

本科毕业设计说明书（论文）

**（2016届）**



论文题目 最优时间窗口覆盖查询算法设 计与实现

作者姓名 侯晨煜

指导教师 范菁

学科(专业)  计算机实验班

所在学院 计算机科学与技术学院

提交日期 2016年6月

摘要

本文拟解决时间数据库中的最优时间窗口覆盖查询问题，即给定一组用户和各自的有效时间区间（每个用户可能有多个有效时间区间），再给定一个时间窗口大小，查询既满足该时间窗口大小，又能够被最多用户覆盖的时间区间。对该问题的解决有着重要的现实应用，例如可以根据每个与会者的空闲时间安排最优的会议时间，也可以为广告商安排最佳的广告投放时间等。该问题的主要难点在于它的统计方式与传统COUNT聚合操作的统计方式不同，所以传统的时间聚合查询算法无法直接用于解决该问题。

针对最优时间窗口覆盖查询问题，本文提出了两种解决算法。第一种滑动时间窗口算法思路简单，在数据量小的情况下能够保持比较好的性能，但是当数据量比较大时，它的性能则无法令人满意。同时滑动时间窗口算法不支持增量计算。为了解决滑动时间窗口算法的缺陷，我们提出了第二种方法——基于SB\*-Tree的查询算法。SB\*-Tree是我们在SB-Tree的基础上，对SB-Tree的结点结构进行修改而产生的一种新的索引结构。相对于SB-Tree, SB\*-Tree增加了用于存储覆盖相应区间的目标集合的域。我们继承了SB-Tree的所有操作：插入、删除、分裂、合并，并做了相应的修改使之适应于SB\*-Tree。另外我们设计了SB\*-Tree的遍历方法以及区间调整策略来进行查询。

最后，本文利用真实数据集来进行实验，分析了不同因素对两种算法性能的影响，并比较了两种算法的性能优劣型。最终实验结果证明SB\*-Tree算法性能明显优于滑动时间窗口算法。

关键词：时间数据库，时间聚合，SB-Tree，最优查询

Abstract

This paper intends to solve the query of the optimal time window in temporal database, namely, given a set of users and their valid time intervals (each user may have multiple time intervals), then given a time window size, query time interval that not only meet the time window size, but also can be covered by most users . The solution to the problem has important practical applications, such as we can arrange the best meeting time based on the free time of each participant, also can arrange the best advertising time for advertisers and so on. The main difficulty of this problem is that statistical method of the problem is different from the traditional statistical method of COUNT aggregation operation, so the traditional time aggregation query algorithm cannot be directly used to solve the problem.

Aiming at the problem of the optimal time window covering query, two algorithms are proposed in this paper. The first algorithm sliding time window algorithm is simple, and it can maintain a relatively good performance in the case of small amount of data, but when the amount of data is large, its performance is not satisfying. At the same time, the sliding time window algorithm does not support incremental computation. In order to solve the weakness of sliding time window algorithm, we propose second methods -- SB\*-Tree based query algorithm.SB\*-Tree is a new index structure which is produced by modifying structure of SB-Tree node. Compared to SB-Tree, SB\*-Tree increases the domain to store targets set of corresponding interval. We have inherited all of SB-Tree's operations: insert, delete, split, merge, and make corresponding changes to adapt to the SB\*-Tree. In addition, we design the SB\*-Tree traversal method and the interval adjustment policy to query.

Finally, we use the real data to experiment, analyze the effects of different factors on the performance of the two algorithms, and compare the performance of the two algorithms. The experimental results show that the performance of SB\*-Tree algorithm is better than that of sliding time window algorithm.

Keywords：temporal database, temporal aggregation, SB-Tree, optimal query

目录

摘要 I

第一章 绪论 1

1.1 引言 1

1.2 研究的目的和意义 1

1.3 国内外相关研究现状 2

1.4 本文的主要工作 6

1.5 本文的组织结构 7

1.6 本章小结 7

第二章 预备知识 8

2.1 数据格式 8

2.2 时间粒度 8

2.3 时间窗口 9

2.4 问题定义 9

2.5 本章小结 10

第三章 滑动时间窗口算法 11

3.1 算法描述 11

3.2 算法详解 11

3.3 算法演示 13

3.4 算法复杂度 14

3.5 本章小结 17

第四章 基于SB\*-Tree的查询算法 18

4.1 SB-Tree 18

4.1.1 SB-Tree结点结构 18

4.1.2 插入 19

4.1.3 删除 19

4.1.4 结点分裂 20

4.1.5 区间合并 21

4.1.6 结点合并 22

4.2 SB\*-Tree 23

4.2.1 插入 23

4.2.2 删除 24

4.2.3 结点分裂 24

4.2.4 区间合并 25

4.2.5 结点合并 26

4.3 查询算法 27

4.3.1 获取中间结果 27

4.3.2 区间调整 27

4.3.3 获得最优区间 29

4.4 算法演示 29

4.5 本章小结 48

第五章 实验说明 50

5.1 数据说明 50

5.2 实验比较 52

5.2.1 时间窗口duration的影响 52

5.2.2 时间粒度granularity的影响 53

5.2.3 时间范围的影响 54

5.2.4 数据量的影响 55

5.2.5 参数b和l对SB\*-Tree算法的影响 56

5.2.6 算法性能比较 57

5.3 本章小结 58

第六章 总结 59

参考文献 60

致谢 62

附录 63

附录1 毕业设计文献综述 63

附件2 毕业设计开题报告 63

附件3 毕业设计外文翻译（中文译文与外文原文） 63

图目录

图 3. 1 滑动时间窗口算法时间线 12

图 3. 2 滑动时间窗口算法伪代码 12

图 3. 3 滑动时间窗口算法时间线 13

图 3. 4 表3-1示意图 14

图 4. 1 SB-Tree中间结点 18

图 4. 2 SB-Tree叶子结点 18

图 4. 3 SB\*-Tree中间结点........ 23

图 4. 4 SB\*-Tree叶子结点........ 23

图 4. 5 SB\*-Tree插入示意图 23

图 4. 6 SB\*-Tree删除示意图 24

图 4. 7 SB\*-Tree结点分裂示意图 25

图 4. 8 SB\*-Tree区间合并示意图 25

图 4. 9 SB\*-Tree结点合并示意图 26

图 4. 10 遍历SB\*-Tree伪代码 27

图 4. 11 区间调整伪代码 28

图 4. 12 表3-1内容构造SB\*-Tree步骤1：插入<Allen,[0,5]>和<Allen,[7,10]> 30

图 4. 13表3-1内容构造SB\*-Tree步骤2：插入<Allen,[11,14> 30

图 4. 14表3-1内容构造SB\*-Tree步骤3：插入<Bob,[3,5]> 31

图 4. 15表3-1内容构造SB\*-Tree步骤4：插入<Bob,[9,11]> 31

图 4. 16表3-1内容构造SB\*-Tree步骤5：插入<Cindy,4,12> 31

图 4. 17表3-1内容构造SB\*-Tree步骤6：插入<David,[1,7]> 32

图 4. 18表3-1内容构造SB\*-Tree步骤7：插入<David,[10,15]> 32

图 4. 19表3-1内容构造SB\*-Tree步骤8：插入<Eva,[2,9]> 32

图 4. 20表3-1内容构造SB\*-Tree步骤9：插入<Eva,[13,15> 33

图 4. 21表3-1内容构造SB\*-Tree步骤10：插入<Fay,[0,15]> 33

图 4. 22 SB\*-Tree类关系说明图 34

图 5. 1 原始数据内容 50

图 5. 2 阅读时间统计图 51

图 5. 3 用户阅读时间段内容 52

图 5. 4 时间窗口duration对算法性能的影响 53

图 5. 5 不同时间粒度下的算法运行时间图 54

图 5. 6 时间范围对滑动时间窗口算法影响图 54

图 5. 7 数据量对算法性能的影响 55

图 5. 8 参数b和l对SB\*-Tree性能影响图 56

图 5. 9 算法运行时间图 57

表目录

表 3 - 1人员空闲时间表.........................................................................................................................................14

表 3 - 2 扫描时间线的结果............................................................................................................................ 14

表 4 - 1 遍历SB\*-Tree的中间结果 28

表 4 - 2 区间调整后的结果 30

表 5 - 1 原始数据字段说明 50

# 绪论

## 引言

在传统的关系型数据库中，聚合操作主要用于计算一些列元组某些属性的总体特征。而在时间数据库中，每个元组都带有时间属性，时间聚合计算不仅考虑元组的某些显示属性，更重要地是考虑了元祖的时间属性。从查询需求上看，人们可能更加关注元组在某一时间上的总体特征[1]而不是元组各自具体的时间，例如：某在线阅读软件需要统计某一时段使用该软件读书的用户数量，而不需要关注每个用户读书的具体时间。所以在时间数据库领域中，时间聚合计算是非常重要的研究内容之一。

到目前为止，国内外已经对时间聚合开展了比较深入的研究，关于时间聚合的算法也有许多种。这将在接下来的研究现状中具体描述。但是，本文要研究的最优区间覆盖问题与传统的时间聚合问题有所不同。以传统时间聚合中的COUNT[2]为例，只要一个用户区间与待查询区间相交，则区间的COUNT值就加1。但是在最优时间窗口覆盖查询问题中，只有当一个用户区间完全覆盖待查询区间时，区间对应的用户数量才增加。这就导致了传统的时间聚合查询算法无法直接用于解决最优时间窗口覆盖查询问题。本文设计了新的索引结构和查询算法来支持最优窗口覆盖问题的计算，主要是通过对SB-Tree[3]的数据结构的改进，我们把新的数据结构叫作SB\*-Tree。SB\*-Tree在原来SB-Tree的结点结构上，增加了一个维度的信息，使得结点存储每个时间区间内的元组集合。接着再对SB\*-Tree进行遍历，获得中间结果，并对中间结果进行判断调整来获得最终结果。

## 研究的目的和意义

最优时间窗口覆盖问题不仅在时间数据库领域中是个新颖的问题，而且在现实生活中也有许多应用场景。若想要筹备一个朋友间的聚会，组织者通常会使用移动通讯应用（如微信）或短信进行沟通，再或者通过一些流行的社交网络（如人人网、facebook）或电子邮件。一些朋友表示没问题可以到，另一些则会建议组织者更换日期、时间或地点。然后组织者不停的更改询问确认、直到达成所有人或大部分人都认可的折中方案。除组织朋友聚会外，公司部门组织开会、团组旅游等场景下也涉及到类似的问题。在这些场景中，协调时间、寻找大家满意的活动地点成为活动组织者最头疼的事情，小规模四五个人的活动尚可解决，若组织大规模几十人、甚至成百上千人的大型活动，传统方式不仅效率低，且组织成本大。本研究能够提供一种高效的活动组织方式，只需要提供活动的持续时间以及参与人员各自的意向时间，算法就能够快速、自动地为活动安排举行时间，同时保证能有最多的人员参加活动。这对于传统的活动组织方式来说是一次巨大的创新，不仅能够大大节约了包括组织者在内所有参与人员的在活动组织时的时间花费，同时也保证了活动举行的最终质量，即有最多的人员来参加活动。

另外，在传统广告行业中，广告的投放成本花费很大程度上与广告的投放时间相关，投放时间越长，花费越高。毫无疑问，广告投放时间越长，就能够被更多的用户所关注，从而体现广告的价值。但是广告商为了能够得到最大的广告效益，往往希望在保证广告能够被尽可能多的用户看到的前提下减少广告投放时间，这样就能够最少的成本下获得最大的广告价值。为了满足广告商的需求，我们可以根据本研究算法找出用户最活跃的一个时间段来投放广告。该时间段的持续时间根据广告商的要求自行给定，另外一个特点是在该时间段活跃的用户是最多的。本研究可以根据用户的历史行为来找出满足上述两个条件的时间段，从而引导广告商投放广告来获得最大收益。

## 国内外相关研究现状

时间聚合分为两种类型：瞬时时间聚合和累积时间聚合。瞬时时间聚合是在某一时间点计算在仅当前时间下的有效元组的聚合值。而累积时间聚合与瞬时时间聚合不同的是，累积时间聚合还有一个额外的参数——窗口偏移量*w*。在计算时间点时的累积聚合时，需要将区间内有效的元组都考虑进去。

接下来将重点介绍当前国内外在时间聚合方面的主要研究。当前的时间聚合算法主要可以分为两类类：基于索引结构的和无索引结构的。无索引结构的聚合在计算时需要多次扫描数据库，接着在内存中生成一些可用于存储局部聚合结果的数据结构。这样的方式不利于增量式地计算。因为当数据库中的元组增加或减少时，原来内存中的数据不能有效反映真实数据库中的数据，所以需要重新对数据库进行扫描，构造新的数据结构来完成计算。基于索引结构的聚合计算是采用存储于外存的数据结构，通过对索引结构的操作来提高算法效率。

Kline和Snodgrass发明了一种叫作聚合树[4]的结构。聚合树支持实际聚合的增量计算。尤其是它们的段树的特征能够有效地处理长时间区间的元组。然而聚合树的一个缺点是它是一种内存的数据结构，作为数据库索引和数据仓库中的维护时间聚合的数据结构，这限制了它的高效性。另一个问题是聚合树是不平衡的。在最坏的情况下，从一个有n个元组的基本表中计算时间聚合，它将花费O(n2)的时间；处理一次插入的操作需要O(n)的时间；根据时间查找需要花费O(n)的时间。后来Kline提出了2-3树[5]。这个算法的思想是：首先，对关系中的元组按照其开始时间进行升序排序；接着，对于每个时间点，找到与这个时间点重叠的元组，然后计算相应的聚合值。在2-3树中，根结点和每个叶结点存储原始聚合信息，每个叶结点都代表了一个时间区间。因为2-3树是平衡树，所以它有较高的搜索效率，同时也便于调整树的形状。但是由于事先需要对各元组按时间排序，因此额外开销很大。2-3树处理聚合的时间复杂度为O(nlogn)。

M. Kaufmann针对于特别的SAP HANA数据库系统提出了一种能够支持时间聚合、时间遍历、和时态结合的统一的数据结构Timeline Index[6]。文中通过扫描两次数据库，建立Version Map和Event List来支持接下来的进一步操作。Kim J. S.提出了PA树[7]，PA树以AVL树为基础，每个结点存储时间戳和对应的局部聚合值，最后通过中序遍历PA树的到最终结果。

Moon B.提出了两种复杂度同样是O(nlogn)的算法，分别是计算COUNT的平衡树算法和计算MAX的归并排序聚合算法[8]。值得注意的是，虽然元组插入聚合树的顺序不会影响最终的结果，但是会很大程度上影响聚合树的性能。如果元组按照开始时间的升序排列，那么聚合树就更像是一个链表。因为对链表的插入的效率远远小于对平衡二叉树的插入的效率，所以聚合树的最坏情况的时间复杂度会达到O(N2)。 Moon B.提出了一种能够解决COUNT聚合的基于时间戳排序的方法。首先，将所有元组加载到内存中，从每个元组中提取出每一个时间戳。同时，每一个时间戳都对应着一个标记，表示该时间戳是元组的开始时间还是结束时间。然后，将这些时间戳按照递增的顺序排列。计算时，算法需要扫描排序好的时间戳和标记来得到最终结果。在扫描过程中，需要维护一个计数器，该计数器初始化为0。当用到一个开始标记时，计数器加一；遇到结束标记时，计数器减一。按照这样的方法，就可以得到最终结果。 但是这种方法仍然需要全部的元组的内存。所以，Moon B.提出了平衡树的算法，来优化有多个重复时间戳的情况。平衡树算法不需要在最初就加载整个数据库的内容，而是通过动态红黑树的插入来增量地对时间戳进行排序。平衡树的每个结点存储一个时间戳，但是不存储开始/结束的标记，而是存储两个计数器。其中一个计数器存储从该时间点开始的元组数量，另一个计数器存储在该结点结束的元组数量。平衡树算法虽然能够有效处理COUNT、AVG、SUM这一类聚合的有效算法，但是并不适合处理MAX、MIN这一类聚合。因为平衡数的结点只存储时间戳和相应的计数器，并不不记录在特定时间内有效的所有元组的信息，而这些信息对于计算MAX这类聚合是非常必要的。为了处理MAX这类聚合，Moon B.提出了归并排序的聚合算法。归并排序聚合算法类似于经典的划分-归并策略，将整个时间线划分为多个小段的时间区间，通过合并两个较小的时间区间的聚合结果，得到两者合并成的较大的时间区间的聚合结果，直到最后得到整个时间线的结果。他又针对大规模的数据设计了基于数据分区模式的连续且并行的桶算法[9]。

