基于多线程的时态数据查询研究与实现

专 业：计算机科学与技术
硕 士 生：池雪辉
指导老师：叶小平 教授

摘 要

随着互联网的发展，数据与时间的联系越来越紧密。传统的时态数据库难以满足时间标签的需求，并且目前还无完整的时态数据库。因此研究时态数据库越来越迫切，时态数据索引便成为了时态数据库中的热点。随着大数据和并行计算的发展，时态数据索引在多线程上的实现也至关重要。

论文研究了一种基于“拟序”关系的时态数据索引框架，是一种处理“数据本体”和“时间标签”的索引技术，且在时态数据查询时实现了多线程，并将其应用在了时态数据平台中。首先通过“拟序”关系，在时态数据集上建立线序划分，并建立了相应的索引树结构。其次，在时态数据中根据建立的索引树结构研究和设计了包含与被包含查询的算法。在这之后在磁盘中实现了TDIndex 索引。接着针对索引树底层的耗时查询，依据线序划分的性质实现了多线程查询，并提出了前部交叉划分策略等四种索引数据划分方法。然后开发了基于现有关系数据库的时态数据平台 TempMT_Index，实现了时态选择，时态投影，时态连接等操作，并在时态选择中成功地运用了时态数据索引与多线程的实现。最后，论文通过实验仿真评估和仿真系统，表明应用的可行性、有效性和优越性。该时态数据索引具有理论支撑，适用于现有的关系数据库系统，具有研究意义。该时态数据索引通过策略划分与多线程的实现，极大地提高了查询效率，适应于多核 CPU 等并行计算环境，且适应于大数据环境下，具有重要的现实意义。

关键词：线序划分，时态数据，时态数据索引，多线程，前部交叉划分策略

RESEARCH AND REALIZATION OF TEMPORAL DATA QUERY BASED ON MULTI-THREAD

Major : Computer science and technology
Name : Chi Xuehui
Supervisor : Prof. Ye Xiaoping

ABSTRACT

With the development of the Internet, data and time are more and more closely linked. It's difficult for the traditional temporal database to meet the need of era, and the temporal database is not yet complete up till now. Therefore, the study of temporal database is increasingly needed, and thus the data index becomes a hot issue. As big data and parallel computing develop, the integration of temporal data index and multi-thread becomes quite essential.

This thesis studies a temporal data index framework which is based on "coherent" relation. It is a kind of indexing technique to deal with "data ontology" and "time tag" which realizes multi-thread on temporal data query and applies it in temporal data platform. Firstly, the linear order is established on the temporal data set through "coherent" relation, and the corresponding index tree structure is set up. Secondly, in the temporal data, the algorithm of contains query and during query is studied and designed according to the index tree structure. After that,realize the TDindex with split file in disk. Then, according to the time-consuming query of the bottom of the index tree, the multi-thread query is realized according to the nature of the linear order partition, and the index data division method of the front cross partition strategy is carried out and so on. Next, the temporal database system based on the current relational data platform, TempMT_Index, is developed. It realizes the tense selection,

temporal projection and temporal connection. And the temporal data index and the multi-thread are successfully applied in the tense selection. Finally, the thesis demonstrates the feasibility, effectiveness and superiority of the application through the experimental simulation evaluation and simulation system. The temporal data index has theoretical support and it is applicable to the current relational database system, which is of great research value. The temporal database index greatly improved by the combination of strategy partition and multi-threading, which has brought about a vast improvement in the efficiency of query. It adapts to the environments of parallel computing of multi-core CPU and the big data, which has great practical significance.

KEY WORDS：linear order partition，temporal data，temporal data index，multithreading，front crossing partition strategy

目 录

图目录

表目录

第1章 引言

本文研究了时态数据索引，根据线序划分（[1]）的特点采用多线程实现了时态索引的检索。并建立了基于传统数据库管理的时态数据平台（[2]），实现了多种时态操作和多线程检索。

1.1 研究背景

时间作为客观事物的基本属性，反映事物的发展变化。随着互联网的发展，新型数据与时间的联系越来越紧密，数据的时态问题经常需要被处理。时态数据由数据本体和时间标签共同构成，所以时态数据的管理可视为常规数据管理的拓展。首先，常规数据库（[3]）默认存储数据的最新状态，实际为快照数据，数据的时间属性不是显示的属性。因此常规数据库仅能管理某一时刻的数据。当数据发生变化，新数据将覆盖相应历史数据，从而丢失历史数据记录，虽然在传统数据库中也容易存储历史数据，但这样打破了数据记录的各项完整性，使得数据的管理变得即为复杂。所以有必要对时间标签单独进行管理。其次，实现数据管理首先需要实现对数据的查询，由于时间具单调性（即时间不断向前推移）、多维性（有效时间、事务时间和用户自定义时间等维度）和相互关系的复杂性（ALLEN 时间关系（[4]）），传统的关系数据框架难以高效查询时态数据。因此，需要有新的时态数据的管理框架，这成为了数据库领域的研究热点。

