面向最优时间窗口覆盖的查询服务

曹 斌 侯晨煜 范 菁 程时伟 邱杰凡
(浙江工业大学计算机科学与技术学院 杭州 310023)

摘 要 该文提出了最优时间窗口覆盖查询问题,该问题是指给定多个用户和对应的时间区间以及持续时间大小的要求,希望找到既满足持续时间大小要求又能够被最多用户覆盖的时间区间段.该问题的解决能够为现实生活提供多种服务,比如:安排直播时段、云服务收费等.根据我们的调查发现,该问题属于时态数据库领域,但是与现有的时态数据库领域问题都不一样,导致目前没有现成的方法能够直接解决最优时间窗口覆盖问题.该文针对该问题设计了一种算法——基于Timeline Index的查询算法(TLI算法),该算法利用Timeline Index数据结构存储原始数据信息.再通过Timeline Index利用相邻的时间点构造多个时间区间,对于不满足时间大小的时间区间,我们设计了专门的调整机制使它们变成新的满足要求的时间区间.我们通过理论证明该算法的时间复杂度$O(N\log N)$,其中N是原始数据中记录的数量.最后通过实验分析,发现TLI算法的运行效率比基准算法普遍快了1个数量级.

关键词 时态数据库;查询服务;最优时间窗口覆盖;时间区间;Timeline Index
中图法分类号 TP311 **DOI号** 10.11897/SP.J.1016.2018.01882

Toward the Query Service For the Optimal Time Window Covering

CAO Bin HOU Chen-Yu FAN Jing CHENG Shi-Wei QIU Jie-Fan
(*Computer Science and Technology College, Zhejiang University of Technology, Hangzhou* 310023)

Abstract In this paper, we propose an interesting problem about temporal query, named optimal time window covering problem. It refers to that given multiple users with corresponding time intervals and a constraint of interval size, how to find the optimal time intervals satisfying the following two constraints: (1) the size of intervals reported must be bigger than or equal to the given size constraint. (2) the intervals can be covered by users as many as possible. The solution of the problem can provide a variety of services for the real life, such as: live show arrangement and cloud services etc. For example, an education host wants to hold a live show and ensures there are most audience so that he can get most profit. To this end, he can gather audience's free time in advance and get the optimal time to live by using our method. According to our survey, this problem belongs to temporal database. However, it is different from existing problem in temporal database, which results in that there is no existing method can solve this problem directly. At first glance, this problem is similar to temporal aggregation because both of them query the overall result of tuples according to temporal information. However, there are two differences between them. Firstly, temporal aggregation is used to query the result in a given interval while our problem has no idea about the exact interval position. Optimal time window covering problem merely has a

《计算机学报》

收稿日期:2017-06-18;在线出版日期:2018-01-10.本课题得到国家重点研发计划(2016YFB1001403)、国家自然科学基金(61602411,61572437,61772468,61502427)、浙江省自然基金(LY16F020034)、浙江省重大科技专项重点工业项目(2015C01034,2015C01029)、杭州市重大科技创新项目(20152011A03)资助.**曹 斌**,男,1985年生,博士,副教授,中国计算机学会(CCF)会员,主要研究方向为时空数据管理.E-mail:bincao@zjut.edu.cn.**侯晨煜**,男,1994生,硕士研究生,中国计算机学会(CCF)学生会员,主要研究方向为服务计算、大数据.**范 菁**(通信作者),女,1969年生,博士,教授,中国计算机学会(CCF)会员,主要研究领域为服务计算、虚拟现实.E-mail:fanjing@zjut.edu.cn.**程时伟**,男,1981年生,博士,副教授,中国计算机学会(CCF)会员,主要研究方向为人机交互技术.**邱杰凡**,男,1984年生,博士,副教授,主要研究方向为物联网、传感器网络.

constraint of interval size so that we need to traverse all possible intervals to get final result. Secondly, temporal aggregation will count the tuples whose interval intersects with given interval while we should count the tuples whose interval completely covers the optimal interval. Therefore, the algorithms of temporal aggregation cannot be used to solve our problem directly. In order to solve this problem efficiently, in this paper, we propose an algorithm for this problem, named the algorithm based on Timeline Index(TLI). We utilize the data structure of Timeline Index to store the information of original dataset. The Timeline Index consists of two data structures, namely VersionMap and EventList. Specifically, we keep track of the endpoints for each interval into VersionMap, and store the sequence of activation or invalidation for users in EventList. Then we can combine adjacent instant time and construct multiple intervals by scanning VersionMap and EventList. However, these intervals may not satisfy the constraint of given size. Therefore, we conduct an adjustment process where we extend those intervals to longer intervals so that they can satisfy the given constraint. Meanwhile, we introduce a pruning strategy which can significantly reduce the number of intervals when we traverse Time line Index. Besides, we prove theoretically that the time complexity of this algorithm is $O(N\log N)$, where N is the number of records in original dataset. Finally, we perform extensive experiment based on a real-world dataset and a synthetic dataset to study the efficiency of TLI and analyze the influential factors for it. Experimental results show that the efficiency of TLI is one order of magnitude faster than a baseline algorithm.

Keywords　temporal database; query service; optimal time window covering; time interval; Timeline Index

1　引　言

本文提出最优时间窗口覆盖问题，该问题是指：给定一个用户集合 U，集合中每个用户都有各自的有效时间区间.给定一个非负的持续时间 $duration$，我们希望找到所有的时间窗口满足以下两个条件：(1)时间窗口的大小至少为 $duration$；(2)最多的用户在该时间窗口内一直有效(用户有一个有效时间区间包含该时间窗口).如图1所示，纵向坐标轴上 A、B、C、D 表示4个用户，他们分别拥有各自的有效时间区间，如 B 用户拥有两个有效时间区间(8:00～11:00,14:00～18:00).给定持续时间 $duration$＝3h，我们可以找到最优解是由 t_3＝14:00 和 t_4＝18:00 组成的时间区间[t_3，t_4]，它的用户集合为{A,B,C}，即存在3个不同的用户的时间区间包含[t_3,t_4].虽然图1中 t_1＝8:00 和 t_2＝11:00 组成的时间区间[t_1,t_2]满足持续时间要求，但是它的用户集合只有两个用户{B,D}，没有达到用户数最多的要求，因此不是最优解.注意，在该问题中，我们事先并不知道最优的时间窗口位置，它可以表示一维时间上的任意一个区间段，因此如何高效地查找该时间窗口位置是最优时间窗口覆盖查询问题的主要难点.

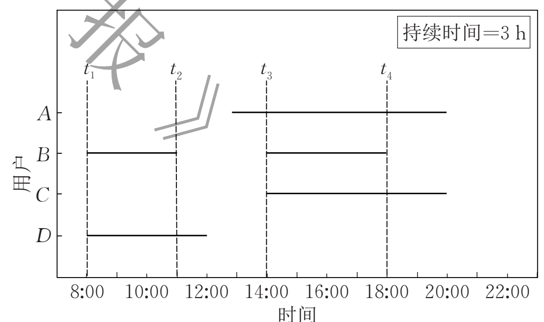


图1　最优时间覆盖查询问题

最优时间窗口覆盖问题的高效解决，能够为现实中许多应用场景提供解决方案.例如当下流行的在线收费教育直播应用中，为了进行一场教育直播，现在普遍的做法是由直播发起方指定一个直播时间段，由参与者自己报名参加.然而这样的直播时间选择方式存在一个问题：如果安排的时间段不合理，许多本来对该教育直播感兴趣的参与者就会无法参加.为了获得最大的潜在收益，直播发起方应该在保证有最多参与者的时间段内进行直播.我们可以根据所有参与者提交的空闲时间段信息，选择有最多参与者空

闲(即能够观看直播)的时间段进行直播.另外,最优时间窗口覆盖问题还能应用于云资源流量计费场景,通过分析流量最大的时间段来进行适当的收费.