文章[10][11]还给出了一些并行计算时间聚合的算法。Yang Jun提出了一种能够支持增量计算和维护的索引结构SB-tree[3]。SB-tree是基于B-tree和segment-tree创造出来的树。它继承了B-tree的优点，能够很好的存储在硬盘上；同时它也继承了segment-tree的特征，能够有效处理长时间区间的元组的插入。SB-tree的每个中间结点存储了多个时间点和每个区间的聚合值，还有指向下一结点的指针；叶子结点也存储了多个时间点和每个区间的聚合值，但没有指向下一结点的指针。在SB-tree中有特殊的区间定义方法，并且在计算聚合值时需要从根开始到叶子，将整条路径上的聚合值都考虑在内。文章中还提出了SB-tree的其他变种，以适应不同的聚合类型。文章中提出用JSB-tree来支持支持任意窗口偏移量的累积SUM,COUNT,AVG聚合；用MSB-tree来支持任意窗口偏移量的累积MIN和MAX聚合。后来在SB-tree的基础上，Zhang d.等人结合了多维B-tree的特征，提出了MVSB-tree[12]的索引结构，能够处理任意键值范围的时间SUM，COUNT和AVG聚合查询。但是以上的这些方法大致都存在一下至少一个缺点：(1)需要大量的空间要求，(2)需要很高的处理时间(3)基于复杂的结构。这些缺点导致了无法在经济型数据库系统中很好的运用这些方法，所以Tao Y.等人针对那些可以接受小范围误差的情况，基于B-tree[13]和R-tree[14]，提出一种能够近似计算时间SUM和COUNT聚合的方法[15]。

另外，时间区间信息可以巧妙地转化为二维的平面点信息[16][18]。因为时间区间的结束点肯定大于区间的开始点，所以可以将一个区间转化为一个点(x,y)，其中x是区间的开始点，y是区间的结束点，且x<y，所以这个点肯定落在直线x=y的上方。当把时间区间信息转化成二维的平面点信息后，时间覆盖问题就转变成了三边矩形问题[17]。所以一些可以运用在空间中的范围查询数据结构也能够用在区间范围查询上。接下来将介绍几种空间索引结构。

四叉树[19]能够适应于2维数据，四叉树索引的基本思想是将空间递归划分为不同层次的树结构。它将已知范围的空间等分成四个相等的子空间，如此递归下去，直至树的层次达到一定深度或者满足某种要求后停止分割。四叉树的结构比较简单，并且当空间数据对象分布比较均匀时，具有比较高的空间数据插入和查询效率。四叉树对于区域查询，效率比较高。但如果空间对象分布不均匀，随着地理空间对象的不断插入，四叉树的层次会不断地加深，将形成一棵严重不平衡的四叉树，那么每次查询的深度将大大的增多，从而导致查询效率的急剧下降。

KD-Tree[20]是一种由二叉搜索树推广而来的用于多维检索的树的结构形式（K即为空间的维数）。它与二叉搜索树不同的是它的每个结点表示k维空间的一个点，并且每一层都根据该层的分辨器对相应对象做出分枝决策。顶层结点按由分辨器决定的一个维度进行划分，第二层则按照该层的分辨器决定的一个维进行划分...以此类推在余下各维之间不断地划分。直至一个结点中的点数少于给定的最大点数时，结束划分。KD-Tree的分辨器根据不同的用途会有不同的分辨器，最普通的分辨器为：n mod k（树的根节点所在层为第0层，根结点孩子所在层为第1层，以此类推）。

## 本文的主要工作

本文主要提出了两种解决最优时间窗口覆盖查询问题的算法。第一种滑动时间窗口算法思路比较简单。算法首先根据所有输入数据构造一条一维时间线，再构造符合时间窗口大小的窗口沿时间线滑动扫描。我们判断每个位置时的窗口与所有目标的覆盖情况，最终选择最优的时间窗口。该算法在数据量小的情况下能够保持比较好的性能，但是当数据量比较大时，它的性能则无法令人满意。同时滑动时间窗口算法不支持增量计算。为了解决滑动时间窗口算法的缺陷，我们提出了第二种方法——基于SB\*-Tree的查询算法。我们首先根据数据构造SB\*-Tree索引结构，再设计相应的查询算法查询索引结构来得到结果。其中SB\*-Tree是我们在SB-Tree的基础上，对SB-Tree的结点结构进行修改而产生的一种新的索引结构。相对于SB-Tree, SB\*-Tree增加了用于存储覆盖相应区间的目标集合的域。我们继承了SB-Tree的所有操作：插入、删除、分裂、合并，并做了相应的修改使之适应于SB\*-Tree。另外我们设计了SB\*-Tree的遍历方法以及区间调整策略来进行查询。

另外本文实验的数据来自于某一手机阅读软件的某一个月内用户所有行为的日志文件。本文从中提取了用户阅读数据的行为的相关信息作为最终实验的数据集。关于数据集的具体说明将在小节5.1中展示。

## 本文的组织结构

本文共分为六章，以“最优时间覆盖查询问题”为背景，主要提出了两种能够解决该问题的算法。并且通过实验详细地比较了两种算法的性能。各章内容如下：

第一章，介绍了课题研究的背景、目的和意义，国内外相关领域的研究，和本文的主要工作。

第二章，详细介绍了关于最优时间窗口覆盖问题的预备知识，并对该问题做出了正式定义。

第三章，重点介绍了扫描时间窗口算法，这是一种低效的解决最优时间窗口覆盖问题的算法。

第四章，介绍了另外一种高效的算法，基于SB\*-Tree的查询算法。

第五章，详细分析了影响两种算法性能的因素，并比较了两种算法的性能。

第六章，对本文的工作进行总结，并针对存在的问题提出下一步的工作。

## 本章小结

本章简要介绍项目的研究背景、在国内外相关领域的开发和应用现状以及项目的研究的任务和意义。最后，给出了本文的主要工作及本文的组织结构。

# 预备知识

在这一章中，我们首先介绍一些关于最优时间窗口覆盖问题的预备知识，这些知识有助于理解问题，主要包括数据对象、时间粒度、持续时间等概念。随后，我们给出时间窗口最优覆盖问题的正式定义。

## 数据格式

本文所研究的数据对象是时序数据类型，该数据可以存在于传统关系型数据库中，也可以来自用户行为日志当中。对于具体的数据主要包括有两种，即用户ID（UserID）和属于该用户的时间区间（Interval）。 UserID是每个用户的唯一标识，不同的用户在数据库中对应不同的UserID。 Interval是一个时间区间，由开始时间t\_start和结束时间t\_end组成，它表示用户在该区间内是“有效”的。注意，每个用户可以有多个有效时间区间。例如，一个用户在一天内会在多个不同的时间区间内阅读书籍。

## 时间粒度

时间粒度(Time Granularity)是描述上述时间区间的最小时间单位，即时间区间的开始和结束时间均是该时间粒度的整数倍，例如当时间粒度为5分钟时，开始或结束时间只能为1:00整，1:05分，1:10分等形式。在本文研究问题中，时间粒度可以选择任意度量，如1秒、1分、15分、1小时等。实际应用中，需要根据不同的场景，选择不同的时间粒度。在本文后续的算法实现中，时间粒度有着重要的作用。

## 时间窗口

时间窗口（Time Window）是指用户不间断参与某项活动的持续时间，例如用户从22:00至22:30持续进行了时间窗口为30分钟的阅读活动。时间窗口与用户时间区间的不同在于，前者是由查询请求者发出，而后者则是用户活动行为对应的时间序列。此外，时间窗口在本文工作中不规定具体的开始时间和结束时间，仅规定其窗口大小，即时间窗口仅用来表示时间区间的大小。因此，一个时间窗口可能会对应多个用户时间区间，例如当时间窗口为10分钟时，可以是从9:00到9:10分，也可以是1:00到1:10分。在本文研究中，时间窗口将用来进行与用户的具体时间区间进行匹配，例如我们想查询“一天中，用户在哪些时间区间持续阅读半小时以上”。

## 问题定义

基于上述基本概念，下面我们给出本文拟解决的问题正式定义：

定义 1（时间窗口最优覆盖问题）：给定一个用户集合U，该集合中的每个用户包含用户ID（UserID）和对应的多个时间区间（Interval\_List），每个时间区间（Interval）的大小表示为|Interval|=t\_end-t\_start。给定一个固定大小的时间窗口Time Window，将该Time Window与用户集合U中的每个用户时间区间进行匹配，找出该Time Window下的最优覆盖：

1）对于任一用户，若存在一时间区间|Interval|>Time Window，则认为该用户在该时间区间可覆盖时间窗口；

2）在条件1基础上，找出某一区间［start, end］，使得该区间对应的用户最多。

例如在阅读书籍的场景中，每个用户都会有对应的历史阅读的时间段，我们希望可以找出在持续一小时的时间，并且在该时间段内进行阅读的用户最多。

## 本章小结

为了能够使读者更加清楚地理解问题，同时为了更自然地在下文中使用一些相关术语，本章主要介绍了数据对象、时间粒度、持续时间等概念，并最后对最优时间窗口覆盖问题做了正式定义。

# 滑动时间窗口算法

针对上述的问题，在这一章中我们先提出一种相对低效的解决方法，我们称之为滑动时间窗口算法。该算法虽然能够解决上述的问题，但是在数据规模较大、用户时间跨度较大的情况下的性能很差。接下来，我们将详细介绍滑动时间窗口算法的算法细节。

## 算法描述

本算法的输入有三部分：1）所有用户以及对应的区间信息。2）时间粒度。3）根据查询要求给定的时间窗口大小。最终，通过本算法，我们能够得到最优时间区间和该区间覆盖的用户的集合。

滑动时间窗口算法是基于时间线扫描的一种算法。算法首先根据所有用户的区间信息构造一条一维的时间线。然后构造时间窗口大小的窗口，从时间线的起点开始滑动，每次滑动到一个新的位置时，都判断每个用户是否被该窗口覆盖，计算该窗口覆盖的用户的集合。当沿时间线扫描结束后，我们会得到一系列时间区间和对应覆盖的用户的集合，我们再从中挑选出覆盖最多用户的区间即为最优时间区间。

## 算法详解

图3.2展示了滑动时间窗口算法的伪代码。首先，通过扫描所有用户的有效区间，我们可以得到最小时间和最大时间，这一步我们可以在读入数据时完成。接下来，我们可以根据和画出如图3.1所示的一条一维时间轴，时间轴的起点是 ，终点是 。然后我们构造一个时间窗口大小duration的窗口w，该窗口的区间表示为[start, end]，并且end – start = duration。窗口w有一个对应的集合set，用来记录该窗口覆盖的用户的ID。接下来将窗口w沿时间线扫描。首先窗口w置于时间轴的起点，即start = 。接下来，我们扫描所有用户，判断每个用户是否窗口w覆盖。对于被覆盖的用户，我们将用户ID存入到对应的集合set中。当所有用户判断完毕后，我们将窗口w沿时间轴后移duration的距离，重新计算当前位置下的窗口w可以覆盖的用户。如此反复操作，直到窗口w沿时间轴扫描到终点，即end = 。在整个扫描过程中我们用max记录覆盖最多用户的数量。

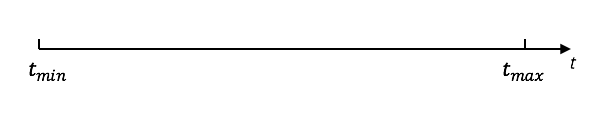


图 3. 1 滑动时间窗口算法时间线

|  |  |
| --- | --- |
| **Input:** | set S of records，time granularity ，duration |
| **Output**: | The best interval with its set of userIds |
|  | read all of records , get and |
|  | max = 0 |
|  | **for** (start = ; start <= - duration; start = start + granularity) |
|  | count =0; set.clear(); |
|  | **for** each user of |
|  | **for** each interval of the user |
|  | if (t\_start<= start && t\_end >= end) |
|  | count ++ |
|  | set.add(userId) |
|  | if (count > max ) |
|  | max = count |
|  | intervalMap←store the interval and its set |
|  | **for** each interval in intervalMap |
|  | if ( set’s size == max) |
|  | output this interval and its set |

图 3. 2 滑动时间窗口算法伪代码

图3.3展示了扫描时间线的过程。①是扫描的起始状态，红色部分表示窗口w，此时区间的开始时间为start = ，结束时间为start+duration。②表示经过一次循环后的情况。红色虚线部分是窗口w之前的位置，红线实现部分表示窗口新的位置。此时的start在之前的基础上加了granularity值。我们可以认为区间沿时间轴平移了granularity的距离，成为了一个新的区间。如此反复循环后，最终达到状态③。③表示遍历时间线的最后状态，此时窗口表示的区间的结束时间为start + duration = 。

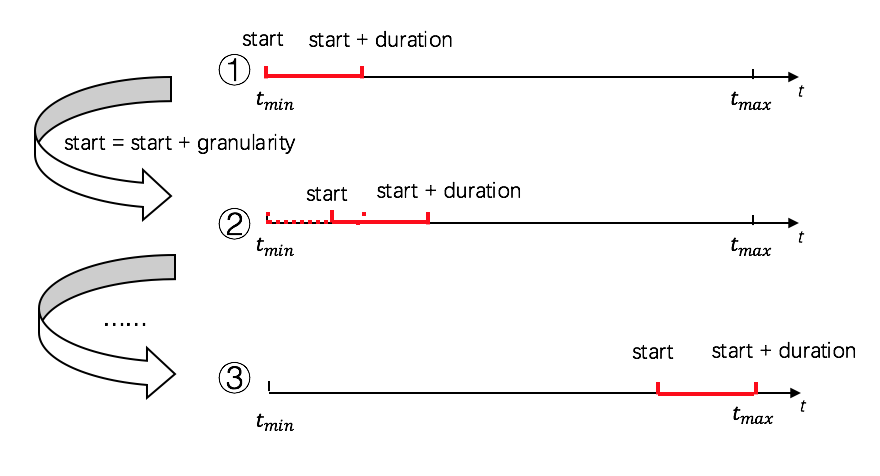


图 3. 3 滑动时间窗口算法时间线

## 算法演示

我们用表3-1的例子来演示滑动时间窗口算法的过程。表3-1中的user表示用户，valid interval表示用户的空闲时间，每个用户可以有多个空闲时间段。例如表中Allen有三个空闲时间段，分别为[0,5]、[7,10]和[11,14]。为了演示的方便，空闲时间段都用了整型的相对时间，而不是真实时间数据，并且时间粒度为1。表3-1的内容在图3.4中用图形的方式展示出来。图3.4中每条线段都代表了对应纵坐标上的用户的一个空闲时间段。例如最下面一行中一条0到5的线段表示Allen的一个空闲时间段为[0, 5]。