由于传统的数据库难以高效查询时态数据，需要建立新的时态数据管理框架，则在该框架中时态数据索引便成为时态数据管理领域的研究热点。传统的数据库中增删查改，已经十分完备，而时态数据的时间检索尤其重要。时态索引以时态数据模型，其核心为“数据本体”与“时态信息”的整合，具体的整合方法主要分为三种：①数据本体和时态信息的单独处理；②时态信息转化为非时态信息的处理；③数据本体与时态信息的协同处理。其各有特点。本文的时态索引是从协同处理为出发点构建的。

而在过去的几十年里，CPU 的发展得十分迅速，一直按照摩尔定律进行

着，提高 CPU 主频成为其最重要的手段。在 CPU 的工艺已经到了 14-32 纳米，CPU 的主频提高带来了散热、漏电、单核心功耗过高等，以及 CPU 速度与存储器速度匹配不上等问题，成为 CPU 主频提高难以解决的问题。CPU 纵向发展的思路逐渐转向横向发展，多核 CPU 大量出现，加之现代计算机的发展由串行计算时代过渡到了并行计算时代，单点计算机的性能要求也更高。随着多核 CPU 平台的到来，计算机虽然在性能上发生了明显的变化。但还是存在着大量的传统单线程的程序时，多核 CPU 的性能过剩，有如 CPU 厂商 Intel 生产大多数的 CPU 都只采用四核心的 CPU，CPU 核还有大幅度增加的空间，制约其发展的原因在于多线程技术的应用还远远不够。所以应用于在多核 CPU 平台时则有必要重新考虑的问题，特别是多线程（[5]）应用的问题。

在论文分布式时态索引技术（[6]）中给本文提供了并行索引的思路，不同的是其是基于多台计算机而言，本文则是基于单台计算机而言。本文中的时态数据平台将大量运用于服务器中，而服务器中往往有大部分都是多个或者多核 CPU，而传统的时态数据库索引都是采用单线程技术实现。时态数据集上进行线序划分的，由于线序有独立的特点，其能很好的适应于多线程技术，采用多线程技术对索引进行优化，能够更好的提升查询效率。

1.2 国内外研究现状

纵观数据库的发展历史，关于时态数据的管理和一般数据库技术的发展是相辅相成的。相较于常规的关系数据库，时态数据库已经在理论上证明其在处理时态数据方面的优越性，然而目前仍未有类似数据库管理系统的完整的时态数据库产品（[7]），只能通过索引方式高效管理时态数据，时态索引技术就管理时态数据管理的有效手段。常规数据库一般利用成熟的 B+-tree（如 MAP21 索引（[8]））和 R-tree（[9]）（如双时态索引 4R-tree（[10]）和 GR-tree（[11]））技术来处理关系数据的时态信息，实际上就是利用索引技术来处理时态数据。而进入二十一世纪，主流数据管理除了关系数据管理之外，得益于计算机网络技术的快速发展，出现了许多新型的应用，同时也带来新型数据的管理需求，时态数据库系统成为数据库研究热点。关于新型数据的管理，包括管理数据的生命周期、状态特征和增量式更新等（例如 XML 等半结构化数据），以及数据

本身需直接处理时态信息，因而对于数据时间维课题的研究使用变得更为迫切，如何借助时态索引对网络环境中数据时态信息进行准确描述和有效处理成为时态数据管理研究范畴的热点研究课题。

目前时态数据库（Temporal Database）还没有像如 Oracle、MySQL、DB2 等大型关系数据库那样的产品。在当前时态数据库技术尚未完全成熟的现状下，DBMS 提供商不会轻易把时态处理功能引入现有的 DBMS 中，因此构建时态数据库的中间件（[12][13][14]）是一个很必要的过程。因此有了 TimeDB（[15]）和 TempDB（[16]，让时态数据库系统能够应用起来。

时态索引以时态数据模型为理论基础，其核心为“数据本体”与“时态信息”的整合。时态信息一般指数据的时间标签，而“数据本体”指数据的自身特征，如关系数据的表结构等。论文通过查阅材料，将与“时态信息”与“数据本体”处理的相关工作分为下列三种情形：①数据本体和时态信息独立处理：其本质为基于成熟的代数操作建立一个对时态信息进行管理的索引框架，先通过时态处理筛选数据，再对筛选后得到的数据进行常规处理，例如时态关系数据库将时态信息处理归类为时态关系投影、时态关系选择和时态关系连接等，而时态连接代数将时态对象数据库将时态信息处理归类为时态继承关联和时态引用关联。②将“时间”归结为非时态数据处理：例如经典时空数据库把时间作为“新”的空间维，将一维时间与二维空间的时空数据直接作为“纯三维”空间立方体处理，通常使用 B+-tree、R-tree 和 R*-tree 等来对数据进行有效的存储、查询和更新，例如 Tao 等人提出的 MV3R-tree（[17]）和 TPR*-Tree（[18]）、Chakka 等人提出的 SETI 索引、Abdelguerfi等人提出的 2-3 TR-tree（[19]）、Procopiuc 等提出的 STAR-Tree 等。③时态与非时态数据协同处理：其本质思想为针对数据本身特征，整合时态查询与非时态查询，例如时态处理基于 B+-tree 的 TB-tree（[20]）和 SEB-tree（[21]），时态处理基于 R-tree 的 RT-tree（[22]）、HR-tree（[23]）和 HR+-tree（[24]），将时态数据与结构协同处理的 TempIndex（[25]）和 TempSumIndex（[26]）等。