解决最优时间窗口覆盖问题,一种最直接的求解方式是滑动时间窗口算法:构建一个窗口大小为 *duration* 的时间窗口,我们将该时间窗口沿着由所有用户时间段构成的时间线进行滑动,记录每次滑动后的用户覆盖结果,直到窗口滑动到时间线末尾后便可判断最优的窗口位置,即对应最多用户的最优时间区间.因为每当时间窗口沿着时间线滑到新的位置时,都需要对所有用户的所有区间进行判断,所以该算法计算复杂度高.一种对滑动时间窗口算法改进的方法是只考虑所有用户时间段的端点,利用这些时间端点构造满足 *duration* 大小的时间区间,并对所有用户的所有区间进行判断,输出最优结果.这样的方法虽然在构造时间区间时进行了优化,但是在为每个时间区间计算覆盖的用户集合时复杂度仍然很高,所以需要设计一种更加高效的算法.

此外,传统时态数据库领域中的时态聚合查询优化算法并不能解决本文所提出的最优时间窗口覆盖查询问题.首先,虽然两个问题考虑的都是区间上的整体信息,即进行 count 操作.但是时态聚合查询的区间是由查询者给出的一个固定的区间,而最优时间窗口覆盖问题中的区间只有大小要求,而没有固定的位置.其次,时态聚合查询中的计数要求与最优时间窗口覆盖问题中的计数要求不同.在时态聚合查询中,用户的一个有效时间区间与查询区间相交,则会对该用户计数.但是在最优时间窗口覆盖问题中,用户的有效时间区间必须覆盖查询区间,才会对该用户计数.计数要求的不同导致了传统时态聚合的索引结构无法直接用于解决该问题.因此,基于以上考虑,需要设计新的算法用来解决最优时间窗口覆盖查询问题.

在本文中,我们提出了一种基于 Timeline Index 的查询算法(TLI 算法). Timeline Index[1] 是 SAP HANA[2] 数据库中针对时序数据查询的专用数据结构,它主要用来解决时态聚合、时态连接查询.我们利用这种数据结构,设计了一套新的查询算法(TLI 算法)来支持最优时间窗口覆盖查询. TLI 算法主要分为三步:(1) 构建区间信息表.首先,我们利用 Timeline Index 存储原始数据集中的时间和用户信息.然后通过遍历 Timeline Index 得到由相邻时刻组成的连续时间区间,同时记录能够覆盖各个区间的用户集合,作为每个区间对应的用户信息,这些信息构成区间信息表;(2) 区间调整.由第一步得到的区间信息表中有部分时间区间会不满足持续时间 *duration* 要求.针对这些区间,我们要进行区间调整,生成新的满足 *duration* 的时间区间.在这一步中,我们设计了一种剪枝策略,直接舍去明显不是最优解的时间区间,不对它们进行区间调整;(3) 计算最终解.在这一步中,我们首先需要对每个时间区间进行求完备解:在保证用户集合不变的前提下,将每个时间区间扩展到最大.若不进行此操作,将无法得到正确的结果.在求完备解的过程中,我们同时记录用户数量最大的时间区间作为最优解.得到最优解后,再进行解的规范化:如果最优解中有两个相交且用户集合相同的时间区间,则把两个时间区间合并成一个新的时间区间.通过以上步骤,我们可以为最优时间窗口查询返回满足持续时间的最优时间区间以及对应的用户集合.

本文的主要贡献有如下几点:

(1) 本文提出了最优时间窗口覆盖问题,该问题的解决能够为许多时间驱动的服务提供通用的解决方案.

(2) 为了解决最优时间窗口覆盖问题,本文提出了基于 Timeline Index 的查询算法.该算法的时间复杂度为 $O(N\log N)$,其中 N 是原始数据中的记录数量.

(3) 本文基于用户真实在线阅读数据集,通过一系列实验分析比较了 TLI 算法的性能,同时通过实验分析了算法性能的影响因素.

本文第 2 节介绍最优时间窗口覆盖问题的预备知识和问题定义;第 3 节介绍基于 Timeline Index 的查询算法;第 4 节基于真实与合成数据进行大量实验评估和分析;第 5 节介绍相关工作;最后第 6 节进行本文总结和指明未来工作.

2 预备知识

在这一节中,我们首先介绍一些有助于理解问题的基本概念,主要包括数据对象、持续时间、覆盖的定义和解的规范化.接着介绍算法过程中涉及的数据结构:区间信息表、Timeline Index.最后给出时间窗口最优覆盖问题的正式定义.

2.1 基本概念

数据对象. 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

《计算机学报》

以来自用户行为日志当中. 具体的数据主要包含两项内容，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. 以阅读书籍场景为例，用户可能会在多个时间段读书，所以该用户就会有多个有效时间区间.

持续时间. 持续时间(Duration Time)是指用户不间断进行某项活动的一段时间大小，例如用户从 22:00 至 22:30 持续进行了持续时间为 30 min 的阅读活动. 在本文的问题中，持续时间作为输入，是时间区间的大小要求. 若要求持续时间为 30 min，则所求的最优时间区间的大小必须大于 30 min.

覆盖. 对于某一区间$[t_{start}, t_{end}]$，若某用户的某一时间区间 interval 满足$[t_{start}, t_{end}] \subseteq interval$，则该用户覆盖该区间$[t_{start}, t_{end}]$.

解的规范化. 如果求得的最终结果有一些区间相交，并且拥有相同的用户集合，就把这些区间合并，形成新的区间. 新区间的用户集合与原来旧区间的用户集合相同，同时舍去旧区间，我们把这样的操作称为解的规范操作. 例如，假设表 1 是求得的最终结果，假设要求时间窗口大小 $duration=5$，此时求得的最优区间有[0,5]和[3,8]，它们相交且对应的用户集合都为$\{A, B\}$，所以要把两个区间进行合并，合并成新的区间[0,8]，新区间的用户集合仍为$\{A, B\}$，如表 2 所示.

表 1 未规范化的解

区间	用户集合
[0,5]	$\{A, B\}$
[3,8]	$\{A, B\}$

表 2 规范化后的解

区间	用户集合
[0,8]	$\{A, B\}$

2.2 数据结构

区间信息表. 区间信息表(interval information table)用来存储在算法过程中得到的区间和对应的用户集合. 如表 3 所示，区间信息表中含两列属性，分别为 interval 和 user set. interval 存储在算法过程中得到的区间，user set 存储所有覆盖对应区间的用户所组成的集合. 区间信息表中一行完整的数据代表一个区间信息. 从表中我们可以看到区间[0,5]的有效用户有 A 和 B.

表 3 区间信息表

区间	用户集合
[0,5]	$\{A, B\}$
…	…

Timeline Index. 首先，为了有助于理解，我们需要介绍“事件”的概念. 用户的一个有效时间区间由开始时间 t_{start} 和结束时间 t_{end} 组成. 该用户在时刻 t_{start} 时变得有效(用户被激活)，在时刻 t_{end} 时失效. 事件分为“激活”事件和“失效”事件，其中激活事件是指用户在其有效区间的开始时间 t_{start} 变得有效；失效事件是指用户在其有效区间的结束时间 t_{end} 失效. 以表 4 为例，表中 3 个用户 A、B、C，每个用户有对应的各自的空闲时间区间. 图 2 是表 4 的图形表示. 其中用户 C 在时刻 1 时变得有效，称为一个“激活”事件. 用户 C 在时刻 8 时变得失效，称为一个“失效”事件.

表 4 用户有效时间区间

用户	区间
A	[5,9]
B	[2,6]
C	[1,9]

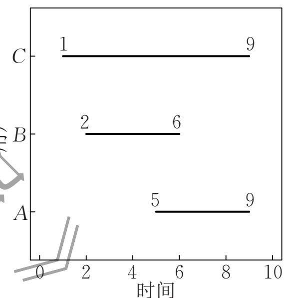


图 2 表 4 内容的图形表示

接下来介绍 Timeline Index 数据结构. Timeline Index 由两张表组成，分别是 EventList 和 VersionMap. EventList 有两列属性，分别为 $userID$ 和 $mark$. $userID$ 中存储的是用户的唯一标志，$mark$ 是用户变得有/失效的标记. 当用户变得有效时，$mark$ 标记为 1；当用户变得失效时，$mark$ 标记为 0. EventList 中完整的一行记录代表着一个事件：激活事件($mark$ 为 1)或者失效事件($mark$ 为 0). 所有的事件按照发生的先后顺序按次序存储在 EventList 中，即 EventList 中上一个事件肯定发生在下一个事件之前或两者同时发生，例如将表 4 的内容构造成的 Timeline Index 如图 3 所示. 其中图 3(b)的 EventList 中首先用户 C 变得有效，然后用户 B 变得有效，两个事件之间存在先后关系.

