达后，计算 B+树结构参数，并根据计算结果统一申请内存资源，然后自底向上、自左向右并行批量装载节点。最后在排序完成后，自底向上、自左向右并行操作节点，从其子节点或所属流元组获取码。具体构建过程如图 1 所示，构建阶段如图 3 所示。

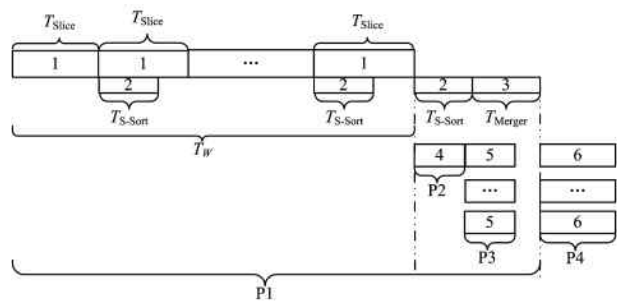

图 3　采用基于多次微批量排序单次批量装载的 B+树构建方法的描述

Fig. 3　Construction process description of MBSortSBLoad

该方法将构建过程分为排序(P1)、B+树结构参数计算(P2)、并行 B+树骨架构建(P3)和批量装载(P4) 4 个阶段。

1)排序阶段

排序阶段(P1)将时间窗口等时间分片，用标示 1 的矩形表示，其中 T_{Slice} 表示每个分片的时长，每个分片完成缓存后即采用快速排序法完成分片排序。用标示 2 的矩形表示排序期间，$T_{\mathrm{S\text{-}Sort}}$ 表示分片的排序时间，T_{Slice} 一般大于 $T_{\mathrm{S\text{-}Sort}}$；同时继续下一分片的接收。整个时间窗结束时，形成了多个有序分片，此时可以进行归并排序，由标示 3 的矩形表示。

2)B+树结构参数计算阶段

B+树结构参数计算阶段(P2)在时间窗口完成缓存后，结合 B+树结构进行计算，由标示 4 的矩形表示，计算数值包括树高、内节点个数、叶子节点个数及每个节点的子节点数等

参数(见表1)。根据计算结果,可知树的节点总数 $N_{\text{node-sum}}$(内节点数+叶子节点数),然后在内存申请长度为 $N_{\text{node-sum}}$ 的节点数组,完成内存资源申请。

表1 B+树结构参数表

Table 1 Parameters of B+tree

参数	说明
B	节点容量(可容纳的子节点数)
W	窗口流元组数
H	树的层数
N_{iNodes}	内节点数
N_{lNodes}	叶子节点数

其中根据B+树的结构特征计算树结构,已知流元组数为 W 时,计算如下参数:

①B+树的层数 H,满足 $B^{H-1}<W\leqslant B^{H}$,$H=\lceil\log_B^W\rceil$

②内节点数 N_{iNodes},$N_{\text{iNodes}}=\sum_{i=0}^{H-2}B^i$

③叶子节点数 N_{lNodes},$N_{\text{lNodes}}=B^{H-1}$

若流元组数 W 不是 B 的整数倍,则最后的叶节点未存满。由于流数据索引不发生更新,因此在构造过程中除了最后一个叶节点,其余节点全是满的。

通过上述计算,得到B+树的节点数 $N_{\text{iNodes}}+N_{\text{lNodes}}$、每个节点的子节点数、流元组数,在内存中分配节点空间,可以开始B+树构建的下一阶段。

3)B+树骨架并行构建阶段

在B+树骨架并行构建阶段(P3)进行自左向右、自底向上的并行构建B+树骨架,直至装配好根节点,其构建顺序见图1中虚线指示的顺序。B+树是典型的平衡树,各节点间通过指针指向形成树。当窗口时间到期且流元组数量已知时,根据时间窗口流数据特征,以及上下两级B+树的场景(第一级B+树将窗口时间作为码值,第二级B+树将元组关键字作为码值),后续时间段到达的元组不会更新到早期第二级B+树中,B+树节点的装填率可以设为100%,结合B+树结构特征,可以计算出要构建的树的大小、各节点在树结构的位置及上下级关系,构建出B+树的型(此时不含具体的码值),本文称之为B+树骨架。因此,B+树骨架可以在归并完成前就开始提前构建。