“①”中的研究工作，关键是将已有的“关系代数”和“对象关联代数”在时态层面进行扩充，以进行时态代数处理；“②”中的相关工作，核心为利用成熟的 B-tree 及 R-tree 相关技术高效地处理时态数据相应操作；伴随网络技术

的快速发展，相关的数据本体复杂，既无可作为支撑的“代数”操作，也无可借鉴的成熟技术。伴随计算机网络技术的快速发展，新型时态数据的管理需求随之出现，“③”中相关的时态数据处理方式已成为人们关注的科研课题。相关课题需研究两个基本问题，一是建立能表现时间特征的一般时态索引框架，二是整合“数据本体”与“时态信息”，建立相应协同处理机制。

1.3 研究意义

时态数据库应用场景十分广泛，而对数据查询的需求愈加愈多。针对大数据进行数据查询的需求越来越多，虽然带有多个或者多核高性能 CPU 的服务器存在，传统的单线程时态数据的查询已经远远不能适应现有的情况。在现有情况下时态数据索引提高对时态数据的查询性能，但其始终是运用于单线程的架构下。在时态数据量增大的前提下，实际应用中服务器 CPU 的核数与个数不断增大，有必要对单个查询做并行化的处理。本文结合时态数据集的线序划分[27]特点，将线序分支分配给小组，按小组分配线程，进行优化，能大大提高数据集在并行服务器中的查询性能，具有重要的实际应用意义。对于索引的更新，由于时态数据库在应用中，随着时间的推移，数据会不断的更新，采用全量式更新远远不能满足，本文还归纳和重新梳理了增量式更新的算法，可用于小数据量的更新。

论文研究的时态数据索引框架 TDIndex 建立在“拟序”（[28]）关系的理论基础之上，是一种协同处理“数据本体”和“时间标签”的索引技术。首先，相对于“代数”关系，论文提出拟序关系（[29]）概念并讨论其基本性质，在时态数据集上进行线序划分，并研究相应的数据结构，建立 TDIndex。其次，论文研究 TDIndex 的数据操作，依据包含关系实现“一次一集合”的查询操作，依据线序划分的性质实现“动态”的增量式更新操作；接着，构建了一个时态数据平台基于多线程实现的 TempMT_Index 系统，研究时态信息和数据之间的协同处理机制，分别建立索引 TDIndex。对 TDIndex 的更新算法做进一步归纳和分类。

本文主要研究基于时态数据库的数据索引技术。首先在现有工作基础上讨论了一般相点集合上的拟序结构（[30]），建立了基于线序划分的相点集合数据

结构，并以此为数学支撑，在一维相点上建立了基于线序划分的数据结构；提出了一种基于数据索引 TDIndex，讨论了基于 TDIndex 的数据查询和更新算法，实现了查询和增量式更新的动态管理；研究多线程技术，对时态数据集合分别进行线序连续分组、线序交叉分组、线序前部策略分组等，以适应于多线程技术。最后，设计了相应实验仿真，采用通用数据，与现有工作进行比较评估。由此之外还建立一套时态数据平台 TempMT_Index，方便用户使用其他各项的数据。论文工作属于时态数据库的选择查询范畴中。论文主要贡献是研究了索引结构的增量式更新。

通过拟序思想提出新的数据结构，并在此基础上建立一般时态索引 TDIndex，凸显时间特性，从而实现时态与非时态数据的协同处理。最后，论文通过实验与相关的工作进行评估，表明 TDIndex 及其应用的可行性、有效性和优越性。同时，论文工作专注于时态信息本身内在结构与“数据本体”的整合配置，满足现实应用中时态数据管理技术实现的基本要求，在 mysql 的基础上采用专门的时态数据语句 atsql[31]，并构建 atsql 与底层数据库的中间件，搭建 TempMT_Index 实验系统，使得应用具有实际的应用价值，具有可操作性。

第2章 相关基础与技术

本章主要介绍了与本论文研究的相关基础和技术，其中包括时态数据库理论，时态索引的理论，timeDB 和 TempMT_Index 系统理论和多线程相关理论。

2.1 时态数据库

时态数据库主要记录那些跟随时间而变化的值的历史，这类历史值对于某些应用系统有一定的价值。

2.1.1 基本概念

时态数据库理论中时间有三种类型：用户自定义时间、有效时间和事务时间，而数据库包含四种类型：快照数据库、回滚数据库、历史数据库和双时态数据库[1]。

用户自定义时间：指用户根据业务需求或者对业务的理解自行定义的时间。时态数据库不处理用户自定义的类型，因此用户自定义时间是和应用相关的，不在时态数据库的处理范围内。