算法首先读入数据后，可以得到=0，＝15，我们以此构造一条一维时间轴。假设我们要求时间窗口duration为3，所以在时间轴的起点处的窗口w表示的区间为[0,3]。我们首先将max初始化为0，然后我们计算得到区间[0,3]覆盖的用户有Allen和Frank，覆盖的人数为2，大于max，所以将max变为2。之后将窗口w后移1个距离（因为时间粒度为1），新的窗口表示的区间为[1,4]，求得该窗口区间覆盖的用户有Allen, David和Frank，覆盖的人数为3，大于max，所以将max变为3。之后每次都将窗口后移1个距离，计算新的窗口区间覆盖的用户。如果覆盖的人数大于max，将覆盖的人数赋给max。这样子一直到扫描结束，将会得到max=4,以及如表3.2的结果，其中interval表示扫描时间线时每个窗口代表的区间，set是对应区间覆盖的人员的集合。最后输出表2结果中集合大小为4的区间，即[2,5]和[4,7]就是最优时间区间。

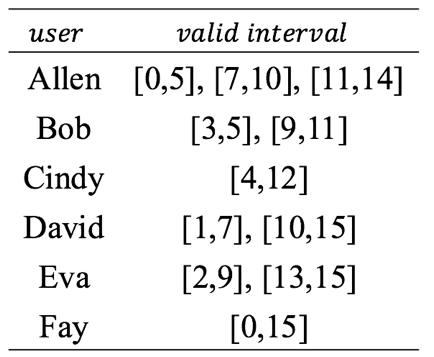
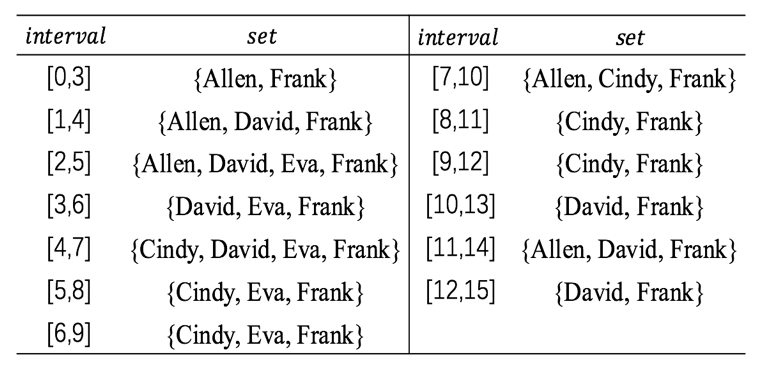
 

表 3 - 1人员空闲时间表 表 3 - 2 扫描时间线的结果

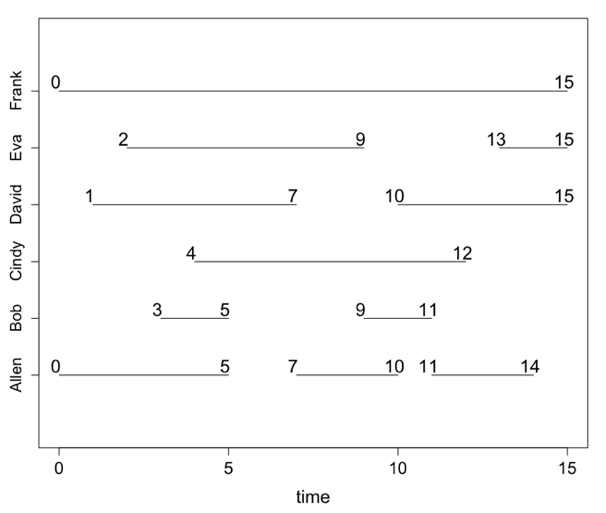


图 3. 4 表3-1示意图

## 算法复杂度

根据算法的伪代码我们可以看出，滑动时间窗口算法主要由两层循环组成。第一层循环是扫描时间线，第二层循环是每次扫描时判断所有记录与当前区间的覆盖关系。所以滑动时间窗口的时间复杂度为O(mn)，其中m是沿时间轴扫描的次数，它的值可以根据时间范围range除以时间粒度granularity得到，而n是记录的数量。

## 代码实现

滑动时间窗口算法主要的核心代码就是沿着时间线扫描的过程。我们从时间线的起点开始构造一个大小为duration的时间区间，判断所有用户与该时间窗口的覆盖关系。判断完后再增加一个步长，修改区间位置，继续判断。直到扫描到时间线的终点，得到所有可能的区间和对应的覆盖用户集合。

/\*\*

\* 从2013-07-01 00:00:00开始，沿着时间线一次扫描，获得所有时间区间的结果，区间的终止时间－开始时间＝duration

\* **@param** db 原始数据库

\* **@param** duration

\* **@return** 最优区间覆盖的记录条数

\*/

**public** **int** scanTimeline(String timeline, Database db, **int** duration, **int** step, String granularity){

//提取timeline中的起始时间和终止时间

String[] t = timeline.split(",");

**int** t1 = Integer.*parseInt*(t[0]); //起始时间

**int** t2 = Integer.*parseInt*(t[1]); //终止时间

**int** time = t1;//表示扫描时当前的时间结点，从t1开始

**int** max = 0; //记录区间覆盖的最多用户数量

String start,end;

**while**(time <= (t2- duration)){

Iterator<Entity> it = db.data.iterator();

Entity e;

ArrayList<String> userSet = **new** ArrayList<String>();

**while**(it.hasNext()){

e = it.next();

**if**(e.getStartTime() <= time && e.getEndTime() >= (time+duration)){

userSet.add(e.getUser());

}

}

**if**(granularity.equals("second")){

start = transformFromSecond(time);

end = transformFromSecond(time+duration);

}

**else** {

start = transformFromMinute(time);

end = transformFromMinute(time+duration);

}

String Interval = "[" + start + "," + end + "]";

**if**(userSet.size() >= max)

{

max = userSet.size();

ir.put(Interval, userSet);

}

time += step ;

}

**return** max;

}

## 本章小结

本章首先用简单的语言介绍了滑动时间窗口算法的输入输出和整体的算法思路，再结合伪代码详细解释了算法的细节。然后用一个例子演示了算法的运行过程，让大家对滑动时间窗口算法有更加深刻的理解。最后给出了滑动时间窗口算法的复杂度和核心代码实现。

# 基于SB\*-Tree的查询算法

本章我们将提出一种高效的解决方法——基于SB\*-Tree的查询算法。SB\*-Tree是在SB-Tree的基础上，对SB-Tree的结点结构进行改进而提出的，而且Sb\*-Tree继承了SB-Tree的很多操作，所以有必要在介绍SB\*-Tree之前，详细介绍一下SB-Tree。如果能够掌握了SB-Tree的思想，那么理解SB\*-Tree将非常轻松。

## SB-Tree

SB-Tree是用于计算时间数据库中的时间聚合操作的一种索引结构。在这一小节中，我们详细介绍SB-Tree的结点结构和相关操作，确保大家对SB-Tree有清楚的了解。

### SB-Tree结点结构

SB-Tree共有三种类型的结点，分别为根结点、中间结点和叶子结点。如图4.1和图4.2分别展示了SB-Tree的中间结点和叶子结点结构示意图。另外每棵SB-Tree都有最大分支参数b和最大叶子容量l来决定SB-Tree的整体结构布局。接下来将详细介绍SB-Tree的结点特征：

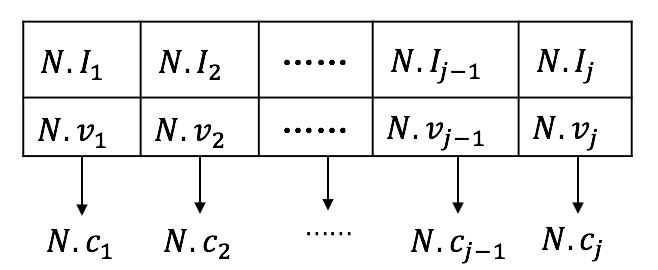
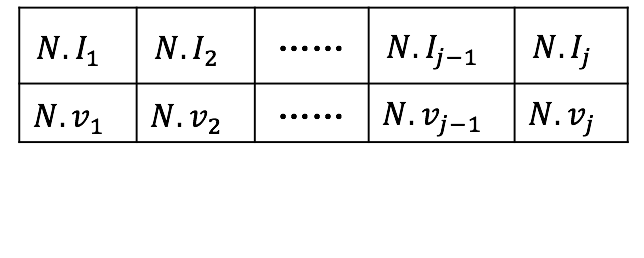
 

图 4. 1 SB-Tree中间结点 图 4. 2 SB-Tree叶子结点

* 每个中间结点最多能够表示个连续的时间区间，并且要保证该结点至少是半满的，即至少表示个时间区间。如图4.1中结点表示了个时间区间，分别为，，，……，。同时每个时间区间都有对应的参与人数和指向下一个子结点的指针。
* 叶子结点最多能够表示个连续的时间区间，并且必须至少表示个时间区间。其结构与中间结点类似，只是没有指向下一个子结点的指针。如Fig.2所示。
* 根结点的结构与中间结点相同，同时最多能够表示个连续的时间区间，但是最少只要表示2个时间区间即可。
* 对于每个非叶子结点 ，所有在以为根的结点中的区间都被包含，同时以为根的结点中的所有区间的开始时间都要严格大于的结束时间。

### 插入

我们调用过程)来完成插入操作，)表示向以N为根的子树中插入。其中v是聚合值，I是对应的区间。)的具体操作为：

* 对于所有满足的*i*
  + 如果,那么其区间对应的聚合值加v
  + 如果：
* 如果N不是叶子结点，调用过程))
* 如果N是叶子结点，则对该结点进行修改来反映插入的影响。

### 删除

删除可以看作是插入的反操作，例如要删除，可以看作是插入。所以关于删除操作)的具体操作如下：

* 对于所有满足的*i*
  + 如果,那么其区间对应的聚合值减去v
  + 如果：
* 如果N不是叶子结点，调用过程))
* 如果N是叶子结点，则对该结点进行修改来反映插入的影响。

### 结点分裂

之前提到每个结点都有最大的容量，所以当该结点表示的时间区间过多时，就会导致溢出，此时需要进行结点分裂操作。因为插入的时间区间都是添加到叶子结点的，所以首先是叶子结点会发生溢出。当叶子结点执行完分裂操作后，可能会导致其父结点溢出，再进一步对父结点的进行分裂操作。具体操作如下：

|  |
| --- |
| 假设当节点发生溢出时包含了个时间区间，则当为叶子节点时，;当为非叶子节点时，。我们将节点的分裂过程记为   * 首先我们按如下方法将节点分裂成： * 包含前一半的区间，即包含,……,以及对应的聚合值,……,。若,……, * 包含剩余的区间，即包含以及对应的聚合值。如果不是叶子节点，,……, * 如果是根节点，那么创建一个新的根节点，只有两个区间，其中，，,分别指向。其中表示区间的结束时间。 * 如果不是根节点，假设有父节点，并且 * 将第j个区间分成和[]，两个区间的聚合值与之前相同，再令。 * 如果溢出，则调用 |

### 区间合并

当发生插入或者删除操作后，可能会出现两个相邻区间的聚合值相同的情况，这时就需要将两个相邻区间进行合并。区间合并分为两种情况：

1.两个相邻的区间在同一个叶子节点中

2.两个相邻的区间在两个相邻的叶子节点中

对于第一种情况，具体操作如下：

|  |
| --- |
| * 两个相邻的区间在同一个叶子节点中，假设它们分别为，那么有 * 删除, * 从第个区间到最后的区间往前移一位 * 如果节点经过合并后包含的区间少于，调用 |

第二种情况相对比较复杂，对于位于两个相邻叶子结点中的两个相邻区间，我们不能根据两个区间的聚合值是否相对就判断两个区间可以合并。我们还需要考虑到两个叶子结点的共同父结点到各自结点路径上的聚合值。所以，第二种情况的具体操作如下：

|  |
| --- |
| * 两个相邻的区间分别位于两个叶子结点和中，且为左结点，为右结点。假设包含个时间区间，包含个时间区间。那么两个相邻的区间分别为和。假设结点是叶子结点和的最近的共同父结点，如果结点到路径上所有聚合值的和等于结点到路径上所有聚合值的和，则进行两个区间合并。 * 如果包含的时间区间个数大于，则把合并到中，具体操作为： * 在中，删除和 * 从到的路径上，每个对应的区间的结束时间改为 * 从到的路径上，每个对应的区间和的开始时间改为 * 否则，把合并到中去，具体操作为： * 在中，删除和。 * 从到的路径上，每个对应的区间开始时间改为 * 从到的路径上，每个对应的区间和的结束时间改为 |

### 结点合并

区间合并后，可能会使叶子节点达不到半满状态，此时需要进行节点合并来防止这种情况发生。节点合并的策略是：如果非根节点没有达到半满状态，首先从包含时间数量超过容量一般的相邻节点中移动一个时间区间到该节点。如果不存在这样的有“多余”时间区间的节点，那么就将该节点与其相邻的节点进行合并。具体操作如下：

|  |
| --- |
| 假设节点没有达到半满状态，节点合并的过程定义如下：   * 如果是根节点： * 如果有且仅有1个子节点，那么将作为新的根，删除原来的根节点 * 否则，不做任何改变。 * 如果不是根节点，假设最多能够包含个时间区间（如果是叶子节点，;如果是非叶子节点,）。现在包含了个时间区间 * 如果的右边兄弟节点包含了至少个时间区间，那么就把的第一个区间添加到中去。 * 如果的左边兄弟节点包含了至少个时间区间，那么就把的最后一个区间添加到开头中去。 * 否则，将要进行两个节点的合并。假设这两个节点分别为，且为左节点，为右节点。假设包含f个时间区间，包含个时间区间。和中其中一个为 * 假设是和共同的父节点，并且 * 将合并为一个新的区间，包含了个时间区间，并删除。 * 在中，将，并指向。 * 如果节点包含的区间少于，调用 |

## SB\*-Tree

我们在SB-Tree的基础上，对结点结构进行改进，增加了能够存储在每个时间区间有效的用户集合。修改后的结点如图4.3和图4.4所示。

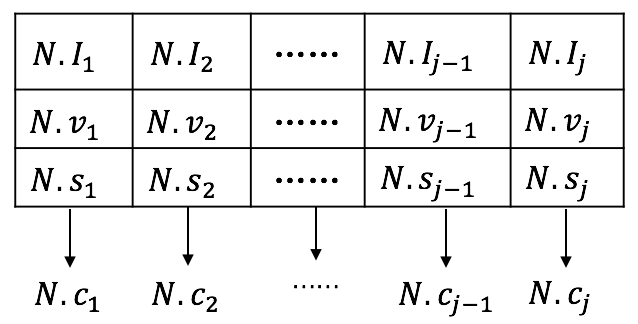
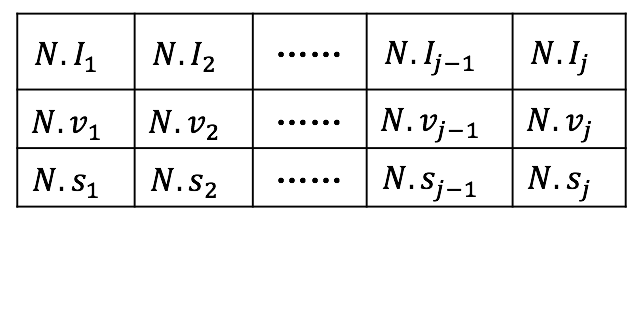
 