计算机学报

time	*eventID*
1	1
2	2
5	3
6	4
9	6

(a) VersionMap

usrtID	*mark*
C	1
B	1
A	1
B	0
A	0
C	0

(b) EventList

图 3 Timeline Index 结构示意图

Timeline Index 中另一张表是 VersionMap. VersionMap 有两列属性,分别是 *time* 和 *eventID*. *time* 列用来存储输入数据中作为区间开始时刻或者结束时刻的所有的时间点,并且这些时间点按照升序排列. *eventID* 存储着每个时刻之前(包括该时刻)发生过的事件数量,发生的具体事件可以从 EventList 中查找前 *eventID* 行可知. 例如,*time*=5 时共发生过 3 次事件,对应于 EventList 前 3 行.

2.3 问题定义

基于上述概念,下面我们给出本文拟解决的问题的定义:

定义(最优时间窗口覆盖问题):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(Time Interval)的大小表示为 $|\text{Interval}|=t_{\text{end}}-t_{\text{start}}$. 给定持续时间大小 d,我们希望找到满足下面两个条件的时间窗口:

(1) $t_{\text{end}}-t_{\text{start}}\geq d$;

(2) 覆盖该时间区间的用户最多.

3 TLI 算法

基于 Timeline Index 的算法需要两个输入:(1) 所有用户以及对应的时间区间信息;(2) 根据查询要求给定的持续时间大小. 算法最终会输出最优时间区间和覆盖该时间区间的用户的集合. 基于 Timeline Index 的查询算法大致可以分为三步:(1) 构造区间信息表. 在这一步中,我们主要利用 Timeline Index 存储原始数据中的时间和用户信息. 然后,遍历 Timeline Index,利用相邻的时刻构造一系列连续时间区间,并求得对应的覆盖该时间区间的用户集合;(2) 区间调整. 在第一步中,我们以相邻的时刻作为端点构造时间区间. 这导致了其中有部分时间区间会不满足持续时间 *duration* 要求. 针对这些区间,我们进行区间调整,将其与相邻区间进行合并,生成新的满足 *duration* 要求的时间区间,而合并后新的时间区间的用户集合是所有参加合并的时间区间的用户集合的交集;(3) 计算最终解. 在这一步中,我们首先需要对每个时间区间进行求完备解:在保证用户集合不变的前提下,将每个时间区间扩展到最大. 如果不进行此操作,我们无法得到正确的结果. 在求完备解的过程中,我们同时记录用户数量最大的时间区间作为最优解. 得到最优解后,我们再进行解的规范化.

3.1 构建区间信息表

构建区间信息表的主要目的是把原始数据集中在许多不同的时间区间进行拆分,变成由相邻时刻组成的一系列连续时间区间和对应的用户集合. 我们首先利用 VersionMap 存储这些时间区间的端点(开始时刻和结束时刻),这样就可以把所有时间区间转换成许多时刻点进行存储. 另外,为了保存区间对应的用户信息,EventList 记录了每个时刻发生的事件. 最后通过同时扫描 VersionMap 和 EventList 可以得到由相邻时刻组成的一系列连续时间区间和对应的用户集合. 构造区间信息表主要分为两步:(1) 构造 Timeline Index;(2) 求连续区间及用户集合.

3.1.1 构造 Timeline Index

构造 Timeline Index 需要两次遍历数据集. 第一次遍历数据集时构造一张临时表. 临时表的作用是用来记录每个时刻出现的次数. 当得到临时表后,我们可以构造出 VersionMap. 接下来第二次遍历数据集,构造 EventList. 所以,构造 Timeline Index 索引共分为三步:(1) 构造临时表;(2) 构造 VersionMap;(3) 构造 EventList. 接下来介绍三步的具体步骤:

(1) 构造临时表. 临时表记录原始数据中所有时刻点出现的次数. 我们遍历所有数据集,对所有时间区间的开始时间和结束时间进行计数,统计每个时间点出现的次数. 以表 4 为例,构造的临时表如图 4 所示. *time* 表示所有出现过的时刻,而 *count* 是对应时刻出现的次数.

时间	1	2	5	6	9
出现次数	1	1	1	1	2

图 4 临时表示意图

(2) 构造 VersionMap. 临时表中每个时刻出现的次数,实际上也是每个时刻发生的事件数量. 而 VersionMap 中存储每个时刻之前(包括该时刻)发生的事件数量. 所以对于中间表中的每一个时刻,累加该时刻之前的所有时刻(包括该时刻)的 *count* 值即为该时刻的 *eventID* 值. 以 *time*=5 为例,在该

《计算机学报》

时刻之前(包括该时刻)共发生了 3 个事件，所以 VersionMap 中 *time*=5 时的 *eventID*=3. 计算*time*=9 的 *eventID* 则把临时表中所有时间点的 *count* 值相加，得到 *eventID*=6. 最终的 VersionMap 如图 3(a) 所示.

(3) 构造 EventList. 当临时表构造好后，我们可以根据临时表中每个时刻 *count* 值，为不同时刻发生的事件提前划分 EventList 的空间. 例如图 4 中 *time*=1 的 *count*=1，它是最小的时间点，则 EventList 中第 1 行用来存储 *time*=1 发生的事件. 图 4 中*time*=2 的 *count*=1，则 EventList 的第 2 行用来存储 *time*=2 发生的事件. 当 EventList 划分完毕后，进行第二次遍历数据集，填充每个时刻发生的具体事件. 当遇到时间区间的开始时刻时，在 EventList 中为该时间点分配的空间中插入该用户的 *userID*，并标记为"1"；遇到区间的结束时间时，同样在 EventList 中为该时间点分配的空间中插入该用户的 *userID*，但是标记为"0". 例如当遍历到表 4 用户 *A* 的时间区间[5,9]时，时刻 5 作为区间的开始时间，在 EventList 中为时刻 5 划分的空间中插入 *userID*=*A*，*mark*=1 的记录；时刻 9 作为区间的结束时间，所以在 EventList 中为时刻 9 划分的行中插入 *userID*=*A*，*mark*=0 的记录. 最终构造好的 EventList 如图 3(b)所示.

3.1.2 求连续时间区间及用户集合

主要思想. Timeline Index 中 VersionMap 存储了所有的时间点，在 EventList 中存储了所有的事件. 所以可以根据 VersionMap 中相邻的两个时间点来构造时间窗口，而对应的用户集合可以通过遍历 EventList 中具体的事件得到. 最后我们把遍历 Timeline Index 得到的所有时间区间和对应的用户集合存储在一张区间信息表中.

算法步骤. 遍历 Timeline Index 的具体的做法是：对于 VersionMap 中的一行，取该行的 time 作为开始时间，取下一行的 time 作为结束时间，构造成时间区间，并添加到区间信息表中. 如果是 VersionMap 的最后一行，则不做任何处理. 然后根据该行对应的 EventID，扫描 EventList 的前 EventID 行. 在扫描 EventList 的过程中，当遇到激活事件(*mark* 为 1)，把该用户添加到对应时间区间的用户集合中，当遇到失效事件(*mark* 为 0)，把该用户从对应时间区间的用户集合中删去，扫描完成后，把最终的用户集合添加到区间信息表中对应区间的用户集合中.

例 1. 我们以图 3 来阐述遍历的过程. 首先根据 VersionMap 中前 2 个时刻构造区间[1,2]，并扫描 EventList 的前 1 行可得区间[1,2]的用户集合为{*C*}. 然后构造得到区间[2,5]，再扫描前 2 行可得用户集合为{*B*,*C*}. 最终遍历图 3 的 Timeline Index 得到最终的结果如表 5 所示.