B+树骨架中各节点的码在流元组数组归并排序完成后通过B+树指针结构从流元组中获取,具体实例参见图1。图1中已知存放指向流元组的〈key,ptr〉有序数组 S_{sort} 的地址空间,即使其排序结果未存入,也可以先计算B+树的结构参数,分配B+树所需空间,令 S_{node} 表示根据B+结构参数计算后分配的全部B+树节点的内存空间,由图1可知,将B+树的节点自底向上、自左向右依次存放在 S_{node} 数组中,则每个节点根据位置关系即可计算出其子节点或有序数组 S_{sort} 的位置,完成指针指向,其中叶子节点通过指针指向有序数组 S_{sort} 的地址空间。在骨架构建时,采用多任务并行装载数组 S_{node} 中的节点即可完成B+树的骨架构建。待归并排序结果存入 S_{sort} 后,再自底向上操作每一个节点,通过指针从数组 S_{sort} 中获取每个节点码,批量装载B+树,从而完成B+树构建。

B+树骨架并行构建阶段由图3中标示为5的矩形表示,多个标示为5的矩形表示多任务并发。构建时初始化节点参数,完成节点与节点之间、叶子节点与流元组之间的链接操作,即通过指针指向来完成树的结构构建,该过程中节点未装填码。

4)B+树批量装载

待归并排序、B+树骨架构建完成后,可以开始B+树批量装载(P4)过程。类似骨架构建过程,批量装载过程在逻辑上按自左向右、自底向上操作节点获取码,由图3中标示为6的矩形表示,多个标示为6的矩形表示多线程并行,完成整棵B+树的构建。

基于多次微批量排序单次批量装载的B+树构建方法如算法1所示。由于涉及多任务并发执行,该算法采用了事件驱动方式进行描述。主要有两个事件:分片时间到期事件OnSliceEvent、窗口时间到期事件OnTWEvent。当发生On-SliceEvent事件时,算法对该分片进行排序,同时继续下一分片的数据接收,两上操作并行执行的。当发生OnTWEvent事件时,开始下一时间窗口并激发并发任务,开始对全部有序分片进行归并排序,同时进行B+树结构参数的计算、B+树骨架构建,等待前两个任务结束后,开始B+树的批量装载。

算法1 MBSortSBLoad(多次微批量排序单次批量装载)

输入:缓存时间窗口流元组数组 S_W;窗口长度 T_W;分片长度 T_{slice};最大子节点数B

输出:B+树的root指针

1.初始化分片结构数组Slices,每个Slice元素是{start,end}元组对,记录该分片对应缓存 S_w 的起讫位置。

2.初始化分片计数s←1,Slices[s].start←S_w 当前位置。

3.接收数据流元组,将其缓存到时间窗 S_w 中,并进行定时检测:

3.1.每隔 T_{slice} 时间:

3.1.1.设置Slices[s].end←S_w 当前位置;

3.1.2.产生OnSliceEvent(Slices[s])事件;

3.1.3.s←s+1,Slices[s].start←S_w 当前位置+1。

3.2 每隔 T_w 时间:

3.2.1.保存当前时间窗 S_w'←S_w;

3.2.2.分配新的时间窗 S_w 用于缓存后续数据流元组;

3.2.3.产生OnTWEvent(S_w')事件。

并发任务:OnSliceEvent(Slice)

1.对分片Slice进行排序。

并发任务:OnTWEvent(S_w)

1.启动任务完成:等待最后一个分片排序完成,对已经分片的Slices进行归并排序,将结果放置于有序数组 S_{sort} 中。

2.启动任务完成:

2.1.ComputingParas(|S_w|);//计算该窗口B+树的结构参数,其中|S_w|为元组数

2.2.CreateB+TreeSkeleton()。//创建B+树骨架

3.BulkLoading()。//批量装载B+树,父节点从子节点、流元组中获取码

过程:ComputingParas()//计算B+树结构参数

1.B+树的层数H;

2.B+树内节点的子节点数m;

3.内节点数 N_{iNodes};

4.叶子节点数 $N_{\text{leafNodes}}$;

5.叶子节点的流元组数。

过程:CreateB+TreeSkeleton()//创建B+树骨架

1. 完成树结构参数计算后,初始化节点数组 S_{node},该数组长度为 $N_{iNodes}+N_{leafNodes}$;//数组前 $N_{leafNodes}$ 个为叶子节点,后 N_{iNodes} 个为内节点,其中最一个节点为根节点
2. BulkLoadingLeaves();//批量装载叶节点
3. BuildInnerStructure()。//构建内部节点骨架