图 4. 3 SB\*-Tree中间结点 图 4. 4 SB\*-Tree叶子结点

SB\*-Tree继承了上述SB-Tree的所有操作，并且在原来的基础增加了上对用户集合的操作。接下来，我们将具体以例子的形式介绍SB\*-Tree的各种操作。在插入操作中，当增加聚合值时，还要将用户ID加入到对应的集合中。在删除操作中，当减去聚合值时，也要将用户ID从对应的集合中删去。结点分裂和结点合并都要考虑集合。区间合并时的判断条件也变成了判断两者的用户集合是否完全相等，而不再是判断两者的聚合值。

### 插入

SB\*-Tree的插入策略与SB-Tree的基本一致。不过当SB\*-Tree在增加区间的聚合值的同时，还要将对应的用户ID添加到相应区间的用户集合中去。

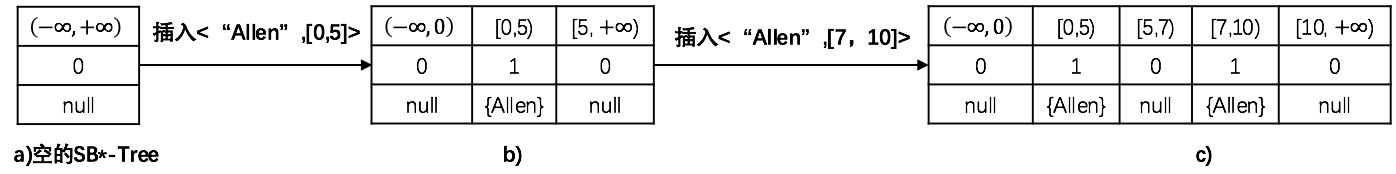


图 4. 5 SB\*-Tree插入示意图

首先一棵初始空的SB\*-Tree如图4.5a)所示。首先我们插入<Allen ,[0,5]>。根结点（也是叶子结点）中的第一个区间与[0,5]相交，所以我们将原来区间分裂成三个区间，分别为、和 ，并且它们分别对应的用户集合为null、{Allen}、null，如图4.5所示。接下来我们插入<Allen,[7,10]>。根结点中的第三个区间与待插入区间[7,10]相交，所以我们将原来区间[5,+∞]分裂成[5,7]，[7,10]，[10,+∞]，它们对应的用户集合为：null、{Allen}、null。最终结果如图4.5 c所示。

### 删除

如图4.6展示了SB\*-Tree删除示意图。图4.6a)与上一节中图4.5c)是一样的。我们按照与插入相反的顺序来进行删除。首先删除<Allen，[7,10]>，我们就把根结点第四个区间的聚合值减去1，用户集合中删除Allen，结果如图4.6所示。接着我们删除<Allen，[0,5]>，把根结点第二个区间的聚合值减去1，用户集合中删除Allen，结果如图4.6所示。按照理论上来将，先插入在删除相同的内容后，结点的结构应该保持不变。但是我们看到图4.6和图4.5的结构有所不同，图4.6和图4.5也不同，这是因为在删除后，没有对具有相同用户结合的相邻的区间进行合并操作。

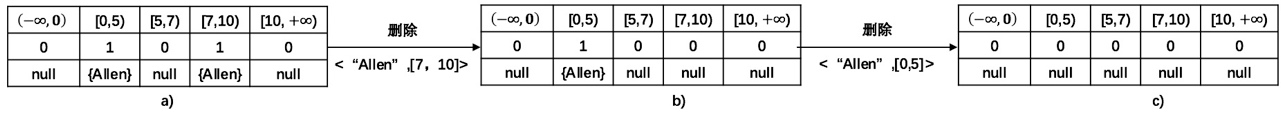


图 4. 6 SB\*-Tree删除示意图

### 结点分裂

SB\*-Tree的结点分裂操作与SB-Tree基本相同，所以直接以例子阐述具体过程。假设b=l=5，当我们在4.5c)的基础上插入<“Allen”,[11，14]>后,结点会变成如图4.7的结构。此时结点超过最大容量5，所以我们需要进行分裂的操作。我们将前四个区间和相应的信息赋给一个新创建的结点，将剩余三个区间赋给另一个新创建的结点。我们还要再创建一个新的根结点，根结点包含区间(-∞,10)和，并指向那两个新创的结点。分裂后的SB\*-Tree如图4.7所示。

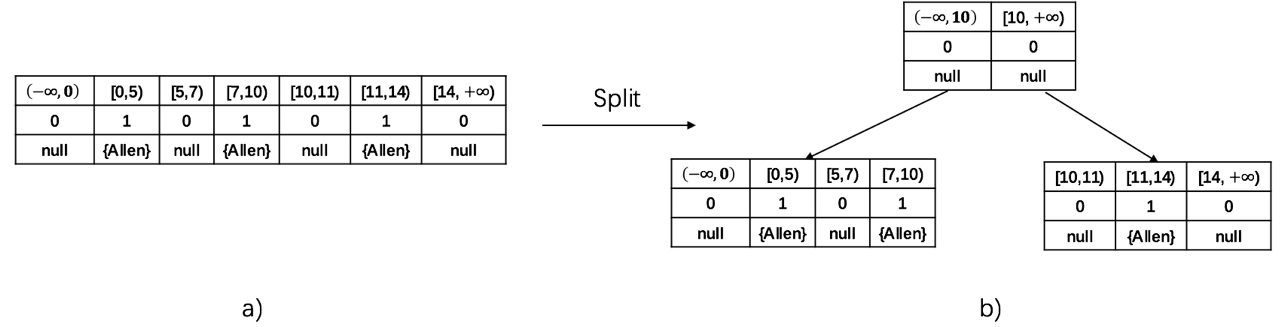


图 4. 7 SB\*-Tree结点分裂示意图

### 区间合并

小节4.2.2中已经具体地解释区间合并的作用，所以在这一小节中我们将介绍SB\*-Tree的区间合并的过程。在SB-Tree中，我们是通过判断相邻的区间的聚合值是否相同来决定是否要进行区间合并。但是在SB\*-Tree中，我们要判断的是相邻区间的用户集合是否想等而不是聚合值。其他SB\*-Tree的区间合并操作与SB-Tree的基本相同。

以图以图4.6b)为例，区间[5,7)和其相邻区间[7,10)的用户集合都为空，所以可以合并成一个区间[5,10)，对应的聚合值为0，用户集合为空，结果如图4.8b)所示。接着我们看到区间[5,10)和[10,+∞)的用户集合也都为空，所以可以将两个区间合并成一个区间[5,+∞)，对应的聚合值为0，用户集合为空，结果如图4.8 c)所示。我们可以看到最终图4.8 c)与图4.5 b)的结构相同，这就体现了区间合并的作用，可以压缩结点结构，使树变得更加紧凑。

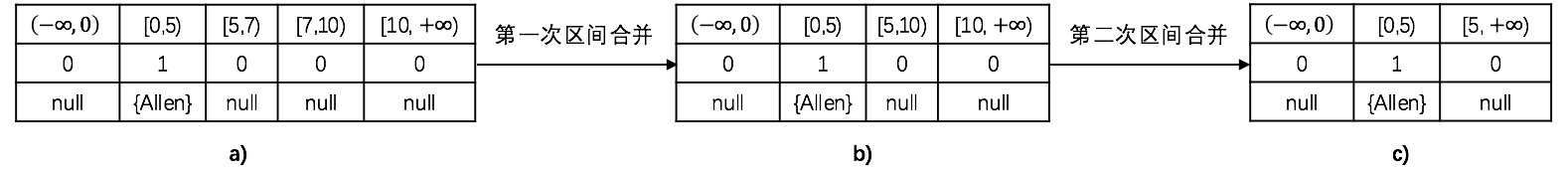


图 4. 8 SB\*-Tree区间合并示意图

### 结点合并

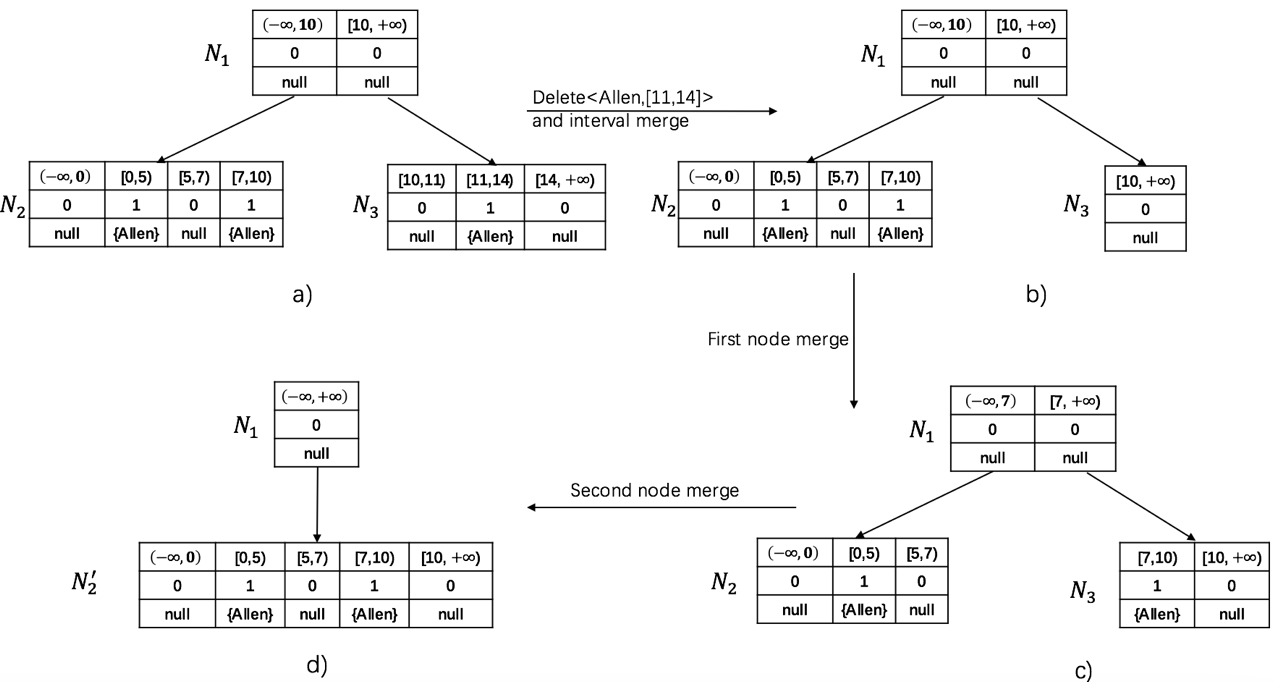


图 4. 9 SB\*-Tree结点合并示意图

SB\*-Tree的结点合并策略与SB-Tree的也基本相同。接下来我们以一个例子来具体阐述SB\*-Tree的结点合并过程。我们以图4.7为例，假设我们要在此基础上删除<Allen,[11,14]>。我们规定b=l=5，那么除了根结点外的每个结点都至少存储三个时间区间。删除且区间合并后的SB\*-Tree结构如图4.9所示。此时我们看到结点只存储了一个区间，所以需要进行结点合并。我们看到结点的左兄弟结点存储了四个区间，可以“分享”一个区间给结点。所以我们将结点的最后一个区间添加到结点，并把父结点中的区间改成，改成，它们的对应聚合值和用户区间不变，结果如图4.9所示。经过第一次结点合并后，结点仍然没有满足半满的要求，需要继续执行结点合并的过程。但是此时没有兄弟结点能够分享区间，所以需要把结点和结点合并成一个新的结点，结点包括了结点和结点的所有区间。同时，父结点的两个区间也要合并成区间，结果如图4.9所示。此时根结点只包含一个区间，它不满足半满的情况，根据4.1.6中SB-Tree的结点合并策略，我们要删除根结点，并将结点作为新的根结点，最终的结果与图4.6一致。

## 查询算法

根据原始数据构造好SB\*-Tree后，我们需要相应的查询算法来得到最优时间区间。我们设计的查询算法大致分为三步：1.获取中间结果。2.区间调整。3.获得最优区间。接下来我们将详细介绍每一步具体的工作。

### 获取中间结果

这一步中，我们按广度优先的方法遍历SB\*-Tree，可以得到所有时间区间以及对应的用户集合。遍历的方法如下：将树的根节点加入到一个队列中，每次先判断队列是否为空，如果队列不为空则取队列中的第一个元素进行判断。取出的结点若是中间结点，则将该结点的每个区间信息加到它的子结点中，并把子结点加入到队列中。如果取出的结点是叶子结点，直接输出时间区间和对应的用户集合。图4.10展示了遍历SB\*-Tree的伪代码。

|  |  |
| --- | --- |
| **Input:** | 一棵构建好的SB\*-Tree |
| **Output** | 所有时间区间和对应的用户集合 |
| 1. | add the tree’s root to a queue |
| 2. | **while** queue is not empty |
| 3. | N ← Get the queue’s first element |
| 4. | delete the queue’s first element |
| 5. | **if** N is not a leaf node |
| 6. | add N’s set to all children of N |
| 7. | Add all N’children to queue |
| 8. | **else** N is leaf node |
| 9. | output all intervals of the N with their sets |

图 4. 10 遍历SB\*-Tree伪代码

### 区间调整

根据第一步得到的时间区间大小可能不满足时间窗口duration，所以为了得到满足时间窗口duration大小的候选方案，我们需要对遍历SB\*-Tree树得到的中间结果中的每个区间进行判断和调整，将区间大小小于duration的区间与其后相邻的区间进行合并，直到新的区间能够满足duration。当合并后的新区间满足duration要求后，我们还需要继续向下合并，直到合并后的区间的用户集合发生改变。

图 4. 11 区间调整伪代码

|  |  |
| --- | --- |
| **Input:** | 遍历SB\*-Tree得到的中间结果 |
| **Output:** | 经过区间调整后的结果 |
| **1.** | max = 0 |
| **2.** | **for** each interval of the intermediate result |
| **3.** | ← |
| **4.** | **while** <duration |
| **5.** | ← the next interval of |
| **6.** | ← combine with |
| **7.** | ← intersection of and |
| **8.** | ← the next interval of |
| **9.** | **While** equals to intersection of and |
| **10.** | ← combine with |
| **11.** | ← intersection of and |
| **12.** | ← the next interval of |
| **13.** | output and |
| **14.** | **if** .size > max |
| **15.** | max = .size |

图4.11展示了对中间结果区间调整的伪代码。其中用来记录要与区间合并的区间。当需要合并时，首先是区间后第一个区间。合并后，若区间仍不满足时间窗口duration大小，那么需要继续合并。这时变成区间后第二个区间。这样子一直合并判断，直到合并后的区间满足duration大小。而对应的用户集合表示在区间内一直有效的用户，所以当两个区间进行合并时，它们的用户集合应该做交集。最终，合并后的区间的用户集合应该是所有与区间合并过的区间的用户集合和的交集。当合并后的区间满足duration大小要求后，我们还需要继续向下合并区间，判断是否新的区间用户集合与不合并的区间用户集合保持不变。若保持不变，我们则继续向下合并，直到合并后的区间的用户集合发生改变。我们用max来记录在区间调整过程中最大的区间用户集合的大小。