有效时间：指一个对象在现实世界存在的时间，即一个对象在现实世界中语义为真的时间。可以是过去的、现在的和未来的时间。有效时间可以是时间点、时间点的集合、时间期间和时间期间的集合。例如，张三在 2010 年到 2014 年是大学生，[2010,2014]是“张三是大学生”这个事实的有效时间，再例如李四在 2011 年到 2015 年是本科生，在外工作一年后读研，又在 2016 年到 2019 年是硕士生，[2011,2015]∪[2016,2019]是“李四是大学生”这个事实的有效时间。有效时间在时态数据库系统中处理，对用户透明，但也可以由用户显示的指示。

事务时间：指一个对象在数据库中操作的时间。在数据库系统中，当一个事实或对象存储时，往往记录一个插入时间、入库时间或者叫新增记录时间，这里一般称为事务的开始时刻。而当这个事实或对象被删除时，此时该记录被删除。而在时态数据库中，往往这个事实或对象并不被隐性的删除，这一删除

的时刻称为当前事务的结束时刻。当这个事实或对象更改时，此时的时刻称为当前事务的结束时刻，然后又会产生一个新事务，所以这个结束时刻又是新事务的开始时刻，这些时刻配套事务组合成事务时间。事务时间往往记录数据库变更的时间，所以有时候又称为系统时间。

快照数据库：指现实世界中某一时刻（一般是当前时刻）的事实或者对象。它记录数据库中特定时刻的数据。也可以认为当前使用的非时态数据库即为快照数据库。

历史数据库：采用有效时间来管理的数据库，它里面的每一个元组代表着事实或者对象存在的一个历史状态，其有效时间的表示方法可以是时间点、时间点集合、时间期间或者时间期间集合。

回滚数据库：采用事务时间来管理的数据库，它里面的每一个元组代表着事实或者对象的事务变更、状态演变的历史状态。

双时态数据库：是元组包含一个系统支持的有效时间和一个系统支持的事务时间的数据库，它同时具备快照数据库、历史数据库和回滚数据库的特点。是比较完整的时态数据库。

时态变元 Now：其是绑定当前时间的一个变元，随着当前时间的变化而变化，有效值依赖于当前时间。

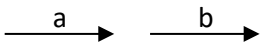
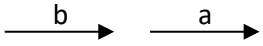
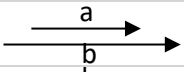
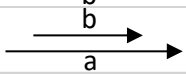
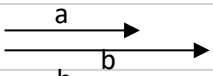
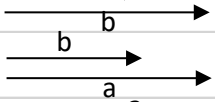
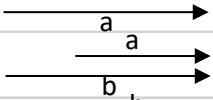
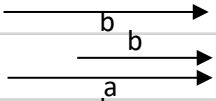
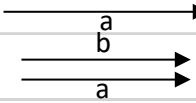
2.1.2 时态期间的关系

Allen 的论文中描述了 13 种时态期间的关系，其为时态关系做出了很大的贡献，如表 2-1 所示，本文主要讨论的包含关系为 Contains(a,b)，被包含关系为 During(a,b)。

2.2 时态索引

在传统的关系数据库中，为了提高查询效率，使用到了索引来解决查询效率的问题。在现有的数据管理系统中，B-tree 和 B^+-tree 得到了大量的应用，在一些空间数据库中还加入了 R-tree 索引来提高空间数据的查询效率。在时态数据库中也可以使用时态索引技术来提高数据查询效率，其采用上述这些数据结构做出了不同特点的索引。

表 2-1 13 种时态期间的关系

时态关系	图例	备注
Before(a, b)		a在b之前发生
After(a, b)		a在b之后发生
During(a, b)		a在b内部
Contains(a, b)		b在a内部
Overlaps(a, b)		a早于b，且ab相交
Overlapped-by(a, b)		b早于a，且ab相交
Meets(a, b)		a结束点与b的开始点重合
Met-by(a, b)		b结束点与a的开始点重合
Starts(a, b)		a与b共同开始，a先结束
Started-by(a, b)		a与b共同开始，b先结束
Finishes(a, b)		b先开始，a与b共同结束
Finished-by(a, b)		a先开始，a与b共同结束
Equals(a, b)		a与b重合

2.2.1 传统数据索引技术

B-tree 是 1972 年提出的，用于基于磁盘检索的很多问题，后来在数据库中大量使用。其主要的特点是树为平衡树，保证检索的时间复杂度总为 O(logn)，其检索和更新只影响部分结点，只影响磁盘的少数页，利用局部访问的特点，既保证磁盘空间利用率，用减少磁盘的读取次数。

B^+-tree 是 B-tree 的变种，其与 B-tree 的区别在于其内部只存储关键字，不存储记录，而在底层的叶节点才存储记录。

R-tree 是对于 B-tree 在空间上的二维扩展，B-tree 的关键字序列可以看成一条线，每一个关键字为一个点，而 R-tree 的关键字序列则可以看成一个坐标轴，每一个关键字为一个矩形。也有很多变种，其是一种时空索引的数据结构。时态数据库中的时间数据有有效时间和事务时间两种，时间数据也是二维