过程:BulkLoading ()//批量装载B+树,父节点从子节点、流元组中获取码过程

1. GetKeyLeaves();//叶节点获取码
2. GetKeyInnerStructure();//内部节点获取码
3. S_{node} 最后一个节点为根节点,返回B+树根节点上的root指针。

过程:BulkLoadingLeaves() //批量装载叶节点

1. 叶节点为 S_{node} 的前 $N_{leafNodes}$ 个,当前待装载叶节点位置 i←1。
2. 启动多任务,每个任务完成:
   2.1. 取待装载叶节点 N_i← S_{node}[i],其中i不大于 $N_{leafNodes}$,i←i+1;
   2.2. 初始化 N_i,包括numKeys等属性值;
   2.3. 对ch=1,…,N_i.numKeys,依次将流元组指针 N_i.chKey[ch]指向 S_{sort} 对应的流元组。

过程:BuildInnerStructure()//构建内部节点骨架

1. S_{node} 内部节点从 $N_{leafNodes}+1$ 到 $N_{leafNodes}+N_{iNodes}$,当前待构建节点位置 i ← $N_{leafNodes}+1$。
2. 启动多任务,每个任务完成:
   2.1. 取待构建节点 N_i← S_{node}[i],其中i不大于 $N_{leafNodes}+N_{iNodes}$,i←i+1;
   2.2. 初始化节点 N_i,包括numKeys等属性值;
   2.3. 对于ch=1,…,N_i.numKeys,依次将子节点指针 N_i.chNode[ch]指向对应的节点。

过程:GetKeyLeaves () //叶节点获取码

1. 叶节点为 S_{node} 的前 $N_{leafNodes}$ 个,当前待获取码叶节点位置 i←1。
2. 启动多任务,每个任务完成:
   2.1. 取待获取码叶节点 N_i←S_{node}[i],其中i不大于 $N_{leafNodes}$,i←i+1;
   2.2. 对于ch=1,…,N_i.numKeys,依次遍历 N_i 的流元组 N_i.chKey[ch];
   2.3. N_i.key[ch] ←N_i.chKey[ch].key。//向上传播码值

过程:GetKeyInnerStructure()//内部节点获取码

1. S_{node} 内部节点从 $N_{leafNodes}+1$ 到 $N_{leafNodes}+N_{iNodes}$,当前待构建节点位置 i ← $N_{leafNodes}+1$。
2. 启动多任务,每个任务完成:
   2.1. 取待获取码内部节点 N_i← S_{node}[i],其中i不大于 $N_{leafNodes}+N_{iNodes}$,i ← i+1;
   2.2. 对于ch=1,…,N_i.numKeys,依次遍历 N_i 的子节点 N_i.chNode[ch];
   2.3. N_i.key[ch] ←N_i.chNode[ch].key[1]。//向上传播码值

4 并行批量构建内存B+树时间代价估算

本节从理论分析角度入手,对前面提出的算法MBSortSBLoad中涉及的内存B+树并行批量构建过程中的各参数进行理论分析,如参数间关联、参数阈值和参数临界值等,并给出一些结论。首先假定数据流流速和整个时间窗口大小为 V_S 和 T_W。表2对前面以及后续分析过程中用到的符号进行了汇总说明。

表2 符号表

Table 2 Notations

符号	说明
T_W	单个时间窗口的长度/s
V_S	数据流流速/(元组/s)
T_{CPU}	CPU一次简单计算的时间/s
$T_{RM}=T_M$	读一次内存的时间/s
$T_{WM}=T_M$	写一次内存的时间/s
$N_{KP\text{-}Node}$	B+树中每个节点Key和Pointer对的个数
T_{Slice}	时间窗口内单个分片的长度/s
N_{Slice}	时间窗口内分片的个数
$T_{S\text{-}Sort}$	单个分片排序时间/s
T_{Merge}	时间窗口结束时所有分片数据的归并排序时间/s
$T_{CompTree}$	计算B+树参数所用时间/s
$T_{CreatTree}$	构建B+树骨架的时间/s
$T_{SetTree}$	批量装载B+树的时间/s
T_{Delay}	时间窗口结束至对应B+树可用的间隔/s

在进入正式分析之前,先讨论参数内存读写时间 T_{RM} 和 T_{WM},以及CPU一次简单计算的时间 T_{CPU}。对于内存读写时间 T_{RM} 和 T_{WM},可以近似认为 $T_{RM}=T_{WM}=T_M$。对于现有CPU,存在 $T_{CPU}\ll T_{RM}$,即CPU单次简单计算相对于内存读写时间来说可以忽略不计。因此,在理论分析过程中合并 T_{RM} 和 T_{WM} 为统一 T_M,同时忽略CPU的计算时间。