### 获得最优区间

经过第二步区间调整得到的每一个区间都是满足时间窗口duration的要求的，所以我们只要根据上一步中的max来遍历所有区间调整后的所有区间，凡是遇到用户集合的大小等于max的区间，都是我们要找的最优区间，我们将区间和对应的用户集合输出即可。

## 算法演示

我们仍然以表3-1为例来演示SB\*-Tree算法的整个过程，假设我们规定了SB\*-Tree的b=l=5，时间窗口duration=3。图4.12～图4.21展示了SB\*-Tree完整的构造过程。首先插入<“Allen”,[0,5]>和<“Allen”,[7，10]>,按照SB\*-Tree的插入算法，可以得到如图4.12所示的SB\*-Tree。当继续插入<“Allen”,[11,14]>后,根结点会溢出，此时会发生结点分裂操作，分裂后的结果如图4.13所示。再之后插入<“Bob”,[9,11]>时也发生了分裂，如图4.15所示。当插入<“Cindy”,[4,12]>时，根结点的区间[5,10]完全被待插入区间[4,12]完全覆盖，所以直接修改区间[5,10]对应的值和集合，不需要继续向子树遍历，如图4.16所示。SB\*-Tree剩余的构造构成如图4.17~4.21所示。表4-1展示了遍历SB\*-Tree后得到的每个时间区间和对应的用户集合。我们可以看到表4-1中的大部分区间都不满足时间窗口大小的要求，所以我们需要对表4-1中的区间进行区间调整，使其变得满足时间窗口大小的要求。我们首先从区间[0,1]开始，因为它的大小小于3，不满足要求，所以要与其接下来的区间进行合并。首先与[1,2]合并，合并后的区间[0,2]仍然不满足要求，继续与后面的区间[2,3]合并。合并后的区间[0,3]满足要求则停止合并，而其对应的用户集合则是区间[0,1]、[1,2]和[2,3]三个区间的用户集合的交集，即{Allen, Fay}。接下来对[1,2]进行同样的操作，直到表4-1中的每个区间都判断完成。最终区间调整结束后，会得到表4-2。表4-2中的每个区间都是满足时间窗口duration的要求，并且我们可以看到最优的区间应该是[2,5]和[4,7]，所得结果与滑动时间窗口算法得到的结果一致。

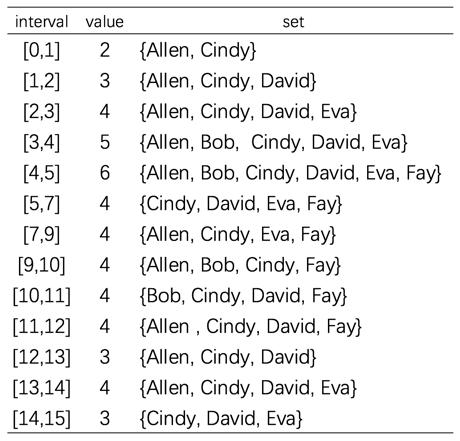
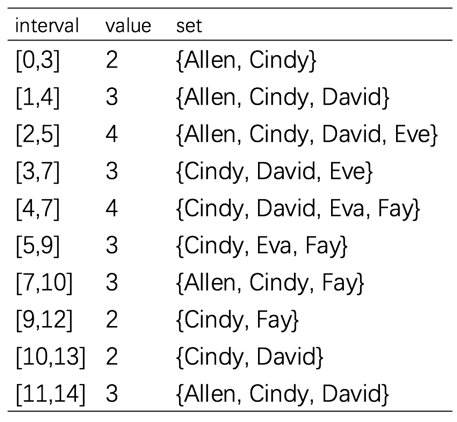
 

表 4 - 1 遍历SB\*-Tree的中间结果 表 4 - 2 区间调整后的结果

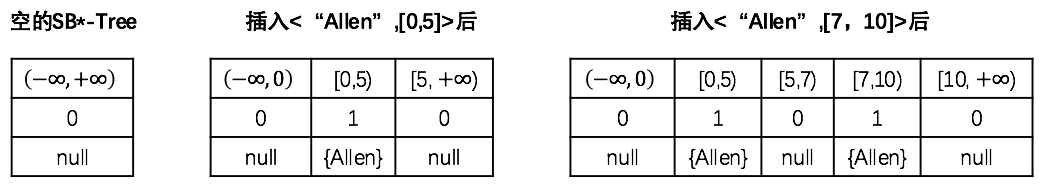


图 4. 12 表3-1内容构造SB\*-Tree步骤1：插入<Allen,[0,5]>和<Allen,[7,10]>

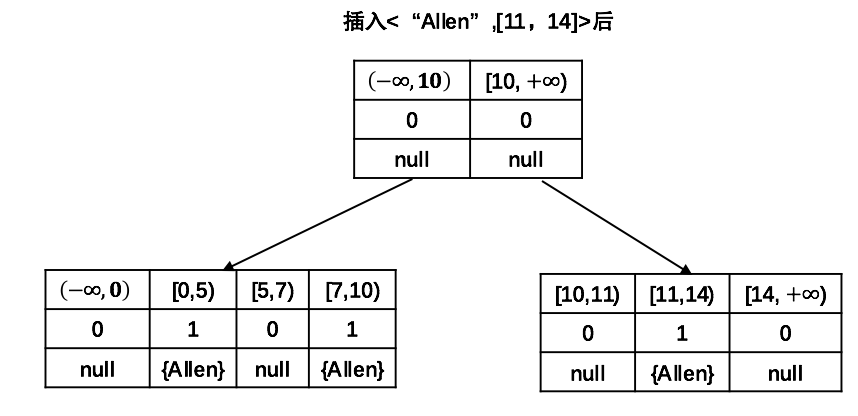


图 4. 13表3-1内容构造SB\*-Tree步骤2：插入<Allen,[11,14>

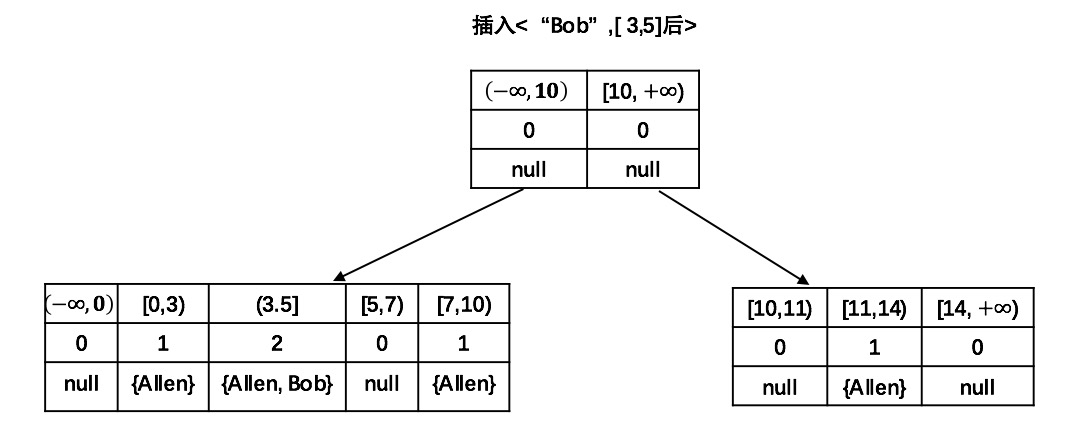


图 4. 14表3-1内容构造SB\*-Tree步骤3：插入<Bob,[3,5]>

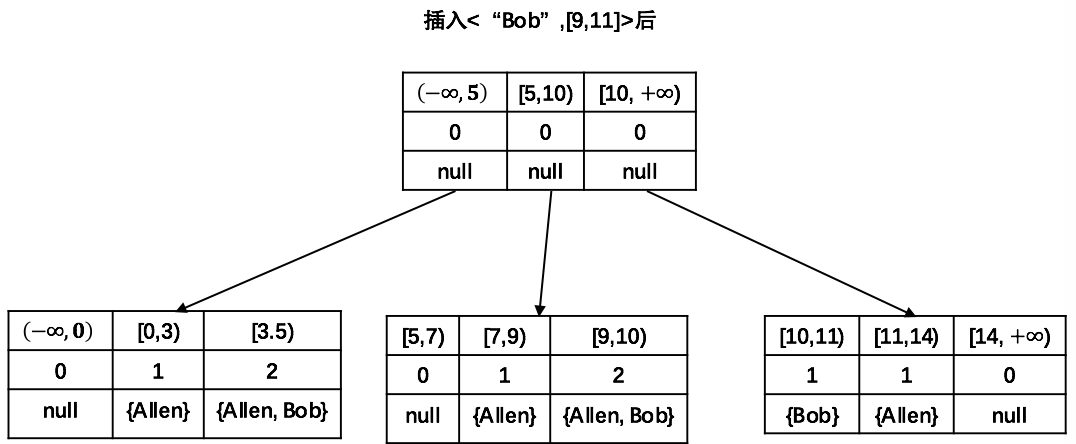


图 4. 15表3-1内容构造SB\*-Tree步骤4：插入<Bob,[9,11]>

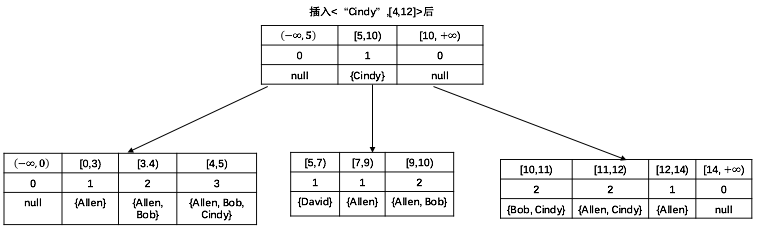


图 4. 16表3-1内容构造SB\*-Tree步骤5：插入<Cindy,4,12>

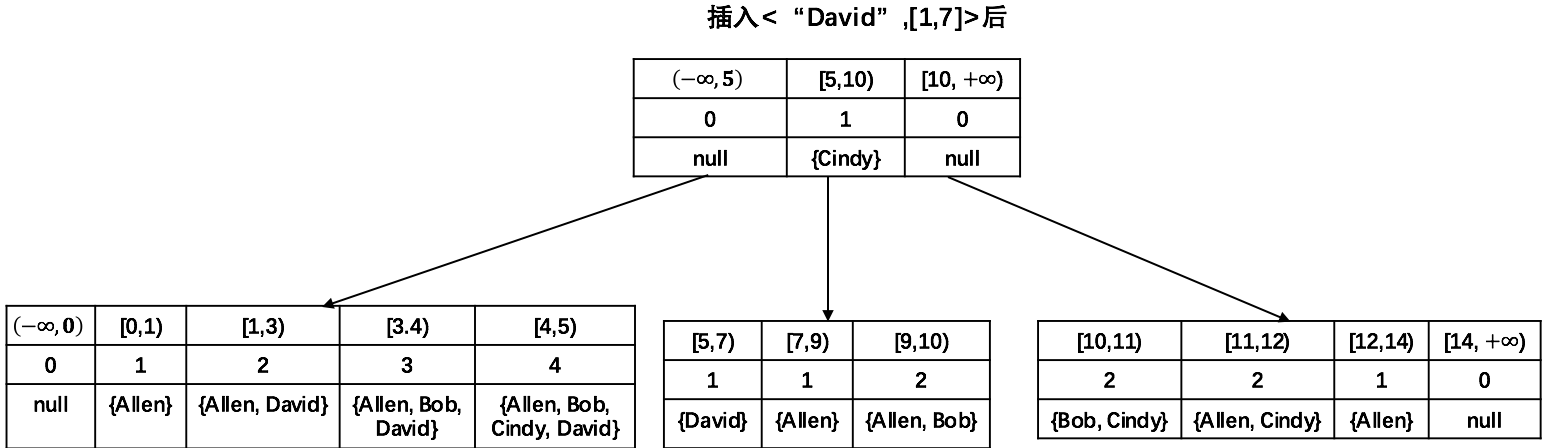


图 4. 17表3-1内容构造SB\*-Tree步骤6：插入<David,[1,7]>

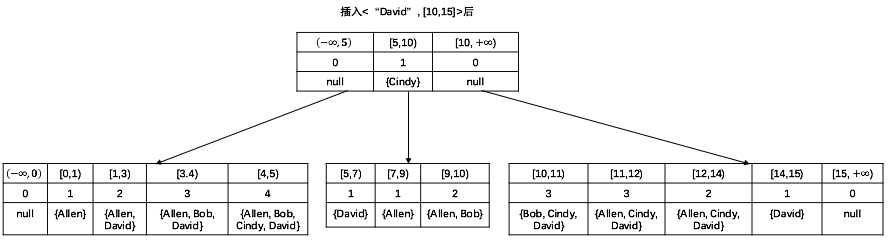


图 4. 18表3-1内容构造SB\*-Tree步骤7：插入<David,[10,15]>

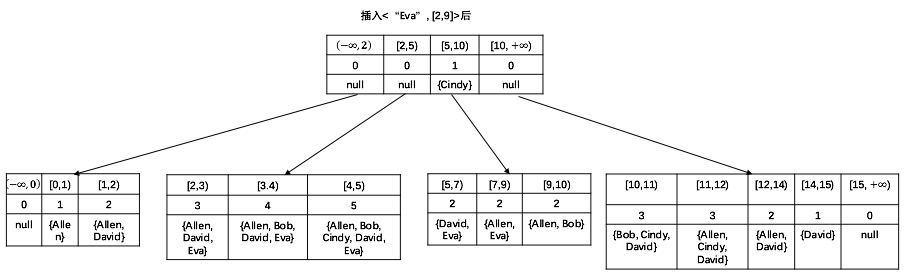


图 4. 19表3-1内容构造SB\*-Tree步骤8：插入<Eva,[2,9]>

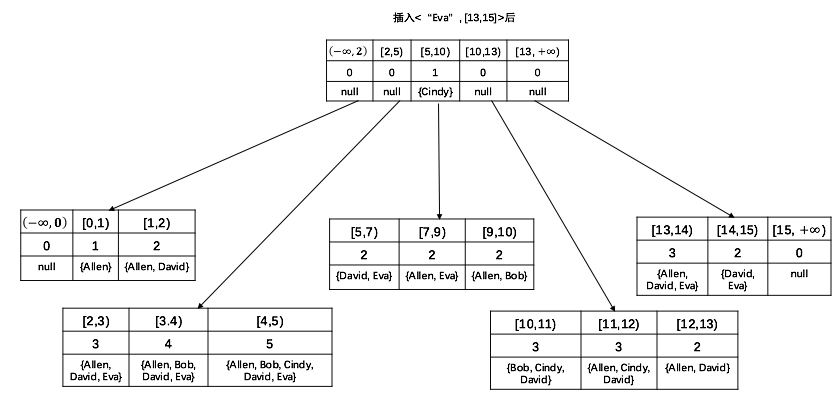


图 4. 20表3-1内容构造SB\*-Tree步骤9：插入<Eva,[13,15>

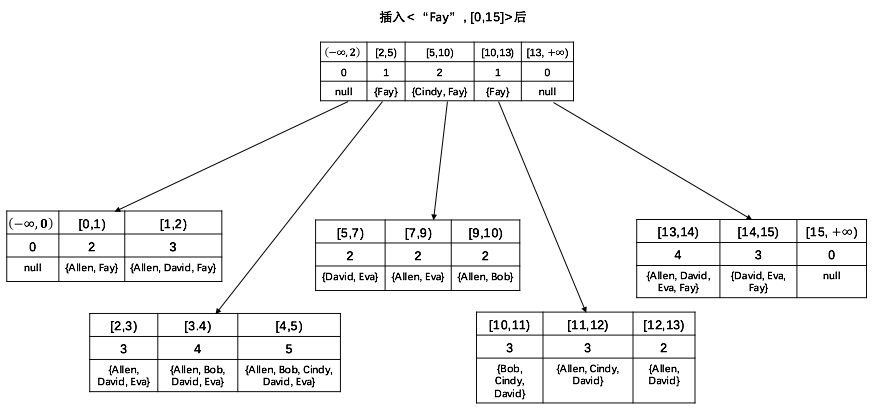


图 4. 21表3-1内容构造SB\*-Tree步骤10：插入<Fay,[0,15]>

## 代码实现

本节首先介绍主要的类和类之间的关系，再介绍核心方法的代码。对于其中没有提到的方法，请详细参考我的源程序。

### 类说明



图 4. 22 SB\*-Tree类关系说明图

如4.22所示是SB\*-Tree主要类所包含的变量以及相应的关系。其中AbstractNode类表示一个树的结点，它的变量如图中所示。注意AbstactNode类中有一类型为Entry的数组变量entries，它表示结点每个区间的入口。每个Entry都存储着一个类型为Interval类的区间变量I和该区间指向的子结点。而Interval类则存储着一个区间的所有信息，如开始时间、结束时间、覆盖该区间的用户数量以及覆盖该区间的用户集合。

### 插入操作代码实现

本小节主要展示SB\*-Tree的插入操作的代码。插入的具体策略与小节4.1.2类似，在这里就不再赘述。

/\*\*

\* Inserts a new interval into the tree. If enough space is available

\* the insertion is straightforward. If not, a split must occur.