的，所以 R-tree 也可以用于时态数据索引。

2.2.2 时态索引的相关研究

Map21 索引技术，利用传统数据库中成熟的 B^+-tree 技术来处理关系数据库中的有效时间数据，对于双时态数据，该文献构建 2LBIT(Two-Level Bitemporal Indexing Tree)来索引双时态数据。该文可以的索引可以直接使用底层数据库中实现 B^+-tree。

文献提出了 MAP21*3B^+-树模型，采用对时态属性数据的不同域建立索引，基于时态数据类型的时态索引模型。

基于 R-tree 的时态索引也有，如双时态索引 4R-tree 和 GR-tree 技术。双时态索引 4R-tree 底层采用 R*-tree。

提出了第一个既适用于时态数据，又适用于空间数据的方法，称为 R^{ST}-tree。其支持 R*-tree，也支持时态变元 Now。

例如 Tao 等人提出的 MV3R-tree 和 TPR*-Tree、Chakka 等人提出的 SETI 索引、Abdelguerfi 等人提出的 2-3 TR-tree、Procopiuc 等提出的 STAR-Tree 等。③时态与非时态数据协同处理：其本质思想为针对数据本身特征，整合时态查询与非时态查询，例如时态处理基于 B+-tree 的 TB-tree 和 SEB-tree，时态处理基于 R-tree 的 RT-tree、HR-tree 和 HR+-tree，将时态数据与结构协同处理的 TempIndex 和 TempSumIndex 等

本文通过拟序思想提出新的数据结构，并在此基础上建立一般时态索引 TDIndex，凸显时间特性，对时态与非时态数据的协同处理。

2.3 timeDB 和 tempDB

在国外，timeDB（[10]）实现了时态数据处理的基本功能，被认为是比较完整的时态数据库系统。tempDB 作为国内时态数据模型软件的探索者，和 timeDB 比较主要有这些不同。

(1) 都全面支持有效时间

timeDB 和 tempDB 一样，使用了标准的结构化的查询 ATSQL(Applied TSQL)作为数据查询、操作和定义，都支持时态数据的有效时间。

(2) 都暂未支持事务时间

考虑到有效时间和事务时间在结构上相同，而且 ATSQL 上的两者有正交关系。所以 timeDB 和 tempDB 在这里采用分阶段支持的方法，先处理有效时间，在处理有效时间技术十分成熟的时候，再将事务时间加进来，进而实现支持事务时间的数据库和双时态数据库。

(3) 时态完整性的差别

与传统关系数据库相似，时态数据库也有其对应的完整性要求，包括：时态实体完整性、时态参照完整性和用户自定义完整性。TimeDB 和 tempDB 都实现了时态实体完整性，tempDB 还采用 TRICU 的方式实现时态完整约束能力，但还不够完善。

(4) 时态归并

是指对于不同元组的非时态部分相同，而时态部分具有相交或者相邻关系时，可以对时态数据做归并处理。而针对不同情况，可以分为更新时归并和查询时归并，在更新时归并可以减少元组的存储个数，增加投影查询效率，但会减慢更新效率，而在查询时归并，不会减慢更新效率，但会减慢查询效率。timeDB 使用了查询时归并的方法，tempDB 实现了更新时归并的方法。

(5) 对时态变元 Now 的支持

timeDB 将时态变元作为当前时间的别名进行处理，再插入时作为一个具体时间字段来处理。而 tempDB 将时态变元进行了诸多处理，再插入时插入一个特殊的数据类型，而在查询时，再根据实际查询的时间翻译为当前时间。

(6) 体系结构的差别

timeDB 的词法和语法分析采用字符串识别的方法，可重用性低，模块之间耦合度高，但识别效率快，准确度高。tempDB 采用词法和语法分析模块化方式，可重用性高，模块耦合度低，识别效率慢，准确度有待增强。

(7) 软件界面

timeDB 提供图形化的界面，用户可以在对话框中输入执行语句，但输出的执行结果只以文本的形式输出在命令行下。而 tempDB 提供统一的图形化界面，使用户可以在界面中输入执行语句，查看输出结果，也可以在界面中查看执行出错的结果。

(8) 查询性能优化

timeDB 在做查询的时候没有考虑到查询性能，即增加时态查询索引的情况，根据论文（[32]）得知 tempDB 系统采用了 MAP21 算法的索引，进行了查询优化。

本文中提到的 TempMT_Index 则是在 tempDB 的基础上建立的。可用于 Web 中，能更好地适应网络并用于互联网中学习与交流。

2.4 多线程原理以及技术

线程是 CPU 调度和分配任务的基本单位。在单线程上，当程序在调用 IO 操作（磁盘，网络，数据库）时，往往 CPU 会等待 IO 而不工作（当然这里的不工作指在本程序中的不工作，在其他进程中还是工作的），这个时候则用另一个线程去等待 IO 而当前线程继续执行当前程序中其他的工作，主要是为了解决阻塞的问题。多线程技术由此而来，即为多个线程并发执行的技术，通俗来讲就是一个任务划分为多个部分执行。其主要体现在软件上的多线程技术和硬件上的多线程技术。