表 5 区间信息表

区间	用户集合
[1,2]	{*C*}
[2,5]	{*B*,*C*}
[5,6]	{*A*,*B*,*C*}
[6,9]	{*A*,*C*}

3.2 区间调整

主要思想. 因为由第一步得到的区间信息表中的时间区间是由相邻的时刻组成的，所以其中可能有区间不满足时间窗口大小的要求. 针对那些不满足要求的时间区间，为了使其满足持续时间要求，同时保证该区间的开始时刻不变，我们把它们与其之后相邻的区间进行合并. 合并后的新区间如果仍然不满足持续时间要求，则继续与之后的时间区间进行合并，直到合并后新的区间满足时间窗口大小的要求. 最后所有参与合并的区间的用户集合的交集作为新区间的用户集合. 在这里有一个剪枝的策略：两个区间合并后，合并后的新区间的用户数量必定小于或等于任何一个原来区间的用户数量. 所以可以用 *max* 记录当前最优时间区间的人数，在遍历和合并区间的过程中，一旦区间的用户数量小于 *max*，就终止对该区间的任何操作，从而加快合并过程.

算法步骤. 区间合并时，我们需要遍历中间结果的每个区间，找出那些不满足时间窗口大小的区间，并进行合并，如算法 1 所示. 首先，我们用变量 *max* 记录当前最优时间区间的人数(初始化为 0). 我们用变量 *next* 记录用于合并的区间位置. 对于每个不满足持续时间的区间，将它与之后的第 *next* 个区间进行合并，并更新用户集合(行 5,6)，累加 *next*(行 10). 重复这个操作直到新合并的区间满足持续时间要求. 每次合并后，我们都需判断合并的新区间的用户集合中用户数量是否小于 *max*，如果小于，则直接终止任何操作，判断下一个区间(行 7～9). 如果得到的区间用户数量大于 *max*，则需要及时更新 *max* 的值(行12～14). 这样能够减少很多不必要的区间合并操作，从而达到剪枝的目的.

《计算机学报》

算法 1. 区间调整.

输入:区间信息表,持续时间 *duration*

输出:所有满足持续时间约束的区间

1. 初始化:$max=0$
2. FOR 中间结果中的每个区间
3. 初始化:$next=1$
4. WHILE $|I|<duration$
5. 将区间 I 与接下来第 $next$ 个区间 I_{next} 合并,并更新 I
6. 区间 I 的用户集合←I 的用户集合和 I_{next} 的用户集合的交集
7. IF I 的用户集合中的用户数量$<max$ THEN
8. 直接返回到 FOR 循环对下一个区间进行调整
9. END IF
10. $next$++
11. END WHILE
12. IF I 的用户集合中的用户数量$>max$ THEN
13. 更新 max 值为 I 的用户集合中的用户数量
14. END IF
15. END FOR

例 2. 以表5为例,假设时间窗口大小 $duration=3$. 首先区间[1,2]不满足 duration 要求,所以与之后的[2,5]区间合并成新区间[1,5],而用户集合仍为{C}. 第二个区间[2,5]满足 *duration* 要求,无需进行区间合并. 最终经过区间调整第一阶段后会得到如表6所示的结果. 从中可以得到最优区间为[2,5]和[5,9],因为它们的用户集合都包含两个用户. 但是从图2中可以看到最优区间应该为[2,6]和[5,9]. 所以,只经过区间合并会得到一个不完备的解. 之所以会得到不完备的解,是因为 Timeline Index 只考虑用户区间的端点,而忽略区间中间的时间点. 比如例子中无法得到区间[3,6]的信息,因为原始数据的时间区间端点没有时刻3. 为了解决这个问题,保证解的完备性,需要进行区间调整的第二步.

表 6 经过区间调整后的结果

区间	用户集合
[1,5]	{C}
[2,5]	{B,C}
[5,9]	{A,C}
[6,9]	{A,C}

3.3 计算最终解

在介绍计算最终解的思路和步骤前,我们先介绍"完备解"的概念. 举个例子:假设有三个用户 A、B、C,他们对应的有效时间区间分别为[0,10]、[0,7]和[3,5],时间粒度为1,且要求时间窗口大小 $duration=5$. 图5是该例子的图形表示. 从图5中可以看到最优的时间区间为[0,7]. 根据图5得到的区间信息表如表7所示. 最终通过进行算法之前的步骤得到最优区间为[0,5],它就是不完备解. 虽然区间[0,5]满足所有最优解的要求,但是[0,5]不是完备解(如表8),因为遗漏了区间[1,6]和[2,7]两个最优解结果. 我们希望通过算法求得完备解,即不遗漏任何一个最优解结果.

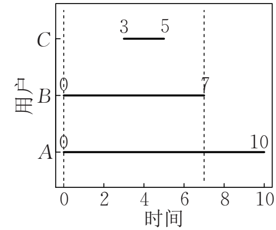


图 5 不完备解示例

表 7 图 5 的区间信息表

区间	用户集合
[0,3]	{A,B}
[3,5]	{A,B,C}
[5,7]	{A,B}
[7,10]	{A}

表 8 图 5 的不完备解

区间	用户集合
[0,5]	{A,B}

3.3.1 求完备解

主要思想. 在区间调整阶段,只要当时间区间满足 *duration* 要求后就停止操作. 但是这会带来一个问题:因为 VersionMap 只存储了所有发生过事件的时刻,在求解连续时间区间时是以 VersionMap 中相邻时刻作为区间端点进行构造的. 这就导致我们会忽略掉一些没有发生事件的时刻所构成的时间区间. 例如图5中,因为时刻1和时刻2没有发生任何事件,导致我们无法得到时间区间[1,6]以及[2,7],所以最终无法得到完备解[0,7].

为了解决这个问题,我们需要对每一个时间区间求完备解. 所谓求一个区间的完备解,就是在保证用户区间不变的前提下,把原来的时间区间尽可能扩大. 只有在 VersionMap 中出现的时刻才发生过事件(用户变得有效或失效),也就是说在那些时刻才可能导致时间区间的用户集合发生改变. 所以,为了求一个区间的完备解,只要保持该区间的开始时间不变,扩大结束时间并判断用户区间是否改变即

《计算机学报》

可. 因为经过区间调整后得到了一系列的连续时间区间，所以扩大结束时间实际上就是将该时间区间与其之后相邻的时间区间进行合并.

算法步骤. 我们用变量 *temp* 来临时存储旧区间与第 *next* 个区间合并后的新区间(行 4)，并判断新区间与旧区间的用户集合是否保持不变(行 5). 如果相同，则更新旧区间，*next* 累加(行 6,7)；如果不同，则直接结束(行 9). 另外我们用 *OptimalTimeWindows* 来存储最优解，如果求得完备解后区间的用户数量目前是最大的，则清空 *OptimalTimeWindows*，把该区间加入到 *OptimalTimeWindows* 中，并更新 *max*(行12～15)；如果区间的用户数量等于 *max*，则直接把该区间加入到 *OptimalTimeWindows* 中. 算法 2 描述如下.

算法 2.　求完备解.

```
输入：根据第一步得到的所有区间表示 R；变量 max
输出：最终最优区间
1. For R 中每个区间 I
2.   初始化：next=1
3.   WHILEDO
4.     将区间 I 与接下来第 next 个区间 I_next 合并，记
       为 temp
5.      IF 区间 I 与区间 temp 的用户集合相同
        THEN
6.        更新 I 为 temp
7.        next++
8.      ELSE
9.        跳出 while 循环
10.     END IF
11.   END WHILE
12.   IF I 的用户集合中的用户数量大于 max THEN
13.     更新 max 值为 I 的用户集合中的用户数量
14.     清空集合 OptimalTimeWindows
15.     把区间 I 加入到集合 OptimalTimeWindows
16.   ELSE IF I 的用户集合中的用户数量等于 max
      THEN
17.       把区间 I 加入到集合 OptimalTimeWindows
        END IF
18.   END IF
19. END FOR
```

例 3.　例如上例中经过区间调整第一阶段得到区间[2,5]和它的用户集合{B,C}后，继续判断区间[2,5]和[5,6]合并后的用户集合是否保持不变. 区间[2,5]和[5,6]合并后的区间[2,6]的用户集合仍为{B,C}，与未合并时区间[2,5]的用户集合不同，所以将旧区间[2,5]更新为[2,6]. 再继续与[6,9]合并后的用户集合为{C}，减少了用户 B，所以舍弃本次合并. 最终区间[2,5]的完备解为[2,6]，对应的用户集合为{B,C}. 我们对其他区间进行同样的操作，最后可以得到表 9 所示的最优的完备解.