\*

\* **@param** I The interval to insert.

\* **@return** True if a split occurred, false otherwise.

\*/

**protected** **boolean** insert(Interval I) {

Entry leftSplit = **null**, rightSplit = **null**, innerSplit = **null**;

/\*\* mark the fact whether only the node's interval value is modified. \*/

**boolean** dirty = **false**;

// special case for empty tree!

**if** (isRoot() && usedSpace == 0) {

insertEntry(**new** Entry(**new** Interval(Integer.***MIN\_VALUE***, Integer.***MAX\_VALUE***, tree.getDefaultData(),**new** ArrayList<String>())));

}

**for** (**int** i = 0; i < usedSpace; i++) {

**if** (entries[i].I.intersects(I)) {

**float** d = entries[i].I.v + I.v;

List set = entries[i].I.getCombinedSet(I);

**if** (entries[i].I.set.equals(set)) {

// do nothing.

} **else** **if** (I.contains(entries[i].I)) {

entries[i].I.v = d;

entries[i].I.set.clear();

entries[i].I.set.addAll(set);

dirty = **true**;

}

**else** {

// check if there is a left, right or inner intersection.

**if** (entries[i].I.s < I.s) {

**if** (entries[i].I.e > I.e) {

innerSplit = entries[i];

}

**else** {

leftSplit = entries[i];

}

}

**else** {

rightSplit = entries[i];

}

}

}

}

**if** (isLeaf()) {

**if** (innerSplit != **null**) {

Interval i1 = **new** Interval(innerSplit.I.s, I.s, innerSplit.I.v, (ArrayList) innerSplit.I.set);

Interval i2 = **new** Interval(I.e, innerSplit.I.e, innerSplit.I.v, (ArrayList) innerSplit.I.set);

Interval i3 = **new** Interval(I.s, I.e, innerSplit.I.v + I.v, (ArrayList) innerSplit.I.getCombinedSet(I));

removeEntry(innerSplit);

insertEntry(**new** Entry(i1));

insertEntry(**new** Entry(i2));

insertEntry(**new** Entry(i3));

**if** (usedSpace > tree.getNodeCapacity()) {

split();

**return** **true**;

}

**else** {

//Write the node to the tempTree instead of the tree ,because we may merge the leaves

tree.tempLeaves.put(**this**, 0);

}

}

**else** **if** (leftSplit != **null** || rightSplit != **null**) {

Interval i1 = **null**, i2 = **null**;

**if** (leftSplit != **null**) {

i1 = **new** Interval(leftSplit.I.s, I.s, leftSplit.I.v, (ArrayList) leftSplit.I.set);

i2 = **new** Interval(I.s, leftSplit.I.e, leftSplit.I.v + I.v,(ArrayList) leftSplit.I.getCombinedSet(I));

removeEntry(leftSplit);

insertEntry(**new** Entry(i1));

insertEntry(**new** Entry(i2));

}

**if** (rightSplit != **null**) {

i1 = **new** Interval(rightSplit.I.s, I.e, rightSplit.I.v + I.v,(ArrayList) rightSplit.I.getCombinedSet(I));

i2 = **new** Interval(I.e, rightSplit.I.e, rightSplit.I.v, (ArrayList) rightSplit.I.set);

removeEntry(rightSplit);

insertEntry(**new** Entry(i1));

insertEntry(**new** Entry(i2));

}

**if** (usedSpace > tree.getNodeCapacity()) {

split();

**return** **true**;

}

**else** {

tree.tempLeaves.put(**this**, 0);

}

}

**else** {

**if** (dirty) {

tree.tempLeaves.put(**this**, 0);

}

}

}

// the node is not a leaf

**else** {

**if** (dirty) {//do nothing

}

**if** (innerSplit != **null**) {

(innerSplit.child).insert((Interval) I.clone());

} **else** {

**if** (leftSplit != **null**) {

(leftSplit.child).insert(I);

}

**if** (rightSplit != **null**) {

(rightSplit.child).insert (I);

}

}

}

**return** **false**;

}

### 分裂操作代码实现

分裂操作的代码如下所示。首先根据要进行分裂的结点的类型（叶子结点或中间结点）创建两个相同类型的结点，再把原来结点中的区间分配给新创建的两个结点。如果我们要分裂的是根结点，那么我们需要创建一个新的根结点来指向新创建的两个结点；否则我们要调整父结点，把父结点中指向原来结点的区间分裂，此时如果父结点溢出，再进行分裂操作。

/\*\*

\* Split a node when it is overflow

\*/

**protected** **void** split() {

AbstractNode n1, n2;

**if** (isLeaf()) {

n1 = **new** Leaf(tree, parent);

n2 = **new** Leaf(tree, parent);

} **else** {

n1 = **new** Index(tree, parent, level);

n2 = **new** Index(tree, parent, level);

}

**int** div = (**int**) Math.*ceil*((**double**) usedSpace / 2);

**final** **int** pivot = entries[div].I.s;

**for** (**int** i = 0; i < div; i++) {

n1.insertEntry(entries[i]);

**if**(entries[i].child != **null**)

entries[i].child.parent = n1;

}

**for** (**int** i = div; i < usedSpace; i++) {

n2.insertEntry(entries[i]);

**if**(entries[i].child != **null**)

entries[i].child.parent = n2;

}

//if root is split

**if** (isRoot()) {

AbstractNode root = **new** Index(tree, **null**, level + 1);

n1.parent = root;

n2.parent = root;

root.insertEntry(**new** Entry(**new** Interval(Integer.***MIN\_VALUE***, pivot, tree.getDefaultData(), **new** ArrayList<String>()), n1));

root.insertEntry(**new** Entry(**new** Interval(pivot, Integer.***MAX\_VALUE***, tree.getDefaultData(), **new** ArrayList<String>()), n2));

tree.root = root;

}

**else** {

Index p = (Index) getParent();

p.adjustNode(**this**, n1, n2, pivot);

}

**if** (n1.isLeaf()){

tree.tempLeaves.put(n1, 0);

tree.tempLeaves.put(n2, 0);

}

}

### 区间合并代码实现

区间合并分为两种情况：1、两个相邻的区间属于同一个结点。2、两个相邻的区间属于相邻的结点，即一个是左边结点最后一个区间，另一个是右边结点第一个区间。

对于第一种情况，操作比较简单，只需要把两个相邻的区间合成一个区间即可。代码如下：

/\*\*

\* merge for the adjacent interval that belong to the same leaf

\* note that only the leveas' intervals need to be merged.

\* **@param** pos the first index of the entry that needs to be merged

\*/

**protected** **void** imerge(**int** pos){

Entry e1 = entries[pos];

Entry e2 = entries[pos+1];

e1.I.e = e2.I.e;

removeEntry(e2);

**if**(usedSpace < tree.getMinimumLoad())

nmerge();

}

对于第二种情况，由于涉及到结点之间的区间操作，所以比较复杂。我们需要先判断两个结点的容量情况。如果左边结点的容量大于最大容量的一半，那么把左边结点的最后一个区间合并到右边结点中去；否则把右边结点的第一个区间合并到左边结点中去，若合并后右边结点容量不足一半，则需要进行结点合并。具体代码如下：

/\*\*

\* merge the adjacent intervals that belong to two different leaves

\* **@param** p the least common ancestor of N1 and N2

\* **@param** n1 the left leaf node

\* **@param** n2 the other adjacent leaf node

\* **@param** pos the position where n1 rooted at p. Obviously, n2's position is pos+1

\*/

**protected** **void** imerge2(AbstractNode p, AbstractNode n1, AbstractNode n2, **int** pos){

Entry e1 = n1.entries[n1.usedSpace-1];

Entry e2 = n2.entries[0];

Entry pe1 = p.entries[pos];

Entry pe2 = p.entries[pos+1];

**if**(n1.usedSpace > tree.getMinimumLoad()) {

//merge the n1's last interval into n2

p.removeEntry(pe1);

p.removeEntry(pe2);

p.insertEntry(**new** Entry(**new** Interval(pe1.I.s, e1.I.s, pe1.I.v, pe1.I.set), pe1.child));

p.insertEntry(**new** Entry(**new** Interval(e1.I.s, pe2.I.e, pe2.I.v, pe2.I.set), pe2.child));

n1.removeEntry(e1);

n2.removeEntry(e2);

//float d = e1.I.v - pe2.I.v;

/\*ArrayList s = new ArrayList<String>();

s.addAll(e1.I.set);

s.removeAll(pe2.I.set);\*/

AbstractNode n = pe2.child;

**while**(n != n2){

**for**(**int** i = 0; i< n.usedSpace; i++)

**if**(n.entries[i].I.contains(e2.I)){

n.entries[i].I.s = e1.I.s;

n = n.entries[i].child;

**break**;

}

}

n2.insertEntry(**new** Entry(**new** Interval(e1.I.s, e2.I.e, e2.I.v, e2.I.set),**null**));

}

**else** {

//merge the n2's first interval into n1

p.removeEntry(pe1);

p.removeEntry(pe2);

p.insertEntry(**new** Entry(**new** Interval(pe1.I.s, e2.I.e, pe1.I.v, pe1.I.set),pe1.child));

p.insertEntry(**new** Entry(**new** Interval(e2.I.e, pe2.I.e, pe2.I.v, pe2.I.set),pe2.child));

n2.removeEntry(e2);

n1.removeEntry(e1);

AbstractNode n = pe1.child;

**while**(n != n1){

**for**(**int** i = 0; i< n.usedSpace; i++)

**if**(n.entries[i].I.contains(e1.I)){

n.entries[i].I.e = e2.I.e;

n = n.entries[i].child;

**break**;

}

}

n1.insertEntry(**new** Entry(**new** Interval(e1.I.s, e2.I.e, e1.I.v, e1.I.set),**null**));

**if**(n2.usedSpace < tree.getMinimumLoad())

n2.nmerge();

}

}

### 区间调整代码实现

区间调整主要是在遍历SB\*-Tree后，对那些不满足持续时间要求的区间进行的操作。代码如下所示。

/\*\*

\* adjust intervals whose (end - start) < duration

\* **@param** duration given constraint of time interval duration

\* **@param** granularity

\* **@return**

\*/

**public** **int** adjustInterval(**int** duration, String granularity) {

Calendar c3 = Calendar.*getInstance*();

**int** i, j, plus, start, end, max = 0;

Time t = **new** Time();

Set<Integer> intervalSet = intervalResult.keySet();

ArrayList<Integer> intervalList = **new** ArrayList<Integer>();

**for**(Integer eachTime : intervalSet) intervalList.add(eachTime);

**for**( i = 1; i< intervalList.size()-1; i++){

plus = 1;

//prune

**if**( (intervalResult.get(intervalList.get(i)).size()) < max)

**continue**;

//判断当前区间是否满足窗口大小，如果不满足，计算需要向下合并的区间数量

**while**( (i+plus < intervalList.size()-1) && ((intervalList.get(i+plus) - intervalList.get(i)) < duration)){

plus++;

}

//merge

Set<String> s = **new** TreeSet<String>();

**for**(j=i;j<=(i+plus);j++){

**if**(j==i) {

s = intervalResult.get(intervalList.get(j));

}

**else** {

s.retainAll(intervalResult.get(intervalList.get(j-1)));

}

}

//合并完成后还需要进行一步：判断与接下来的区间合并后的用户集合是否保持不变，如果保持不变，则继续向下合并；如果改变，则结束合并。

**while**(**true**){

Set<String> temp = **new** TreeSet<String>();

Set<String> replace = **new** TreeSet<String>();

temp.addAll(s);

replace.addAll(s);

temp.retainAll(intervalResult.get(intervalList.get(j-1)));

replace.removeAll(temp);

**if**(!replace.isEmpty()) **break**;

**else** j++;

}

start = intervalList.get(i);

end = intervalList.get(j-1);

String interval;

**if**(granularity.equals("null")){

interval = "[" + start + "," + end +"]";

}

**else** **if** (granularity.equals("second")){

interval = "["+t.transformFromSecond(start)+","+ t.transformFromSecond(end)+"]";

}

**else** {

interval = "[" + t.transformFromMinute(start)+","+t.transformFromMinute(end)+"]";

}

**if** (finalIR.isEmpty()) {

finalIR.put(interval, s);

**if**(s.size() > max )

max = s.size();

}**else** {

String preInterval = finalIR.lastKey();

TreeSet preSet = (TreeSet) finalIR.get(preInterval);

**if**(preSet.equals(s)) {

String[] time = preInterval.split(",");

**int** preIntervalStart;

**int** preIntervalEnd;

**if**(granularity.equals("null")){

preIntervalStart = Integer.*parseInt*(time[0].substring(1));

preIntervalEnd = Integer.*parseInt*(time[1].substring(0, time[1].length()-1));

}

**else** **if**(granularity.equals("second")) {

preIntervalStart = t.uniformToSecond(time[0].substring(1));

preIntervalEnd = t.uniformToSecond(time[1].substring(0, time[1].length()-1));

}

**else** {

preIntervalStart =t.uniformToMinute(time[0].substring(1));

preIntervalEnd = t.uniformToMinute(time[1].substring(0, time[1].length()-1));

}

**if**(preIntervalStart <= start && preIntervalEnd >= end){

//do nothing

}

**else** **if**(start <= preIntervalStart && end >= preIntervalEnd) {

finalIR.remove(preInterval);

finalIR.put(interval, s);

}

**else** **if**( start < preIntervalEnd && end > preIntervalStart){

interval = "[" + (start < preIntervalStart?start : preIntervalStart) +

"," + (end > preIntervalEnd? end : preIntervalEnd) +"]";

finalIR.remove(preInterval);

finalIR.put(interval, s);

}

}

**else** {

finalIR.put(interval, s);

**if**(s.size() > max )

max = s.size();

}

}

}

Calendar c4 = Calendar.*getInstance*();

System.***out***.println("区间调整已完成. 花费时间(ms): " + (c4.getTimeInMillis() - c3.getTimeInMillis()) );