2.4.1 多线程的硬件支持

对于单台计算机的单个核心的单线程处理器而言，在软件层面，一个任务被划分成了多个部分执行，其对于用户而言，是多个线程并发执行的，但在实际的底层采用的方式是则是 CPU 在不同线程之间进行切换，是一种伪并行计算。只是由于线程切换得太快，很难感知，而在线程进行切换的时候，问题就来了，如何切换回原来线程，这里采用的是每一个线程需要保存自己的一组寄存器集，保证线程可以来回切换。

超线程技术：指采用特殊的硬件指令，将两个逻辑内核模拟成两个物理芯片的技术，从而使单核 CPU 实现线程级的并行计算。其采用的初衷来源于，单个 CPU 在执行计算时往往不能充分被利用。在这种情况下可以大大提高 CPU 的空闲率，实质上是调动 CPU 的空闲资源。超线程技术最早应用于 Intel 的 Xeon 处理器，后来在其公司生产的各种 CPU 上使用。

多核 CPU：即将多个核心封装在一起形成一个 CPU，每个核都是单独的处

理器。其特点是可以在硬件级别上实现多线程处理，即一个核心执行一个线程。目前市面上众多的计算机基本都采用这种 CPU，这也是后面本文讲多线程优化索引的意义所在。多核 CPU 技术的演进如图所示。

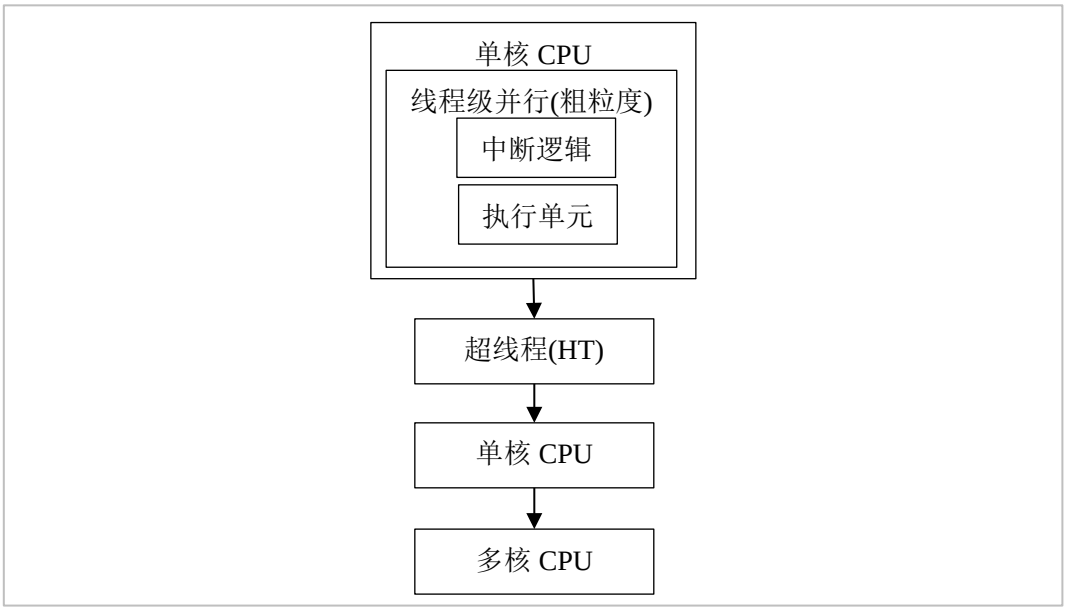


图 2-1 多核 CPU 技术演进图

多处理器计算机：将多个 CPU 通过一定的通信方式组合在一起，多见于大型机、巨型机和超级计算机上，多用于大量并行的计算，在应用方面虽然不能进行多线程，但其本质也是一种并行的方式。

2.4.2 多线程与算法设计

在多核 CPU 技术，超线程技术等出现之前，多线程就已经应用了。主要应用在 IO 的阻塞上，即当程序调用 IO 时，程序不等待，创建另一个线程去等待 IO 的结果，程序继续执行本程序中的其他工作。即使这样，程序是多线程的，但在单核平台上，也只能并发执行，而不是并行的执行，而且多线程在单核平台上来回切换也许需要花费时间和资源开销，所以往往在性能上有时候比不上单线程程序的情况。当然在 CPU 执行单元没有充分利用时，多线程程序还是能更快。

由于硬件的发展，多核 CPU 平台大量的普及，单线程的程序无法也不能由硬件层面自动转换为并行运算（[33]），因为其涉及到程序的先后执行顺序问题。在操作系统层面，linux 和 windows 等系统也只是做了将多线程序分配到多

核 CPU 的这一重要支持，无法自动将已有的单线程程序进行转换。但这就浪费大量的资源，而又没能提高程序的效率。

为了发挥这一重要硬件优势，在进行软件开发和算法设计阶段，就必须考虑到多线程设计的问题。而为了进行并行算法的设计，就必须考虑三方面的问题：首先必须由串行思维向并行思维进行转变，即将算法中，能并行计算的问题，尽量并行处理，尤其是那些数据量大而又耗时长的任务；其次单一数据的共享访问，即数据复制多份给不同的线程同时访问，尤其是大量读操作时；最后保证各个线程负载均衡，即每个线程的计算任务量分配均衡，保证不因个别线程拖慢程序总体效率，尤其是对于那些需要进行结果合并的程序。