表 9　步骤 2 后的最优结果

区间	用户集合
[1,9]	{C}
[2,6]	{B,C}
[5,9]	{A,C}
[6,9]	{A,C}

3.3.2　解的规范化

解的规范化操作就是将拥有相同用户集合且相交的区间进行合并. 例如表 9 中区间[5,9]与区间[6,9]相交且用户集合相同，所以将两者合并成[5,9]，用户集合仍为{A,C}.

3.4　算法复杂度

假设数据集的大小为 N，共有 M 个区间. 其中 $1 \le M \le N$. 假设 VersionMap 中共有 K 个时间点，则 $2 \le K \le 2M$，因为每个区间最多贡献 2 个时间点. 基于 Timeline Index 的查询算法首先为了构造 VersionMap 需要遍历一次数据集，消耗 $O(N)$的时间. 遍历一次数据集后，构造临时表，并对临时表进行排序，消耗 $O(K\log K)$的时间，接着通过遍历临时表构造 VersionMap 的时间复杂度为 $O(N)$的时间. 算法构造 EventList 需要第二次遍历原始数据集，时间复杂度为 $O(N)$. 算法遍历 Timeline Index 得到中间结果时，需要遍历 Timeline Index 的每一个时间点，时间复杂度为 $O(K)$. 所以基于 Timeline Index 的查询算法的时间复杂度 $O(N\log N)$.

4　实验评估

为了保证用户体验，服务需要能够及时得到响应，即尽可能快地提供服务，所以实验主要考察算法的运行效率. 根据 TLI 算法伪代码可知算法的输入是原始数据集和持续时间 *duration*，所以 TLI 算法的主要因素应该是数据量和持续时间 *duration*. 我们采取控制变量法的实验方法，研究数据量和持续时间 *duration* 对算法性能的影响. 首先，我们把改进的滑动时间窗口算法(STW 算法)作为基准算法，通过比较 STW 算法和 TLI 算法的整体运行时间来验证 TLI 算法的效率. 其次，为了更加详细地分析不同因素对基于 Timeline Index 的查询算法的性能影响，我们细化考察不同因素对该算法三个阶段的

《计算机学报》

运行效率的影响. 最后我们对基于 Timeline Index 的算法区间调整阶段中三个步骤的性能进行了详细地分析. 所有实验都是在算法的结果正确且一致的前提下进行的.

本文实验数据集来自于某阅读软件一个月内所有用户的阅读行为,经过对原始数据的清洗和预处理后,最终实验数据共包含 67549 个用户和 387868 条记录,其中每条记录的内容是一个用户对应着一个时间区间,代表的含义是该用户在该时间范围内一直在进行阅读,且时间粒度为 s. 我们基于该实验数据,再进行适当地调整,获得不同时间粒度的实验数据. 另外,我们人工模拟了 40 万条记录,其中包括 8000 名用户,每个用户有 50 条时间区间记录.

4.1 数据预处理

本实验数据集来自于某手机阅读软件一个月的用户行为日志. 该行为日志中包括用户多种行为,例如:翻页、点击封面、加入收藏夹等,并且每一个行为都有对应的时间戳,时间粒度为 s. 我们首先从用户行为日志中筛选出用户的翻页行为,再对筛选出的数据进行进一步处理.

我们需要根据得到的用户翻页行为日志来分析用户每次连续阅读的时间段,具体做法是:首先统计用户相邻两次翻页行为的时间差,作为用户阅读一页书籍的时间. 经过统计发现,有 83% 的用户阅读一页书籍的时间在 5 min 以内. 所以规定:若用户某两次翻页的时间差在 5 min 以上,则用户进行了两次阅读,即用户中途中断了阅读,后来又重新阅读书籍. 根据这一规定,把相邻两次翻页时间在 5 min 以内的记为一次连续的阅读行为,把相邻两次翻页时间在 5 min 以上的翻页行为记为两次阅读行为. 最终,我们可以得到每个用户的多个阅读时间段,每个阅读时间段都代表用户进行一次连续阅读的时间,且时间粒度为 s.

另外,为了增大数据量,我们模仿真实阅读数据的格式,模拟了 40 万条记录. 其中包括 8000 名用户,每个用户有 50 条时间区间记录.

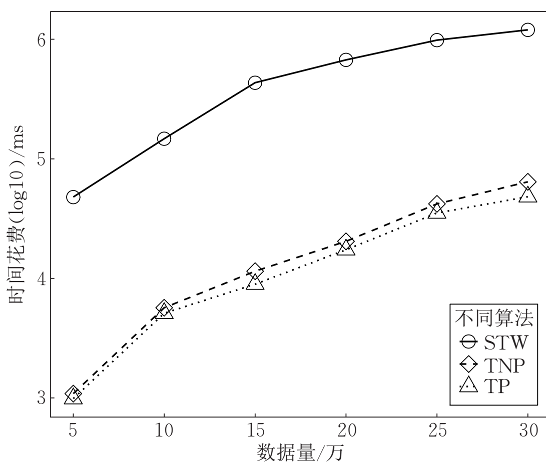
4.2 算法运行时间

在这一小节,我们把 STW 算法作为基准算法,比较 TLI 算法和 STW 算法的总体运行时间. 另外,为了比较 TLI 算法中剪枝策略的效果,我们还把 TLI 算法分为:(1) 有剪枝策略的 TLI 算法(简称 TP);(2) 没有剪枝策略的 TLI 算法(简称 TNP). 我们通过比较 STW、TP、TNP 执行一次查询的运行时间,来分析 TLI 算法的运行效率以及 TLI 算法中剪枝策略的效果.

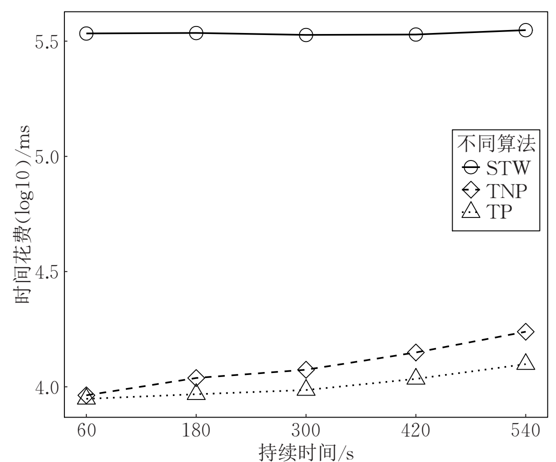
图 6 展示了三种算法在不同因素改变下执行一次查询的运行时间图,运行时间单位为 ms. 由于算法运行时间的跨度比较大,我们对运行时间取了常用对数值(log). 从图 6 两图中可以看到 TLI 算法的运行效率比 STW 算法最多快了 1.5 个数量级,而且 TP 算法的运行效率略优于 TNP 算法.

图 6(a) 展示了数据量的改变对算法查询时间的影响. 数据量从 5 万变化到 30 万,同时保持持续时间为 300 s 不变. 我们可以看到随着数据量的增加,三种算法的查询时间都增加了.

图 6(b) 展示了持续时间的改变对算法查询时间的影响,其中数据量保持 15 万条记录不变,持续时间从 60 s 变化到 540 s. 我们发现,随着持续时间增大,STW 算法的查询时间几乎不改变,而 TP 算法和 TNP 算法的查询时间都增加了,这是因为随



(a) 数据量



(b) 持续时间

图 6 算法执行一次查询的运行时间分析图

《计算机学报》

着持续时间增加，算法区间调整阶段需要进行更多的区间合并操作来得到满足持续时间要求的时间窗口. 而且 TNP 算法的增加幅度略大于 TP 算法，这是因为剪枝策略有效地减少了 TP 算法在区间调整阶段的不必要的区间合并操作，从而优化运行效率.