4.1 时间窗口分片数 N_{Slice}

对于算法MBSortSBLoad,时间窗口内单个分片的长度 T_{Slice} 需要满足一个前提:在下一个分片元组全部到达前,前一分片的元组已完成排序。这样可以保证分片缓存元组与排序重叠,避免排序慢于数据流到达的情况以及计算能力不足的问题。即 T_{Slice} 和 $T_{S\text{-}Sort}$ 需要满足如下关系:

$$T_{Slice}\geqslant T_{S\text{-}Sort} \tag{1}$$

分片数据排序采用了快速排序,最坏情形的时间复杂度为 $O(n^2)$,n 是元组数;假设数据流到达速度 V_S 是均匀的,则一个分片内的元组数为分片时长与流速的乘积,即 $T_{Slice}\times V_S$,因此单个分片内数据的排序时间 $T_{S\text{-}Sort}$ 为:

$$T_{S\text{-}Sort}=7\times T_M\times(T_{Slice}\times V_S)^2 \tag{2}$$

结合 $T_W=N_{Slice}\times T_{Slice}$,可知:

$$T_{S\text{-}Sort}=\frac{7\times T_M\times T_W^2\times V_S^2}{N_{Slice}^2} \tag{3}$$

其中,$7\times T_M$ 为快速排序单次循环所需的时间。

由式(1)和式(2)可得:

$$T_{Slice}\geqslant 7\times T_M\times(T_{Slice}\times V_S)^2 \tag{4}$$

结合 $T_{Slice}=T_W/N_{Slice}$,由式(4)可知:

$$N_{Slice}\leqslant\frac{1}{7\times T_M\times T_W\times V_S^2} \tag{5}$$

由式(5)可知,滑动窗口内分片数 N_{Slice} 有上限。

4.2 时间窗口内B+树可用的时延 T_{Delay}

算法中最后一个分片的排序及与之前已排序分片数据的归并需要按顺序执行。同时,根据时间窗口内数据量计算B+树参数、分配空间并构建B+树骨架也按顺序执行。但上述两组操作可以并行完成,最终耗时取两组耗时较长者。当两组操作全部完成后,可以开始B+树批量装载时间窗口数据。据此,T_{Delay} 与 $T_{S\text{-}Sort}$,T_{Merge},$T_{CompTree}$,$T_{CreatTree}$ 及 $T_{SetTree}$ 满足如下关系:

$$T_{\text{Delay}}=\max\{T_{\text{S-Sort}}+T_{\text{Merge}},T_{\text{CompTree}}+T_{\text{CreatTree}}\}+T_{\text{SetTree}} \tag{6}$$

其中，通过式(2)来计算最后一个分片的排序时间 $T_{\text{S-Sort}}$。时间窗口内全体分片(N_{Slice}个)的归并排序时间 T_{Merge}(按最坏情况估算)的估算公式如下：

$$T_{\text{Merge}}=2\times T_M\times N_{\text{Slice}}\times T_W\times V_S \tag{7}$$

其中，$T_W\times V_S$ 为时间窗口元组数，$2\times T_M$ 为读取一次内存和写回一次内存所用的时间。计算 B+树结构参数的计算耗时 T_{CompTree}只涉及 CPU 操作，因此可以忽略不计。

创建 B+树骨架的过程是由最底层逐层向上创建，所用时间 $T_{\text{CreatTree}}$的计算公式如下：

$$\begin{aligned}T_{\text{CreatTree}}&=\frac{6\times T_M\times T_W\times V_S}{N_{\text{KP-Node}}}+T_M\times T_W\times V_S+\\&\frac{6\times T_M\times T_W\times V_S}{N_{\text{KP-Node}}^2}+\frac{T_M\times T_W\times V_S}{N_{\text{KP-Node}}}+\\&\frac{6\times T_M\times T_W\times V_S}{N_{\text{KP-Node}}^3}+\frac{T_M\times T_W\times V_S}{N_{\text{KP-Node}}^2}+\cdots+\\&\frac{6\times T_M\times T_W\times V_S}{N_{\text{KP-Node}}^H}+\frac{T_M\times T_W\times V_S}{N_{\text{KP-Node}}^{H-1}}\\&=\frac{6\times T_M\times T_W\times V_S\times(N_{\text{KP-Node}}^H-1)}{(N_{\text{KP-Node}}^{H+1}-N_{\text{KP-Node}}^H)}+\\&\frac{T_M\times T_W\times V_S\times(N_{\text{KP-Node}}^H-1)}{(N_{\text{KP-Node}}^H-N_{\text{KP-Node}}^{H-1})}\end{aligned} \tag{8}$$