由于一个程序中存在无法分割的串行部分，和可以分割的并行部分，还包括线程的一些其他开销，所以多线程的执行效率（[34]）并不是单纯的单线程乘以多，这里引入一个概念加速系数 S(t)来衡量多线程的执行效率，见公式（2-1）多线程加速系数所示。

$$S(t) = \frac{一个线程的执行开销}{t个线程的执行开销} \qquad 公式（2\text{-}1）$$

2.5 本章小结

本章阐述了时态数据库相关概念，以及基于 B-tree 和 R-tree 相关时态索引的研究，然后介绍了 timeDB 和 tempDB 的特点，最后介绍了多线程的原理以及相关技术，并引入衡量多线程效率的加速系数。

第3章 时态数据索引

时态数据（TD）定义为一个常规数据 RD 和一个有效时间期间标签 VT 组成的二元组 TD=<RD,VT>，VT=[VTs,VTe]，VTs 和 VTe 表示 VT 的起止点（VTs<=VTe）。若 VTs=VTe，VT=[VTs,VTe]即为时刻（instant）。TD 有效时间期间记为 VT(TD]。如果将 VT=[VTs,VTe]看做 VTs-VTe 平面上坐标点[VTs,VTe]，时间期间集Γ就和 VTs-VTe 平面上坐标点集 H（Γ）建立一一对应的关系。本章主要内容包括，时态数据结构、创建时态数据索引以及时态数据的操作。

3.1 时态数据结构

定义 3.1：（时态拟序关系）

拟序关系(quasi-ordering relation)是一种重要的二元关系，指集合 A 上同时满足自反性和传递性的二元关系 R，A 称为拟序集。设存在时态数据集 E，①自反性：$\forall TD_1 \in E$，满足 VT（TD_1）⊆VT（TD_1）；②传递性：$\forall TD_1 \in E$，$\forall TD_2 \in E$，$\forall TD_3 \in E$，若 VT（TD_1）⊆VT（TD_2），且 VT（TD_2）⊆VT（TD_3），则有 VT（TD_1）⊆VT（TD_3）。则集合 A 存在拟序关系$\lesssim$：$\exists TD_1 \in E$，$\exists TD_2 \in E$，$TD_1 \lesssim TD_2 \Leftrightarrow$VT（$TD_1$）⊆VT（$TD_2$）。这种关系称为时态拟序关系。

本文主要讨论时态数据操作，将时间期间集记为Γ，时间期间的 VTs 记作横坐标，VTe 记作纵坐标，构成平面坐标集记为 H（Γ）。为了简化操作，Γ 和 H（Γ）将不进行区分，由于没有讨论常规数据 RD，本文的 TD 和 VT（TD）也不进行区分。

为在Γ上建立数据结构，需要先讨论相应的数据构建算法，其中一种算法为下（右）优先构建算法。

算法 3.1：（H（Γ）“下（右）优先”构建算法）

Step1：找 H（Γ）“最左上方”坐标点 u（i,j）（i=min(VTs),j=（min(VTs)列的 max（VTe））设为 u_{i0}。

Step2：有 u_{i0} 向下遍历到该列上属于 H（Γ）的“最后”坐标点 u_{ik}（k 为该列上的 min（VTe））。

Step3：接着向紧挨 u_{ik} 的右列遍历，即令 i=i+g（g 代表时间粒度）在该列上向下遍历。如果该列存在 $v_{i0}\in H(\Gamma)$：满足 $VTe(u_{ik})>=VTe(v_{ij})$，则令 $u_{i0}=v_{i0}$ 开始返回执行“step2”；否则，$u_{i0}=v_{i0}$，继续执行“step3”直至 $w_{i0}\in H(\Gamma)$：$VTe(w_{i0})=\min\{VTe(u)\mid u\in H(\Gamma)\}$。

Step4：输出列表 $L1=<u_{i0},v_{i0},\cdots,w_{i0}>$。

Step5：$H(\Gamma)=H(\Gamma)\backslash L1$，转向“step1”。

按算法循环次序得到 H（Γ）子集分别记为 L1,L2,…,Lm，将它们首尾相接得 H（Γ）的“下（右）优先”遍历序列记为 S（Γ）=<L1,L2,…,Lm>，Li（1<=i<=m）中元素按照算法获取顺序排序。设 m=max{VTs(u)|u∈Γ}，n=max{VTe(u)|u∈Γ}，则算法时间复杂度为 m*n/2