4.3　TLI 算法三个阶段时间消耗

为了深层次地研究 TLI 算法的各阶段性能，这一小节我们统计不同因素下 TLI 算法三个阶段的运行时间情况，分析改变不同因素对这三个阶段的影响，从而比较不同因素改变时对 Timeline Index 算法查询效率的影响. 通过横向比较图 7 中 TP 算法和 TNP 算法发现，TP 算法和 TNP 算法在阶段一上的运行效率几乎没有差异，在第二、三阶段上 TP 算法优于 TNP 算法. 尤其是在第三阶段，TP 算法的时间消耗几乎可以忽略不计. 这是因为剪枝策略应用于算法第二阶段，减少区间调整的操作次数，从而也减少求完备解的区间数量，最终减少阶段三的时间花费.

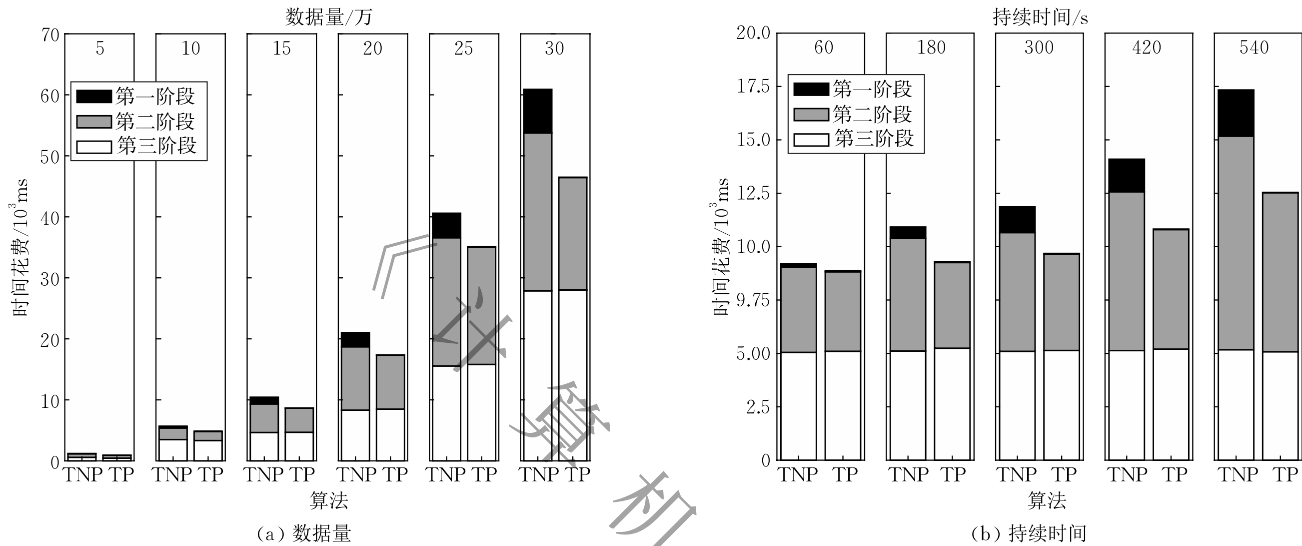


图 7　算法各阶段运行时间分析图

图 7(a)展示了数据量的改变对 TLI 算法三个阶段运行效率的影响. 数据量从 5 万变化到 30 万，同时保持持续时间为 300 s 不变. 如图 7(a)所示，数据量越大，算法的查询时间越长，并且三个阶段的时间都会相应增加. 但是 TNP 算法在第三阶段的运行时间有明显的增加，而 TP 算法在第三阶段的运行时间的增加微乎其微.

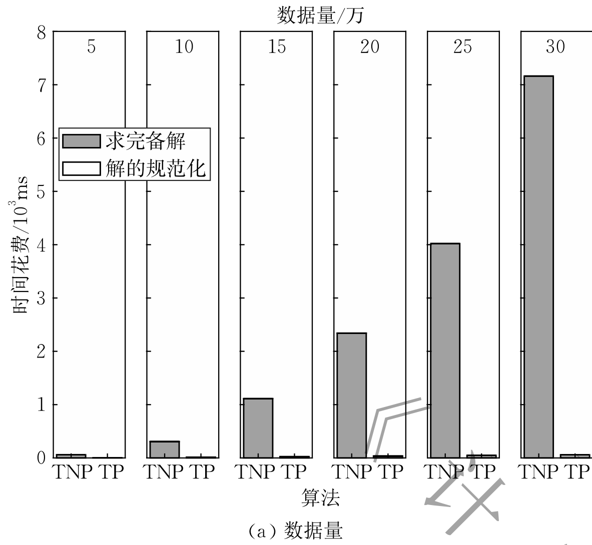
图 7(b)展示了持续时间的改变对 TLI 算法三个阶段运行效率的影响，其中数据量保持 15 万条记录不变，持续时间从 60 s 变化到 540 s. 如图 7(b)所示，随着持续时间增多，算法的查询时间大致呈增长趋势. 但是其中阶段一的运行时间几乎没有改变，而区间调整和计算最终解的运行时间有所改变. 在 TNP 算法中，随着持续时间增大，区间调整和计算最终解的时间花费大致呈增长的趋势. 这是因为持续时间的增大，对算法的第一阶段没有影响，而区间调整阶段需要合并更多的时间区间才能满足持续时间要求. 在 TP 算法中，由于剪枝策略的存在，区间调整的增加幅度并没有 TNP 算法明显，而计算最终解的时间花费反而减少了，这将在 4.4 节具体分析.

4.4　TLI 算法计算最终解阶段两个步骤时间消耗

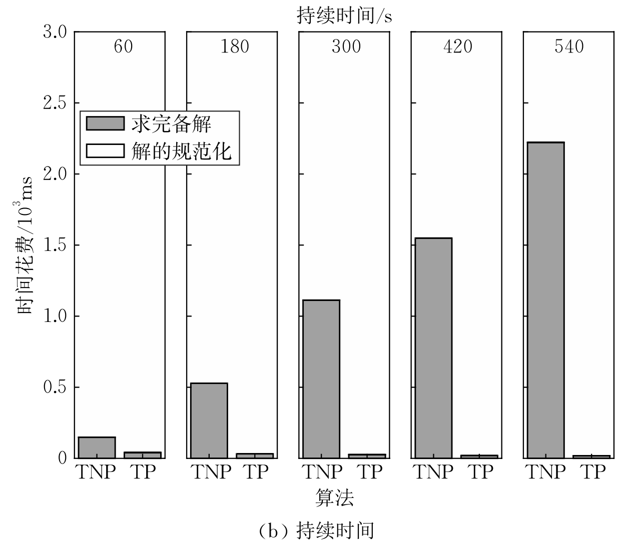
因为计算最终解包含了两个步骤，为了进一步研究剪枝策略的影响，在这一小节中我们比较不同因素下计算最终解中两个步骤的时间变化，从而分析不同因素的改变对计算最终解的影响以及剪枝策略对计算最终解的具体影响. 计算最终解共有两个步骤，分别是：(1) 求完备解；(2) 解的规范化. 另外我们统计这两个步骤的操作次数，统计结果如图 9 所示. 对两个步骤的操作次数统计，为分析两个步骤的时间花费提供了有效的凭证. 如图 9 所示，解的规范化的操作次数几乎可以忽略(据统计，只有 1,2 次)，这导致了规范化的运行时间几乎可以忽略.