其中，H 代表 B+树的高度，$6\times T_M$ 为 B+树节点 6 个参数赋值的写内存时间，T_M 为计算 Pointer 值并赋值给 B+树 Pointer 域的写内存时间。批量装载 B+树所用时间 T_{SetTree}的计算公式如下：

$$\begin{aligned}T_{\text{SetTree}}&=2\times T_M\times T_W\times V_S+\frac{2\times T_M\times T_W\times V_S}{N_{\text{KP-Node}}}+\cdots+\\&\frac{2\times T_M\times T_W\times V_S}{N_{\text{KP-Node}}^{H-1}}\\&=\frac{2\times T_M\times T_W\times V_S\times(N_{\text{KP-Node}}^H-1)}{(N_{\text{KP-Node}}^H-N_{\text{KP-Node}}^{H-1})}\end{aligned} \tag{9}$$

其中，$2\times T_M$ 为读取码并赋值给 B+树 Key 域的读写内存时间。

由式(3)、式(7)和式(8)可知，$T_{\text{S-Sort}}+T_{\text{Merge}}>T_{\text{CompTree}}+T_{\text{CreatTree}}$，因此 T_{Delay}的计算公式如下：

$$\begin{aligned}T_{\text{Delay}}&=T_{\text{S-Sort}}+T_{\text{Merge}}+T_{\text{SetTree}}\\&=\frac{7\times T_M\times T_W^2\times V_S^2}{N_{\text{Slice}}^2}+2\times T_M\times N_{\text{Slice}}\times T_W\times V_S+\\&\frac{2\times T_M\times T_W\times V_S\times(N_{\text{KP-Node}}^H-1)}{(N_{\text{KP-Node}}^H-N_{\text{KP-Node}}^{H-1})}\end{aligned} \tag{10}$$

5 实验评价

本节通过实验比较 MBSortSBLoad 与插入法构建 B+树、批量装载构建 B+树的构建时延，以考察 MBSortSBLoad 的构建性能。

5.1 实验环境

实验平台采用 Intel(R) Core(TM) i5-4590 四核处理器，3.30GHz 主频，16GB 内存(海力士 DDR3 1600 MHz)，希捷硬盘(ST1000DM003-1ER62，1TB，7200RPM)。操作系统为 Windows 7。

如本文第 1 节所述，本文采用 B+树的二级索引结构来构建索引，一级索引按窗口时间戳来索引，二级索引采用元组的码值来索引；另外，由于本文采用基于排序的方式构建B+树索引，对于带有重复性码值的元组而言，构建代价比不重复码值的元组数据集更小，因此实验采用的数据集均是唯一码值的人工数据元组(本文采用的是 TPC-H 产生的〈key，value〉元组)。为模拟数据流，事先将产生的数据缓存到内存中，并按照指定的流速从内存读取数据来模拟数据流的流入。

实验考察数据流相关参数对于所提算法 MBSortSBLoad 性能的影响：流速、数据量、时间窗口和分片数，其中数据量和时间窗口的大小决定了平均流速。根据式(1)的要求，时间窗口内所能接受的数据量(最大流速)是有上限的。本文限于篇幅不讨论由于数据流猝发性而带来的复杂构建时延情况。

实验主要从以下几个方面展开：所提算法构建 B+树的性能；所提算法与插入法构建 B+树、批量装载构建 B+树的构建时延比较；所提算法所能承受的最大流速。

5.2 B+树构建性能分析

首先，针对算法 MBSortSBLoad，时间窗口设置为 30s，数据量为 1 亿条，即数据流平均流速为 1 亿条/30s。通过分片数变化观察利用算法 MBSortSBLoad 构建 B+树的性能，主要包括时间窗口对应 B+树构建时延 T_{Delay}、所有数据到来之后的排序时间 $T_{\text{Sort}}=T_{\text{S-Sort}}+T_{\text{Merge}}$、构建 B+树骨架的时间 $T_{\text{CreatTree}}$与为 B+树赋值的时间 T_{SetTree}，实验结果如图 4 所示。

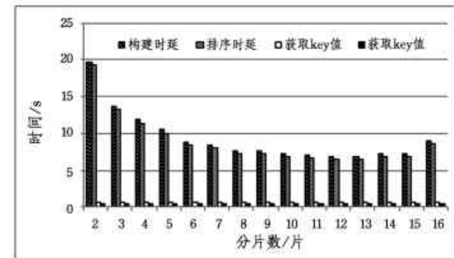

图 4 MBSortSBLoad 算法分片数与构建时延的关系

Fig. 4 Relationship between number of slices and build delay of MBSortSBLoad algorithm