**return** max;

}

代码中判断每个区间是否满足持续时间要求，如果满足，直接保存到TreeMap类型的finalIR中。对于不满足的区间，按照区间调整的策略调整后，与finalIR的最后一个区间进行判断，判断两者是否存在覆盖关系且用户结合相同，如果是，则舍去较小的区间，把较大的区间添加到finalIR中。

## 本章小结

本章主要介绍了基于SB\*-Tree的查询算法。首先我们详细介绍了SB-Tree的相关知识，包括SB-Tree的结点结构以及各种相关操作。其次我们介绍了我们对SB-Tree所进行的改进，即SB\*-Tree的相关知识。最后，我们介绍如何基于SB\*-Tree来得到最优时间窗口的方法和过程。为了加深大家的理解，本章还根据一个例子具体演示了算法的每一个步骤。最后展示了算法核心操作的代码。

# 实验说明

本章通过实验比较第三章和第四章提出的两种算法的性能。在正式实验比较之前，我们将先介绍使用的实验数据。然后针对不同的影响因素做比较性实验，确定每个算法的影响因素。最后再比较两种算法的性能。

## 数据说明

本文实验的数据来自于某手机阅读软件在一个月内所有用户的行为数据。我们从中挑选出了用户进行翻页操作的数据，作为我们接下来实验的原始数据。原始数据总共有79166名用户，20175本电子书籍。原始数据的部分内容如图5.1所示。

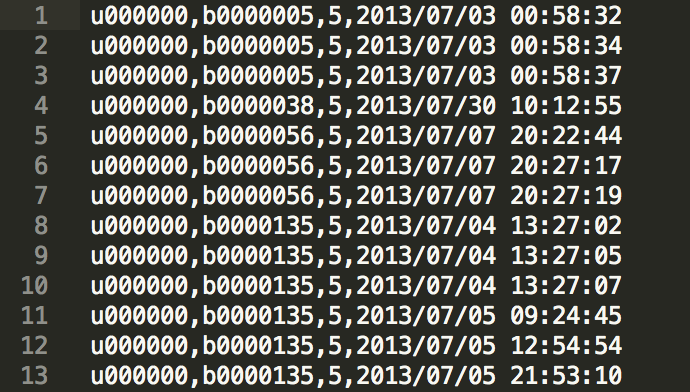


图 5. 1 原始数据内容

|  |  |
| --- | --- |
| **名称** | **说明** |
| userID | 用户的唯一ID |
| bookID | 用户发生翻页行为的对象书籍的唯一ID |
| type | 翻页的行为类型编号 |
| time | 用户翻页的时间 |

表 5 - 1 原始数据字段说明

数据中每个字段的含义如表5.1所示。原始数据中的数据以<userID, bookID>按升序排列，这样的数据不利于我们根据两次相近翻页时间来计算用户阅读书籍一页的时间。因为对于正在连载的书籍，用户可能在读完一本后马上读另一本书，然而对原来这本书的下次阅读则要等到更新后。所以用户可能在一次连续的阅读中对多本书进行阅读，而对同一本书的两次阅读可能相隔比较长的时间。所以我们对原始数据进行修改，使其按<userID, time>升序排列，并删除type列，因为type列属性对于实验没有影响。得到重新排序后的数据集后，我们计算每个用户的前后相邻两次时间差，单位为秒，作为我们计算阅读一页的最大时间的根据，从而把用户多次的翻页行为划分成多次连续的阅读行为。

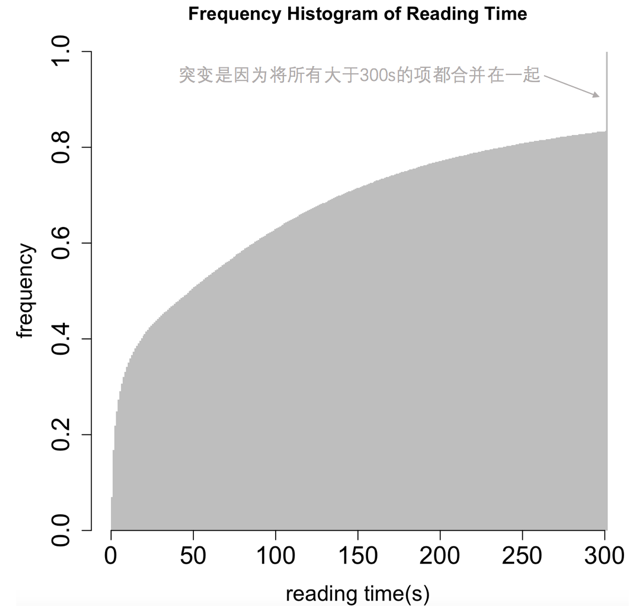


图 5. 2 阅读时间统计图

图5.2表示了阅读时间不超过t的频率。横坐标为阅读时间，纵坐标为频率。例如在300s处，频率为0.83，它的含义是83%的用户阅读时间在5分钟之内。根据这一标准，我们可以处理排序后的数据，得到每个用户的所有阅读区间。说明：如果用户的某两次操作时间间隔在5分钟以上，被认为是两次不同时间段的阅读。最后再去除掉区间开始时间和结束时间相同的异常数据，我们得到了每个用户的阅读时间段。用户的阅读时间段部分内容如图5.3所示

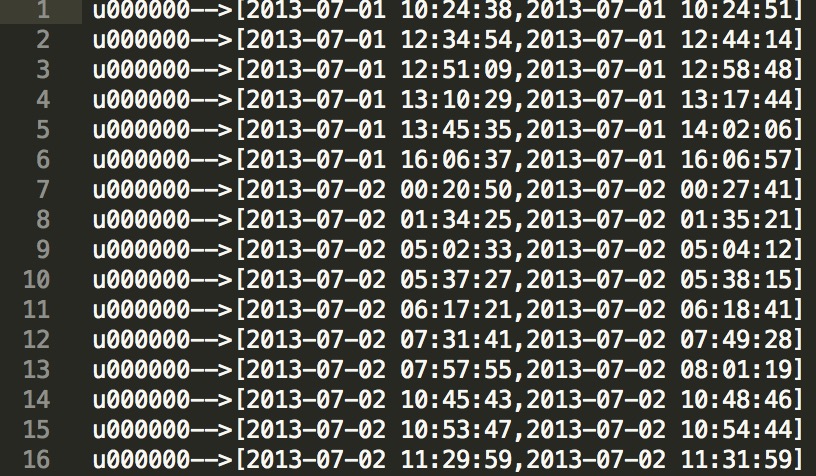


图 5. 3 用户阅读时间段内容

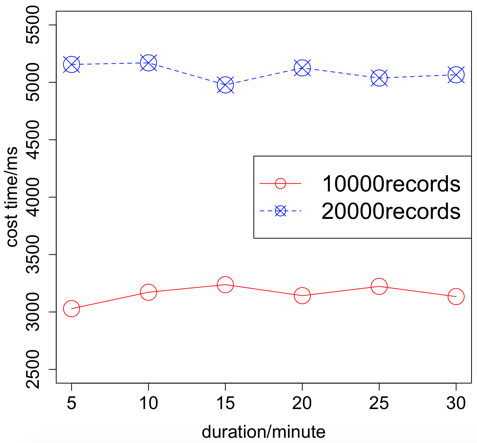
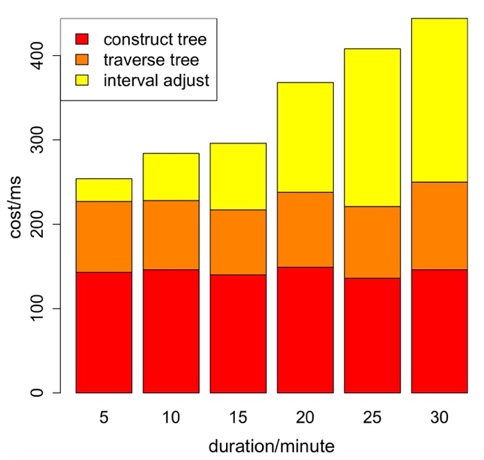
数据中第一列表示用户的userID，“-->”是分隔符，没有特殊含义。其后表示用户的阅读时间区间，区间有开始时间和结束时间，并且前后都是闭合的，其中时间粒度为1秒。我们可以看到一个用户可以有多个阅读时间区间。最终数据共有67549个用户，共387868条记录。

我们基于上述的数据，通过一定的调整，另外还得到时间粒度分别为1分钟、15分钟、30分钟和一个小时的数据集。

## 实验比较

### 时间窗口duration的影响

为了比较时间窗口duration对滑动时间窗口算法和SB\*-Tree算法的影响，我们要控制记录数、时间范围、时间粒度保持不变。我们基于小节5.1中的最终用户阅读时间段数据，从中提取出时间范围为一个月，时间粒度为一分钟的10000条记录和20000条记录的数据来作为本次实验数据。

a)滑动时间窗口算法 b)SB\*-Tree算法

图 5. 4 时间窗口duration对算法性能的影响

图5.4a)展示了不同的时间窗口duraiton大小对滑动时间窗口算法性能的影响。图中蓝色实线部分是算法处理10000条记录的数据的时间消耗，红色虚线部分是算法处理20000条记录的数据的时间消耗。我们可以看到，算法在不同的duration的情况下的运行时间有轻微的差异，但是这个差异相对于整体的时间消耗来说可以忽略不计，我们可以将其理解为是程序在运行时不可避免的性能差异，而不是由于duration的改变造成的。最终我们可以指出，在其他因素不变的情况下，时间窗口duration对滑动时间窗口算法的性能没有影响。

图5.4b)展示了SB\*-Tree的各个关键阶段在不同时间窗口duration的情况下的时间花费。我们可以看到在不同时间窗口duration的情况下，对于同一个数据集来说，SB\*-Tree的建树时间和遍历树时间几乎相同，而对于在区间调整所耗的时间差异比较大。这是因为当duration增大时，需要进行区间调整的区间变多，从而增加了区间调整整个过程的时间花费。所以，我们认为，在其他因素不变的情况下，时间窗口duration的增加，会使SB\*-Tree算法的整体时间增加，这主要是因为区间调整过程的时间增加。

### 时间粒度granularity的影响

我们基于多个时间范围为一星期、时间窗口为5分钟、有10000条记录的不同时间粒度的数据，比较了滑动时间窗口算法和SB\*-Tree的性能，最终得到了如图5.5所示的结果。由于算法的时间差异跨度过大，我们比较log(t)的大小。

从图中可以看到，对于滑动时间窗口算法来说，在时间粒度增大时，算法的运行时间减小了。并且减小的程度与时间粒度增大的程度有关。例如图中时间粒度1分钟相对于1秒钟增大了59倍，而时间粒度15分钟只相对于1分钟的粒度增大到14倍。所以时间粒度1分钟的算法运行时间相对于时间粒度为1秒钟的算法运行时间的减少的程度比15分钟的算法时间相对于1分钟的算法时间减少程度大。

对于SB\*-Tree算法来说，当时间粒度granularity大于时间窗口duration 时，算法的性能随时间粒度的改变时变化不大。而当时间粒度granularity小于时间窗口duration时，算法的运行时间随时间粒度的增大而减少。这是因为当时间粒度小于时间窗口时，时间粒度越大，在区间调整时需要调整的区间和每次调整的操作就相对减少，从而减少了算法的总体运行时间。但是时间粒度granularity对滑动时间窗口算法的影响程度比对SB\*-Tree的影响程度要大。

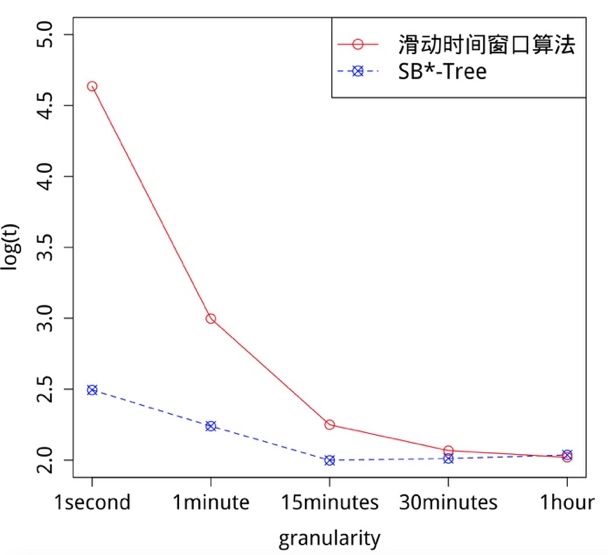
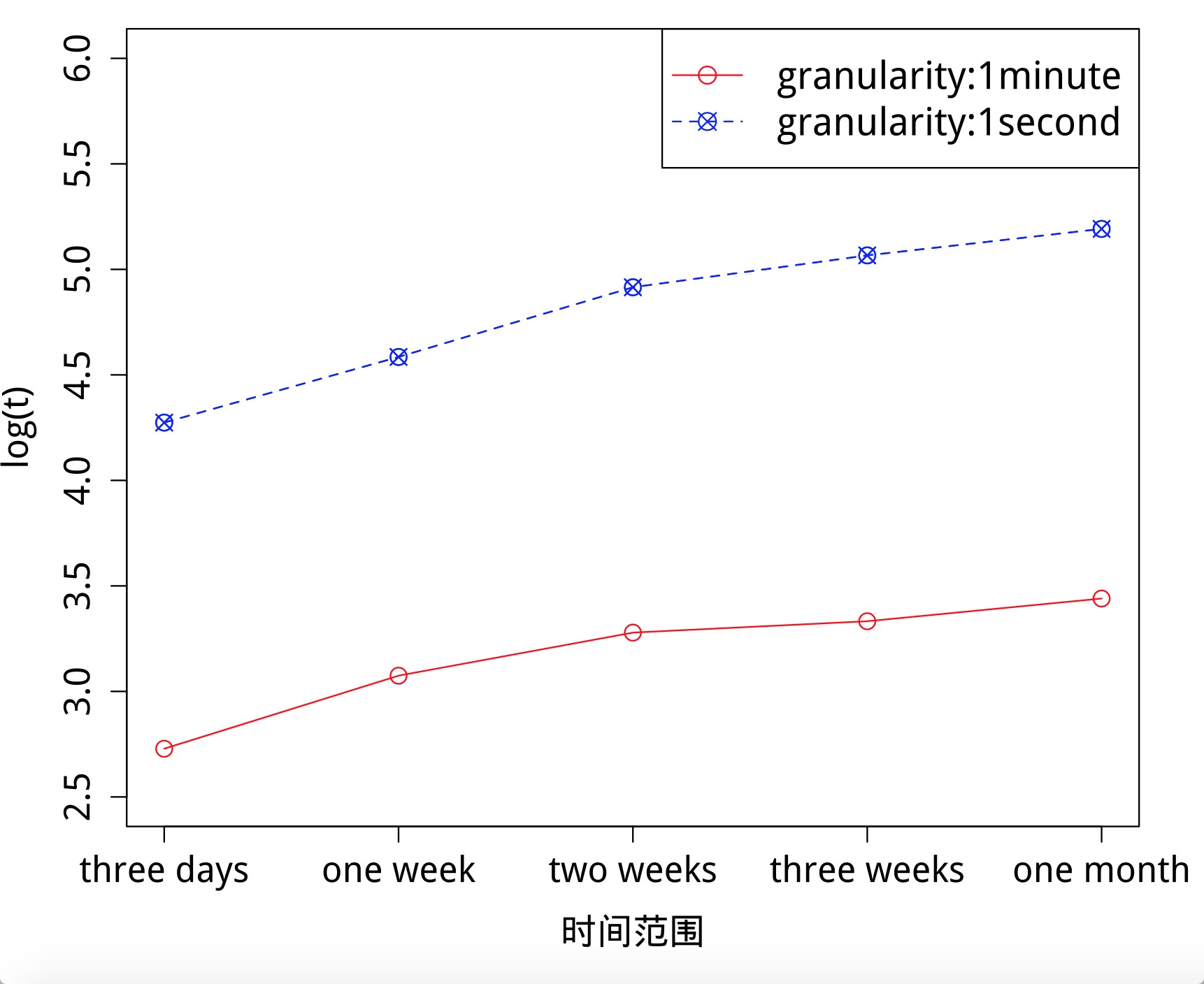
 