图 8(a)展示了数据量的改变对 TLI 算法第三阶段两个步骤运行效率的影响. 如图 8(a)所示，数据量越大，TNP 算法的求完备解的时间有明显的增加，而 TP 算法由于剪枝策略的存在，求完备解的时间明显少于 TNP 算法，且增加的幅度也没有 TNP 算法明显. 这从图 9(a)的操作次数统计中也可以得到证实. 图 8(b)展示了持续时间的改变对 TLI 算法第三阶段两个步骤运行效率的影响. 如图 8(b)所示，随着持续时间大小增加，TNP 算法求完备解的

时间随持续时间大小增加而增加. 这是因为本文在进行持续时间实验时选取的数据集中所有记录的时间区间大小 540 s. 当持续时间较小时,经过阶段二区间调整得到的时间区间会进行较少次数的区间合并,直到达到某一个时间区间的结束时间点为止. 而当持续时间较大时,会经过更多次的区间合并才能到达一个时间区间的结束点. 而 TP 算法中求完备解的时间反而减少. 这是由于剪枝策略使区间调整后的区间数量减少,从而减少了步骤一的操作次数(可从图 9(b)中可知).

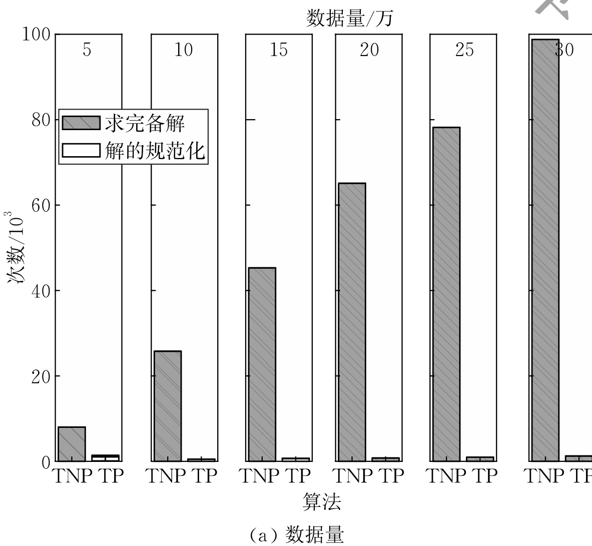


(a) 数据量

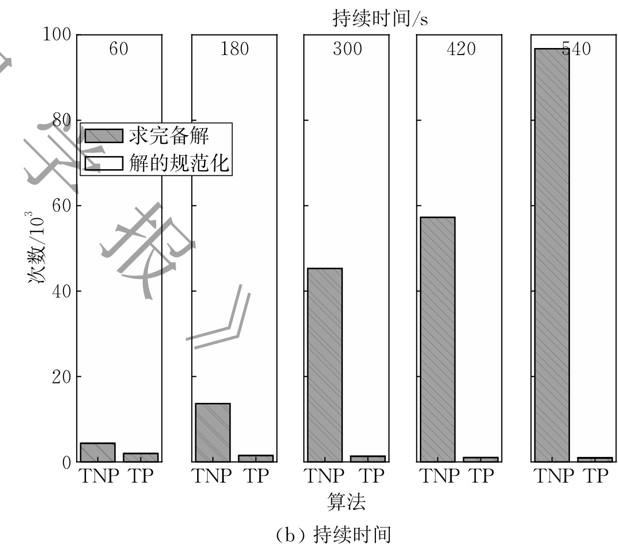


(b) 持续时间

图 8 第三阶段各步骤运行时间分析图



(a) 数据量



(b) 持续时间

图 9 第三阶段各步骤操作次数统计图

《计算机学报》

4.5 实验总结

通过实验分析,我们发现 TLI 算法在运行效率上比我们的基准算法快了最多 1.5 个数量级. 另外,我们发现数据量与持续时间都会对 TLI 算法的查询效率产生影响. 总体来说,随着数据量或持续时间的增加,算法的查询时间都有所增加. 但是细化到算法的每个阶段时:当数据量增加时,TLI 算法各个阶段的运行时间都有所增加;而当持续时间增大时,TLI 算法第一阶段没有影响,第二阶段运行时间增加,第三阶段的运行时间反而减少. 另外,通过比较 TLI 算法在有无剪枝策略下的运行效率,我们发现剪枝策略在算法的第二、第三阶段都能明显提高运行效率.

5 相关工作

与本文所提的最优时间窗口覆盖查询相关的研究主要是时态数据库中的聚合查询优化算法,主要分为精确性和近似性查询算法两大类.

时态数据库从发展至今已经有成熟的数据库查

询语言被提出，如 TQuel[3] 和 TSQL2[4]，它们都包含了时态聚合操作. 同样也有各种各样关于时态聚合的算法被提出[5]. Kline 和 Snodgrass 发明了一种叫作聚合树[6-7]的索引结构. 聚合树支持时态聚合的增量计算. 但是聚合树存在一个严重的问题，即聚合树是不平衡的. 在最坏的情况下，根据 n 个排好序的元组构造的聚合树近似是一个链表，所以在最坏情况下，构造聚合树的复杂度将是 $O(n^2)$，处理一次时间聚合查询的时间复杂度为 $O(n)$，其中 n 为数据量大小.

Moon 等人[8]提出了两种复杂度是 $O(N\log N)$ 的算法，分别是计算 COUNT 的平衡树算法[9] 和计算 MAX 的归并排序聚合算法. 平衡树算法通过动态红黑树的插入来构造平衡树. 在每个结点中，都存储着一个时间戳和两个计数器. 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. 当创建好平衡树后，对其进行中序遍历，累加各结点的开始计数器的值，累减结束计数器的值，最终可以得到相邻结点的时间戳构成的区间的聚合值. 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. 因为平衡树结点中的计数器只存储各个聚合值的累积信息，并不记录特定时间戳时所有元组的信息，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 等人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. 之后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[10].

Yang 等人提出了一种能够支持增量计算和维护的索引结构 SB-tree[11-12]. SB-tree 是基于 B-tree[13-15] 和 segment-tree[16-17] 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 等人[18] 结合了多维 B-tree 的特征，提出了 MVSB-tree 的索引结构，能够处理任意范围的时间 SUM，COUNT 和 AVG 聚合查询.

Kaufmann 等人[1] 提出了一种集成在 SAP HANA 数据库中的统一数据结构 Timeline Index 以支持时态聚合查询，其中 N 为数据量大小. Timeline Index 主要利用 VersionMap 和 EventList 来存储每个版本发生的事件，通过同时遍历两个数据结构来进行查询.

但是以上的这些方法大致都至少存在以下一个缺点：(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 等人[19-21] 针对那些可以接受小范围误差的情况，基于 B-tree 和 R-tree，提出一种能够近似计算时间 SUM 和 COUNT 聚合的方法[22].

以上方法都是用于解决时态聚合查询的，而不适用于解决最优时间窗口覆盖查询问题. 两个问题主要存在两个区别：(1) 查询区间要求不同：时态聚合查询的区间是由查询者给出的一个固定的区间，而最优时间窗口覆盖问题中的区间只有大小要求，而没有固定的位置；(2) 计数对象不同. 时态聚合查询中 COUNT 聚合函数的计数对象只需要存在某一区间与查询区间相交即可. 而最优时间窗口覆盖问题中要求用户的某一区间需要完全覆盖查询区间才会被计数. 计数对象的不同导致了传统时态聚合的索引结构无法直接用于解决该问题.

另外，我们还对最长公共子区间的问题进行了调研. 最长公共子区间问题是指：给定若干个整数区间，要求得这些整数区间两两相交的最长子区间的长度. 该问题是线段相交问题的特殊情况. McKenney 等人[23] 提出了一种用 sweep-line 算法，该算法需要 $O(n\log n+k\log n)$ 的时间以及 $O(n+k)$ 的空间，其中 n 为线段数量，k 为最终得到的相交线段对的数量. 之后，Mehlhorn[24] 成功地将该算法的空间复杂度降低到 $O(n)$. Chazelle 和 Edelsbrunner[25] 提出了一种更加高效的算法，该算法的时间复杂度为 $O(n\log n+k)$，同时在最坏情况下的空间复杂度为 $O(n+k)$. 但是本文提出的最优时间窗口覆盖问题要求找出最多区间相交的公共子区间，而不是仅仅只是两个区间相交即可. 另外，在该问题中，还有持

计算机学报

续时间的要求,对于最终找到的子区间必须满足持续时间要求大小.