根据图 4 所得的实验数据计算可得，T_{Sort}在 T_{Delay}中的占比维持在 95%左右(不同分片数时有小范围波动)，$T_{\text{CreatTree}}$与 T_{SetTree}在 T_{Delay}中的占比均为 5%左右，印证了式(6)－式(10)，即在使用算法 MBSortSBLoad 构建 B+树时，最终影响 B+树在窗口结束之后多久才能进行查询的关键因素为最后的排序时间 T_{Sort}，即构建时延 T_{Delay}中占支配地位的是 T_{Sort}。对于排序，其时间复杂度为 $O(n^2/m+(m-1)\times n$，其中 n 为数据量，m 为分片数。如果 n 不变，随着 m 的增加，排序时间曲线具有先降后增的趋势，这与图 4 显示的排序时间 T_{Sort}的实验结果一致。既然 T_{Delay}由 T_{Sort}决定，那么 T_{Delay}同样具有类似趋势，这一点在图 4 中也得到了验证，即在时间窗口和数据量一定的情况下，随着分片数的增加，B+树的可用时延先降低后缓慢增加。当数据流时间窗口内数据量为 1 亿条，且构建 B+树的可用时延最优情况出现在分片数为 13 时，可用时延为 6.813s。

为了进一步分析在时间窗口不变而数据量变化(即数据流流速变化)的情况下，分片数与树构建时延的关系，分别设

置时间窗口大小为 30s，60s，90s，120s，测试数据量为 1 亿条、5000 万条和 1000 万条，可得到使用算法 MBSortSBLoad 构建 B+树的构建时延随着分片数逐渐增加的变化情况，实验结果如图 5 所示。注意，由于给定了数据量(流速)和数据分片，对于不同的时间窗口大小，其构建时延是相同的，因此只呈现了一幅图。

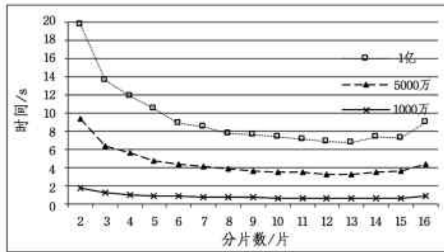

图 5　MBSortSBLoad 不同流速、分片数与构建时延的关系

Fig. 5　Relationships among different stream speeds，slices and building delay for MBSortSBLoad

由图 5 可知，无论数据量为 1 亿条、5000 万条或 1000 万条，利用算法 MBSortSBLoad 构建 B＋树的构建时延都会随着分片数的递增呈现先降再增的趋势，且在分片数为 13 时达到最优。

5.3　MBSortSBLoad 与插入法、批量装载法的比较

为了比较所提算法与插入法、批量装载法的性能，将时间窗口设置为 30s，分片数设置为 13(此分片数使所提算法的构建性能最优，见图 5)。通过改变数据量，比较所提算法与其他两者的构建时延差异，所提算法与插入法的比较结果如表 3 所列。

表 3　MBSortSBLoad 与插入法的构建时延比较

Table 3　Comparison of build delay between MBSortSBLoad and insert method

数据量/万条	MBSortSBLoad/s	插入法/s
100	0.063	33.099
200	0.125	67.110
300	0.187	103.329
400	0.250	137.219
500	0.313	173.470
600	0.391	212.267

由表 3 的对比可知，所提算法的构建时延明显优于插入法。显然，逐条插入的代价是昂贵的。

图 6 给出了两者在不同数据量下的构建时延，所提算法的构建时延显著低于普通批量装载技术的构建时延。且随着数据量的增大，优势更加明显。这是因为所提算法将 B+树骨架构建与排序并行执行，同时优化了批量装载时的构建顺序。

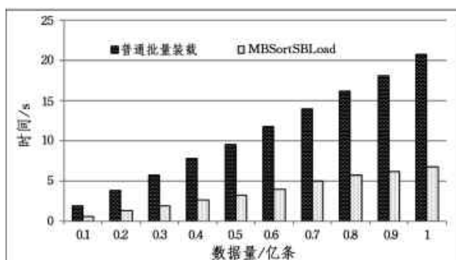

图 6　MBSortSBLoad 与批量装载法的构建时延比较

Fig. 6　Comparison of build delay between MBSortSBLoad and BulkLoad

5.4　算法所能承受的数据流最大流速

为测试 MBSortSBLoad 能够承受的最大流速，将时间窗口设置为 30s，分片数设置为 13，通过控制数据量来改变数据流速。因为时间窗口一定时，MBSortSBLoad 分片数是有上限的(见式(5))，即单个分片内接收数据进行快速排序的耗时不能大于分片时长。相反，如果分片数和时间窗口大小固定，那么数据量就会受到限制，从而导致流速也受到限制。MBSortSBLoad 的分片内快速排序时间随着数据量的增长而变化的趋势如图 7 所示。