图 5. 5 不同时间粒度下的算法运行时间图 图 5. 6 时间范围对滑动时间窗口算法影响图

### 时间范围的影响

所谓的时间范围就是我们根据用户提交的时间信息得到的和之差。在图5.6中，展示了时间范围对滑动时间窗口算法性能的影响。图中横坐标分别是三个不同的时间范围，分别为一个星期，两个星期和一个月。纵坐标是算法的运行时间的取对数之后的值。我们比较了分别在1秒和1分钟两种不同的粒度下时间范围对算法性能的影响，因为两者运行时间相差了多个数量级，所以我们取对数来进行比较。从图中我们可以看到随着时间范围增大，不管是在1秒的时间粒度，还是1分钟的时间粒度下，滑动时间窗口算法的运行时间都增大了。这是因为随着时间范围增大，算法扫描时间线的次数增加了。所以，我们可以指出，时间范围的变化会对滑动时间窗口算法性能产生影响，并且时间范围越大，算法运行时间越长。

### 数据量的影响

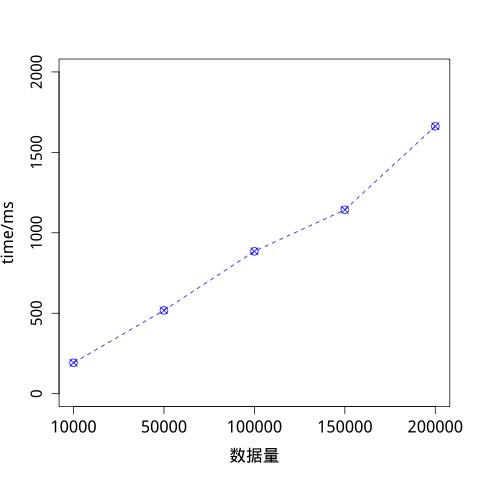
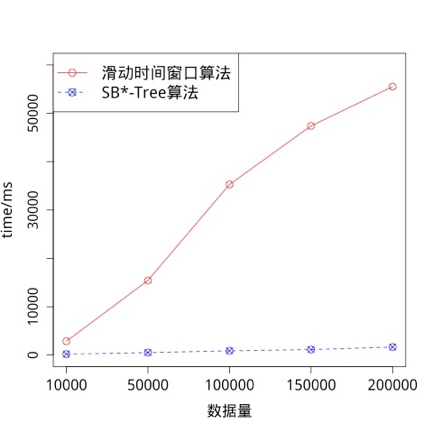


图 5. 7 数据量对算法性能的影响

在这一节中，我们比较数据量对两种算法的性能的影响。我们以原始数据为基础，选出10000、50000、100000、150000、200000条记录的四种规格的数据集，并进行实验。最终实验结果如下图5.8所示。图中横坐标表示记录的数量，纵坐标表示算法运行不同数据集对应的时间消耗，单位为毫秒。因为图5.7中左图中SB\*-Tree的算法性能变化相对于滑动时间窗口算法来说比较小，不易观察，所以我们又单独划了SB\*-Tree算法的性能的图，如图5.7右图所示。从两幅图中我们可以看到，随着数据量的增大，算法的运行时间都增加了，且两者的运行时间大致呈线性增加。

### 参数b和l对SB\*-Tree算法的影响

在第四章中已经提到SB\*-Tree会继承SB-Tree的结点结构，而决定结点结构的有两个参数b和l，它们分别决定了SB\*-Tree的中间结点和叶子结点的最大容量。当b和l改变时，树的整体结构会发生改变，遍历树的效率也会发生改变。究竟b和l取什么值才能使得算法性能达到最高效？为了解决这个问题，我们进行实验来研究不同的b和l取值对SB\*-Tree算法性能的影响。因为b和l的取值不会影响SB\*-Tree算法区间调整时的性能，所以我们只需考虑不同b和l的取值下，构造SB\*-Tree和遍历SB\*-Tree算法的时间花费。

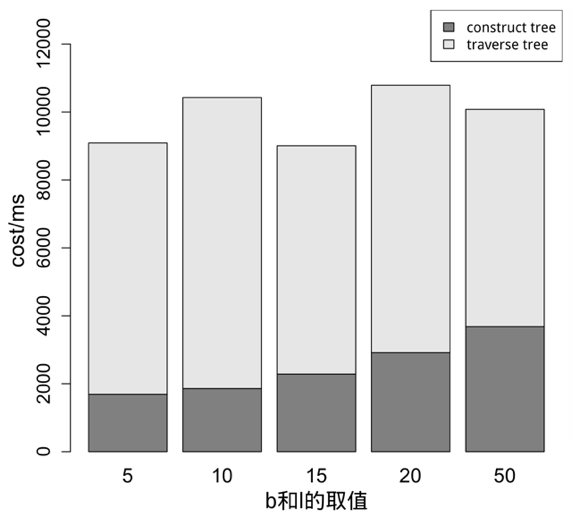


图 5. 8 参数b和l对SB\*-Tree性能影响图

图5.8展示了不同b和l的取值对SB\*-Tree建树和遍历树的性能影响。为了实验的方便，我们让b和l始终保持相等。我们共选取了5组不同的取值，分别为5，10，15，20和50。从图中我们可以看到，随着取值的不断增大，构建SB\*-Tree的时间花费随着增大，这是因为每次插入一个新的区间时，都要与结点的所有区间进行是否相交的判断。当取值增大时，每个结点能够容纳的最大区间数量增大，那么需要进行判断的次数也增加，从而增加了构造树的时间。而遍历树的时间花费随b和l的取值的增大没有明显的变化趋势，时加时减。所以为了确定最优的b和l取值，我们需要进行一系列的取值来确定。

### 算法性能比较

在之前的讨论中，我们已经发现对于滑动时间窗口算法和SB\*-Tree有许多影响因素。在这一小节中，我们横向比较滑动时间窗口算法和SB\*-Tree算法对于同一数据集的运行时间，从而比较两者的性能。如图5.9所示，图中横坐标1到5代表了5次不同的实验。在第一次实验中，我们使用的数据集的粒度为1秒，共包含387868条记录，给定的时间窗口为300秒。在第二次实验中，我们使用的数据集的粒度为1分，共包含319771条记录，给定的时间窗口为5分钟。在第三次实验中，我们使用的数据集的粒度为15分，共包含128599条记录，给定的时间窗口为30分钟。在第四次实验中，我们使用的数据集的粒度为30分钟，共包含90183条记录，给定的时间窗口为60分钟。在第五次实验中，我们使用的数据集的粒度为1小时，共包含62248条记录，给定的时间窗口为2小时。五次实验数据集的时间范围都是一个月。纵坐标仍然是算法运行时间取对数的值。

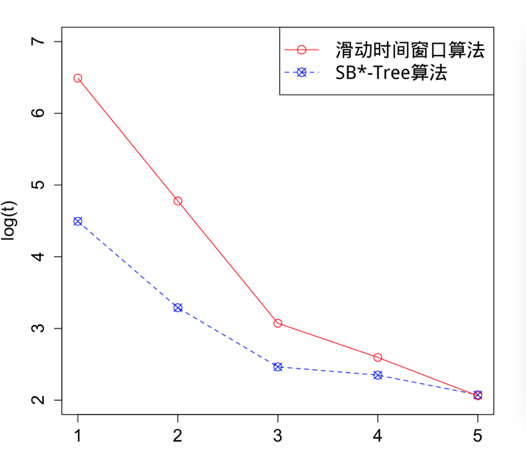


图 5. 9 算法运行时间图

从图中我们可以看到，在前四次实验中，SB\*-Tree算法的性能优于滑动时间窗口算法，并且随着时间粒度的增大和数据量的减少，两者算法的运行时间差距越来越小。而在第五次实验中，两者的运行时间是差不多。这是由于五次实验的数据集的时间粒度和数据量不同导致的。我们可以指出：在时间范围相同的情况下，滑动时间窗口算法在时间粒度小，数据量大的情况下的性能十分差。例如在第一次实验中，算法运行时间高达50分钟。而SB\*-Tree算法在时间粒度小、数据量大的情况下仍能够保持比较好的性能。在第一次实验中，SB\*-Tree算法仅需要30秒钟。而当算法时间粒度较大、数据量比较小的情况下（实验5），SB\*-Tree算法和滑动时间窗口算法的性能相差不大。

## 本章小结

本章对第三章和第四章提出的两种算法进行了详细的实验比较。首先通过实验纵向分析了影响算法的各个因素，主要包括时间窗口、时间粒度、时间范围和数据量。然后通过横向比较两个算法在不同规格的数据集下的性能。

# 总结

本文主要针对最优区间覆盖问题提出了两种解决算法：滑动时间窗口算法和基于SB\*-Tree的查询算法。滑动时间窗口思路比较简单，但是当数据量比较大或者时间范围比较长时效率非常地低。为了解决这个问题，我们对SB-Tree数据结构进行改进，在原来的结点结构基础上增加了一维集合属性，我们称之为SB\*-Tree。另外我们还针对SB\*-Tree设计了相应的查询算法来得到最终结果。最后，我们通过实验比较了各种参数对两种算法性能的效率影响，以及两个算法针对相同数据集的效率比较。

通过实验比较发现，在数据量比较大的情况下，SB\*-Tree的构造和遍历都十分高效，但是在区间调整阶段耗时比较久。在整个SB\*-Tree的算法运行中，区间调整占了大部分的时间消耗。这一阶段的时间消耗是有待减少的。我们希望能够加快从SB\*-Tree到最终结果这一过程的效率，解决的大致方法可以有两种：1、优化区间调整阶段的效率。可以在区间调整时加入新的优化策略来降低调整的次数。2、设计新的遍历算法，直接从SB\*-Tree得到满足时间窗口duration要求的区间，而不需要进行区间调整。

# 参考文献

1. 包磊, 秦小麟. 时空聚集计算研究进展[J]. 计算机科学, 2006, 33(1):22-24.
2. Snodgrass R T, Gomez S, Mckenzie L E. Aggregates in the Temporal Query Language TQuel.[J]. IEEE Transactions on Knowledge & Data Engineering, 1993, 5(5):826-842.
3. Yang J, Widom J. Incremental computation and maintenance of temporal aggregates[C]//Data Engineering, 2001. Proceedings. 17th International Conference on. IEEE, 2001: 51-60.
4. Kline N, Snodgrass R T. Computing temporal aggregates[C]//Data Engineering, 1995. Proceedings of the Eleventh International Conference on. IEEE, 1995: 222-231.
5. Kline R N, Kline R N. Aggregation in temporal databases[J]. 1999.
6. Kaufmann M, Manjili A A, Vagenas P, et al. Timeline index: A unified data structure for processing queries on temporal data in SAP HANA[C]//Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. ACM, 2013: 1173-1184.
7. Kim J S, Kang S T, Kim M H. On temporal aggregate processing based on time points[J]. Information Processing Letters, 1999, 71(5): 213-220.
8. Moon B, López I F V, Immanuel V. Scalable algorithms for large temporal aggregation[C]//Data Engineering, 2000. Proceedings. 16th International Conference on. IEEE, 2000: 145-154.
9. Moon B, Lopez I F V, Immanuel V. Efficient algorithms for large-scale temporal aggregation[J]. Knowledge and Data Engineering, IEEE Transactions on, 2003, 15(3): 744-759.
10. Gendrano J A G, Huang B C, Rodrigue J M, et al. Parallel algorithms for computing temporal aggregates[C]//Data Engineering, 1999. Proceedings., 15th International Conference on. IEEE, 1999: 418-427.
11. Ye X, Keane J A. Processing temporal aggregates in parallel[C]//Systems, Man, and Cybernetics, 1997. Computational Cybernetics and Simulation., 1997 IEEE International Conference on. IEEE, 1997, 2: 1373-1378.
12. Zhang D, Markowetz A, Tsotras V, et al. Efficient computation of temporal aggregates with range predicates[C]//Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. ACM, 2001: 237-245.
13. Douglas Comer. The ubiquitous B-tree[C]// 1979:121--137.
14. Guttman A. R-trees: a dynamic index structure for spatial searching[M]. ACM, 1984.
15. Tao Y, Papadias D, Faloutsos C. Approximate temporal aggregation[C]//Data Engineering, 2004. Proceedings. 20th International Conference on. IEEE, 2004: 190-201.
16. Kanellakis P C, Ramaswamy S, Vengroff D E, et al. Indexing for data models with constraints and classes[C]//Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. ACM, 1993: 233-243.
17. Vengroff D E, Vitter J S. Efficient 3-d range searching in external memory[C]//Proceedings of the twenty-eighth annual ACM symposium on Theory of computing. ACM, 1996: 192-201.
18. Arge L. On two-dimensional indexability and optimal range search indexing [J]. IEEE Computer Society, 1995, 44(1):13-19.
19. Samet H. Applications of spatial data structures[J]. 1990.
20. Moore A W. An intoductory tutorial on kd-trees[J]. 1991.

# 致谢

毕业设计已经接近了尾声，这也意味着我的大学生活就要结束了，学生活一晃而过，回首走过的岁月，心中倍感充实，当我写完这篇毕业论文的时候，有一种如释重负的感觉，感慨良多。

首先我十分感谢我的导师范菁老师。范菁老师为人友善，性格温柔，她高尚的人格潜移默化地对我产生着影响。范菁老师在学术上严谨的态度让我督促自己对学术保持端正的态度。范菁老师不仅在学习上为我提供很多锻炼的机会，也在生活中给予我很大的帮助，能够引导我走出迷茫。选范菁老师作为导师是我最正确的决定之一。

其次，我要感谢我的二导曹斌老师，能够给我这么一个有意义的研究课题。虽然在完成的过程中遇到了很多的问题，但是在曹斌的帮助以及自己的努力下都都成功地解决了。每周我都固定与曹斌老师讨论我的毕业设计进度以及遇到的困难，老师都能够不厌其烦地给予我具有指导性的建议。当我在完成算法代码时遇到算法性能过差的问题时，曹斌能够帮助我分析原因，解决问题。同时在我后期撰写论文时也为我指明整体的大纲思路，指出我的论文上存在的格式问题以及不规范、不正确的地方，督促我加以改正。曹斌兢兢业业的态度使我由衷的敬佩，在此谨向范菁老师和曹斌老师置以最诚挚的谢意与最崇高的敬意。

另外，我也十分感谢我的同班同学，大学四年里最宝贵的收获就是拥有各位同学，谢谢你们给我带来了宝贵的经历和回忆。

最后，还要感谢对这次毕业设计论文进行评阅的各位老师专家们，辛苦了。

这次毕业设计虽然完成的过程十分漫长，但是也为我提供了宝贵的经验，为我以后研究生阶段的论文奠定了基础。这次毕业既是结束，也是新的开始，希望大家毕业以后都能够朝着自己的目标继续努力。

# 附录

### 附录1 毕业设计文献综述

### 附件2 毕业设计开题报告

### 附件3 毕业设计外文翻译（中文译文与外文原文）