6 总结和未来工作

本篇文章提出最优时间窗口覆盖问题,该问题的解决能够为时间驱动的服务提供技术支持.本文基于 SAP HANA 中的 Timeline Index 数据结构,设计了高效的查询算法.针对算法中时间区间调整效率低的问题,本文还进一步提出了减少区间调整次数的剪枝策略,使得整体算法的执行效率更加高效.基于大规模真实数据集验证,本文所提 TLI 算法普遍比基准滑动时间窗口扫描算法快了两个数量级,在最好情况下下能达到快三个数量级的效果,能够满足最优时间窗口覆盖查询问题的大部分应用场景.

在未来的工作中,我们将考虑进一步对算法进行优化,如扩展算法以支持计算不同权重的用户;此外,当面对在线实时更新的时序数据增加/删除等操作时,设计增量的索引方法以解决动态更新的操作问题.

《计算机学报》

参 考 文 献

[1] Kaufmann M, Manjili A A, Vagenas P, et al. Timeline index: A unified data structure for processing queries on temporal data in SAP HANA//Proceedings of the ACM SIGMOD International Conference on Management of Data. New York, USA, 2013: 1173-1184

[2] Sikka V, Färber F, Goel A, et al. SAP HANA: The evolution from a modern main-memory data platform to an enterprise application platform. Proceedings of the VLDB Endowment, 2013, 6(11): 1184-1185

[3] Snodgrass R. The temporal query language TQuel. ACM Transactions on Database Systems, 1987, 12(2): 247-298

[4] Snodgrass R T. The TSQL2 Temporal Query Language. Berlin, Germany: Springer, 2012

[5] Liu L, Özsu M T. Encyclopedia of Database Systems. Berlin, Germany: Springer, 2009

[6] Kline N, Snodgrass R T. Computing temporal aggregates//Proceedings of the International Conference on Data Engineering. Taiwan, China, 1995: 222-231

[7] Kuo T W, Lin C J, Tsai M J. On the construction of data aggregation tree with minimum energy cost in wireless sensor networks: NP-completeness and approximation algorithms//Proceedings of the IEEE International Conference on Computer Communications. Toronto, Canada, 2014: 2591-2595

[8] Moon B, López I F V, Immanuel V. Scalable algorithms for large temporal aggregation//Proceedings of the International Conference on Data Engineering. California, USA, 2000: 145-154

[9] Wu H, Luo B, Sun Z. Balanced tree-based support vector machine for friendly analysis on mobile network//Proceedings of the International Conference on Intelligent Computation. Lanzhou, China, 2016: 183-191

[10] Moon B, Lopez I F V, Immanuel V. Efficient algorithms for large-scale temporal aggregation. IEEE Transactions on Knowledge and Data Engineering, 2003, 15(3): 744-759

[11] Yang J, Widom J. Incremental computation and maintenance of temporal aggregates//Proceedings of the International Conference on Data Engineering. Heidelberg, Germany, 2001: 51-60

[12] Leis V, Kundhikanjana K, Kemper A, et al. Efficient processing of window functions in analytical SQL queries. Proceedings of the VLDB Endowment, 2015, 8(10): 1058-1069

[13] Rodeh O, Bacik J, Mason C. BTRFS: The Linux B-tree filesystem. ACM Transactions on Storage, 2013, 9(3): 9

[14] Levandoski J J, Lomet D B, Sengupta S. The Bw-tree: A B-tree for new hardware platforms//Proceedings of the International Conference on Data Engineering. Brisbane, Australia. 2013: 302-313

[15] Ferragina P, Venturini R. Compressed cache-oblivious string B-tree. ACM Transactions on Algorithms, 2016, 12(4): 52

[16] Mei X, Sun X, Dong W, et al. Segment-tree based cost aggregation for stereo matching//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. Portland, USA, 2013: 313-320

[17] Wu W, Li L, Jin W. Disparity refinement based on segment-tree and fast weighted median filter//Proceedings of the IEEE International Conference on Image Processing. Phoenix, USA, 2016: 3449-3453

[18] Zhang D, Markowetz A, Tsotras V, et al. Efficient computation of temporal aggregates with range predicates//Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. California, USA, 2001: 237-245

[19] Yang Z, Zheng L, Li M, et al. Matching algorithm for plant protecting unmanned aerial vehicles and plant protecting jobs based on R-tree spatial index. Transactions of the Chinese Society of Agricultural Engineering, 2017, 33(1): 92-98

[20] Beckmann N, Kriegel H P, Schneider R, et al. The R*-tree: An efficient and robust access method for points and rectangles. ACM SIGMOD Record, 1990, 19(2): 322-331

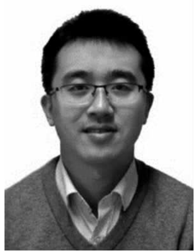
[21] Wang S, Maier D, Ooi B C. Fast and adaptive indexing of multi-dimensional observational data. Proceedings of the VLDB Endowment, 2016, 9(14): 1683-1694

[22] Tao Y, Papadias D, Faloutsos C. Approximate temporal aggregation//Proceedings of the International Conference on Data Engineering. Boston, USA, 2004: 190-201

[23] McKenney M, Frye R, Dellamano M, et al. Multi-core parallelism for plane sweep algorithms as a foundation for GIS operations. GeoInformatica, 2017, 21(1): 151-174

[24] Mehlhorn K. Data Structures and Algorithms 1: Sorting and Searching. Berlin, Germany: Springer, 2013

[25] Chazelle B, Edelsbrunner H. An optimal algorithm for intersecting line segments in the plane. Journal of the ACM, 1992, 39(1): 1-54



CAO Bin, born in 1985, Ph. D., associate professor. His research interests include management of spatio-temporal data.

HOU Chen-Yu, born in 1994, M. S. candidate. His research interests include service computation and big data.

FAN Jing, born in 1969, Ph. D., professor. Her research interests include service computation and virtual reality.

CHEN Shi-Wei, born in 1981, Ph. D., associate professor. His research interests include human-computer interaction.

QIU Jie-Fan, born in 1984, Ph. D., associate professor. His research interests include IoT and sensor networks.

Background

This paper focuses on a new problem named the optimal time window covering problem, which is related to the temporal aggregation in database. In database management, aggregation denotes the process of summarizing a database instance. In temporal relational aggregation, main concerns are temporal relations, and the tuples can be grouped by their timestamp values. Then an aggregation function is used to obtain results.

Since 1980s, there were intensive research focused on temporal databases, and temporal aggregates had been incorporated in temporal query languages, e. g., TSQL and TQuel.

Tuma is the first one who concerned about the efficient processing of temporal aggregates. Following Tuma's pioneering work, many research aimed at the development of efficient in-memory algorithms for temporal aggregates were put forward to. There are some classical data structure were designed to support temporal aggregations. For example, Kline and Snodgrass proposed a data structure named aggregation tree, and Moon B. proposed balanced tree.

Furthermore, with the flush of OLAP and data warehouse, disk-based index structures for the incremental computation and maintenance of temporal aggregates attracted many attentions. Yang invented a data structure named SB-Tree and it was extended by Zhang et al. to include non-temporal range predicates. Then Tao et al. proposed an approximate solution for temporal aggregation.

However, in our problem, there is no given interval to query. Besides, we count the users who have an interval covering query intervals. These differences result in that there is no current methods can be directly adopted to solve our problem. Therefore, we proposed two methods in this paper. The first one is a baseline, named Sliding Time Window algorithm. And the second one is an improving method, named Timeline Index based algorithm.

This paper proposed an interesting problem which will attract many attention in database. And this paper puts forward two solutions for reference.

This research was supported by Grants from National Key Research & Development Program of China (Grant No. 2016YFB1001403), the National Natural Science Foundation of China (Grant Nos. 61602411, 61572437, 61772468 and 61502427), the Key Research and Development Project of Zhejiang Province (Grant Nos. 2015C01029, 2015C01034), the Natural Science Foundation of Zhejiang Province (Grant No. LY16F020034), and the Major Science and Technology Innovation Project of Hangzhou (Grant No. 20152011A03).

《计算机学报》