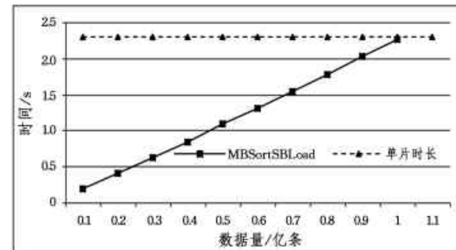

图 7　MBSortSBLoad 片内快速排序时延随数据量的变化趋势

Fig. 7　Trend of intra-slice sorting-time in MBSortSBLoad

由图 7 可知，MBSortSBLoad 在数据量为 1 亿时，与单个分片时长(基准线)相交，且数据量大于 1 亿时，系统崩溃，因此 MBSortSBLoad 在时间窗口为 30s 且分片数为 13 的情况下，最大承载流速为 330 万条/秒。

结束语　本文结合数据流的特点，针对数据实时性要求高的场景，提出了多次微批量排序单次批量装载(MBSortSBLoad)构建 B+树。该方法构建速度快，能承载的最大流速大，适用于高流速的场景。今后将进一步研究适用于分布式场景的 MBSortSBLoad 构建方案。

参 考 文 献

[1] BABCOCK B，BABU S，DATAR M，et al. Models and issues in data streams[C]// Proceedings of PODS Symposium. 2002：1-16.

[2] ARASU A，BABCOCK B，BABU S，et al. Stream：The Stanford data stream management system[M]. Springer：Data Stream Management，2016：317-336.

[3] CHANDRASEKARAN S，COOPER O，DESHPANDE A，et al. TelegraphCQ：Continuous dataflow processing[C]// Acm Sigmod International Conference on Management of Data. 2003：668.

[4] CARNEY D，CETINTERNEL U，CHERNIACK M，et al. Monitoring streams-a new class of DBMS applications[C]// Proceedings of VLDB Conference. 2002：215-226.

[5] ZHANG D D，LI J Z，WANG W P，et al. Algorithms for Storing and Aggregating Historical Streaming Data[J]. Journal of Software，2005，16(12)：2089-2098. (in Chinese)
张冬冬，李建中，王伟平，等. 数据流历史数据的存储与聚集查询处理算法[J]. 软件学报，2005，16(12)：2089-2098.

[6] FUSCO F，VLACHOS M，STOECKLIN M P. Real-time creation of bitmap indexes on streaming network data[J]. The VLDB Journal，2012，21(3)：287-307.

(下转第 212 页)

REST 服务和 SOAP 服务提取相关特征，从服务描述信息、领域标签、QoS 信息等层面对服务进行相似度计算，进一步结合服务的社交属性，采用 GSSN 算法将计算结果进行聚类处理，划分服务簇，以展现各服务在全局环境下的服务社交关系。最后，以 PWeb 等系统上真实的服务集进行实验，实验结果显示该方法能够有效改善 Web 服务聚类效果及提高服务聚类的精度，具有较好的实际应用价值。

在接下来的工作中，我们将在现阶段研究的基础上进一步完善软件平台。在理论上，也还有许多问题有待于进一步解决。例如，在动态环境下，还需研究基于聚类后的全局社交服务网实现最优 QoS 的服务发现。此外，当某个服务无法正常工作时，在 GSSN 环境下，如何实现相同服务簇内最优服务的自动推荐也将是今后的研究重点。

参考文献

[1] AL-MASRI E, MAHMOUD Q H. Investigating Web Services on the World Wide Web[C]//International Conference on World Wide Web(WWW 2008). Beijing, China, 2008: 795-804.

[2] CHEN W, PAIK I, HUNG P C K. Constructing a Global Social Service Network for Better Quality of Web Service Discovery [J]. IEEE Transactions on Services Computing, 2015, 8(2): 284-298.

[3] LI Z, WANG J, ZHANG N, et al. A Topic-Oriented Clustering Approach for Domain Services[J]. Journal of Computer Research and Development, 2014, 51(2): 408-419. (in Chinese)
李征，王健，张能，等. 一种面向主题的领域服务聚类方法[J]. 计算机研究与发展，2014，51(2)：408-419.

[4] TIAN G, HE K Q, WANG J, et al. Domain-Oriented and Tag-Aided Web Service Clustering Method[J]. Acta Electronica Sinica, 2015, 43(7): 1266-1274. (in Chinese)
田刚，何克清，王健，等. 面向领域标签辅助的服务聚类方法[J]. 电子学报，2015，43(7)：1266-1274.

[5] LIU W, WONG W. Web service clustering using text mining techniques[J]. International Journal of Agent-Oriented Software Engineering, 2009, 3(1): 6-26.

[6] CHERIFI C, LABATUT V. Web Services Dependency Networks Analysis[C]//International Conference on New Media and Interactivity. 2010: 115-120.

[7] GUO F, WEI G, DENG M M, et al. Service Oriented Petri Net Model and It's Structural Operational Semantics[J]. Journal of Chinese Computer Systems, 2013, 34(12): 2739-2743. (in Chinese)
郭峰，魏光，邓蒙蒙. 一种面向服务 Petri 网模型及其结构化操作语义[J]. 小型微型计算机系统，2013，34(12)：2739-2743.

[8] WANG X, WANG Z, XU X. Semi-empirical Service Composition: A Clustering Based Approach[C]//IEEE International Conference on Web Services (ICWS 2011). Washington DC, USA, DBLP, 2011: 219-226.

[9] CHEN L, WANG Y, YU Q, et al. WT-LDA: User Tagging Augmented LDA for Web Service Clustering[M]// Service-Oriented Computing. 2013: 162-176.

[10] KUMARA B T, PAIK I, KOSWATTE K R. Ontology learning with complex data type for Web service clustering[C]//IEEE Symposium on Computational Intelligence and Data Mining (CIDM). IEEE, 2014: 129-136.

[11] NAYAK R, LEE B. Web Service Discovery with additional Semantics and Clustering[C]//IEEE/WIC/ACM International Conference on Web Intelligence. IEEE, 2007: 555-558.

[12] WU J, CHEN L, ZHENG Z, et al. Clustering Web services to facilitate service discovery[J]. Knowledge & Information Systems, 2014, 38(1): 207-229.

[13] XIE L L, CHEN F Z, KOU J S. Ontology-based semantic web services clustering[C]//2011 IEEE 18Th International Conference on Industrial Engineering and Engineering Management (IE&EM). IEEE, 2011: 2075-2079.

[14] KUMARA B T G S, YAGUCHI Y, PAIK I, et al. Clustering and Spherical Visualization of Web Services[C]//IEEE International Conference on Services Computing. IEEE Computer Society, 2013: 89-96.

（上接第 177 页）

[7] PU K Q, ZHU Y. Efficient indexing of heterogeneous data streams with automatic performance configurations[C]//Proceedings of Scientific and Statistical Database Management Conference. 2007: 34-34.

[8] JIN C Q, QIAN W N, ZHOU A Y. Analysis and Management of Streaming Data: A Survey[J]. Journal of Software, 2004, 15(8): 1172-1179. (in Chinese)
金澈清，钱卫宁，周傲英. 流数据分析与管理综述[J]. 软件学报，2004，15(8)：1172-1179.

[9] LU H, NG Y Y, TIAN Z. T-tree or B-tree: Main memory database index structure revisited[C]//Proceedings of Australian Database Conference. 2000: 65-73.

[10] BERCKEN J, SEEGER B. An evaluation of generic bulk loading techniques[C]//Proceedings of VLDB Conference. 2001: 461-470.

[11] CIACCIA P, PATELLA M. Bulk loading the M-tree[C]//Proceedings of Australasian Database Conference. 1998: 15-26.

[12] LO M L, RAVISHANKAR C V. The design and implementation of seeded trees: An efficient method for spatial joins[J]. IEEE TKDE, 1998, 10(1): 136-152.

[13] ARGE L, HINRICHS K H, et al. Efficient bulk operations on dynamic R-trees[J]. Algorithmica, 2002, 33(1): 104-128.

[14] JERMAINE C, DATTA A, OMIECINSKI E. A novel index supporting high volume data warehouse insertion[C]//Proceedings of VLDB Conference. 1999: 235-246.

[15] KHOLGHI M, KEYVANPOUR M R. Comparative evaluation of data stream indexing models[J]. International Journal of Machine Learning and Computing, 2012, 2(3): 257-260.

[16] SHIVAKUMAR N, GARCIA-MOLINA H. Wave-indices: indexing evolving databases[C]//Proceedings of SIGMOD Conference. 1997: 381-392.