例 3.1：设 H（Γ）=<[1,10],[1,9],[1,8],[1,7], [2,9],[2,8], [2,7],[2,6],[2,5],[2,4], [3,9], [3,8],[3,5], [3,4], [4,9],[4,7], [4,5], [4,4], [5,10],[5,9],[5,8], [5,7],[5,6], [6,10], [6,7], [7,9],[7,8]>，相应“下（右）优先构建”如图 3-1 所示，此时 S（Γ）=<L_1,L_2,L_3,L_4,L_5>，其中 L_1=<[1,10],[1,9],[1,8],[1,7], [2,7],[2,6],[2,5],[2,4], [3,4],[4,4]>，L_2=<[2,9],[2,8], [3,8],[3,5],[4,5]>，L_3=<[3,9],[4,9],[4,7],[5,7],[5,6]>，L_4=<[5,10],[5,9],[5,8],[6,7]>，L_5=<[6,10],[7,9],[7,8]>。

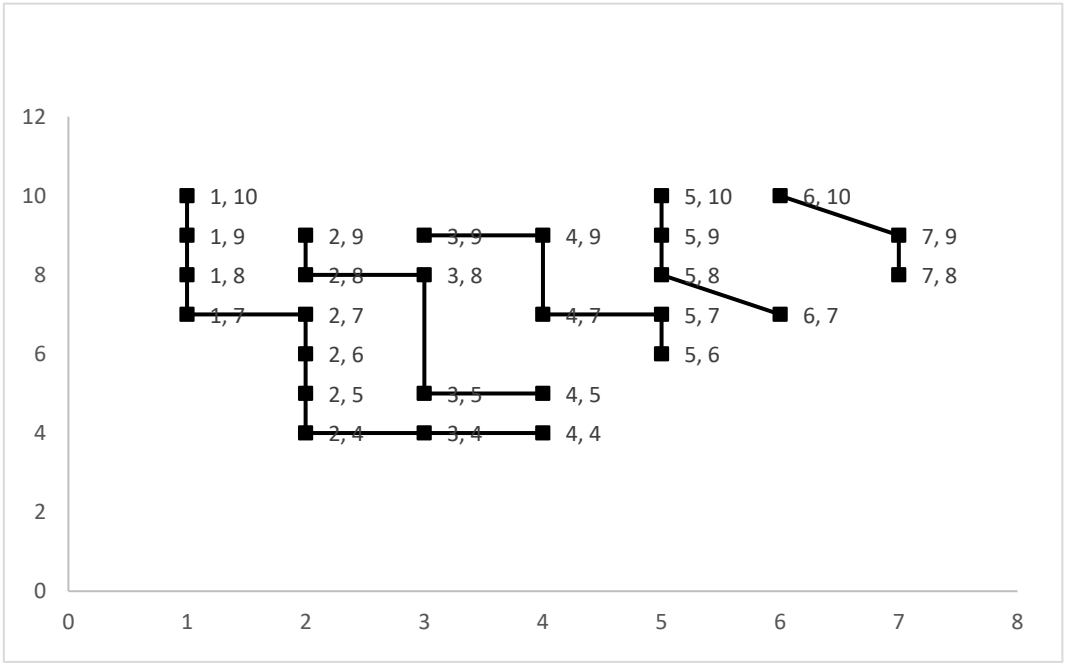


图 3-1 下（右）优先算法实例

定理 3.1：（S（Γ）的基本性质）

①S（Γ）=<L_1,L_2,…,L_m>作为子集 L_i（1<=i<=m）集合是 H（Γ）的一个划分（partition）。

②设 VTs（L_i）和 VTe（L_i）分别是 L_i（1<=i<=m）中元素开始点和结束点序列，VTs（L_i）单调增加，VTe（L_i）单调减少。

定义 3.2：（四分区域）对∀v0∈H(Γ)，通过 v_0 可将 H（Γ）平面分为四个区域：

LU (v_0)={v| v∈H（Γ）∧VTs(v)<=VTs(v_0)∧VTe(v_0)<=VTe(v)}

LD (v_0)={v| v∈H（Γ）∧VTs(v)＜VTs(v_0)∧VTe(v)＜VTe(v_0)}

RU (v_0)={v| v∈H（Γ）∧VTs(v_0)＜VTs(v)∧VTe(v_0)＜VTe(v)}

RD (v_0)={v| v∈H（Γ）∧VTs(v_0)<=VTs(v)∧VTe(v)<=VTe(v_0)}

LU (v_0)、LD (v_0)、RU(v_0)和 RD(v_0)分别为 v0 在 H（Γ）上的“左上”、“左下”、“右上”和“右下”区域。定义 RD (v_0)区域子区域 RDO 如下：RDO (v_0)={v|v∈H（Γ）∧(VTs(v_0)<=VTs(v))∧(VTe(v)<VTe(v_0))}。

例 3.2：设 v0=[4,6]，可将 H（Γ）平面分为四个区域，如图 3-2 所示。

LU[4,6]={[1,10],[1,9],[1,8],[1,7],[2,9],[2,8],[2,7],[2,6],[3,9],[3,8],[4,9],[4,7]}

LD[4,6]={[2,5],[2,4],[3,5],[3,4]}

RU[4,6]={[5,10],[5,9],[5,8],[5,7],[6,10],[6,7],[7,9],[7,8]}

RD[4,6]={[4,5],[4,4],[5,6]}

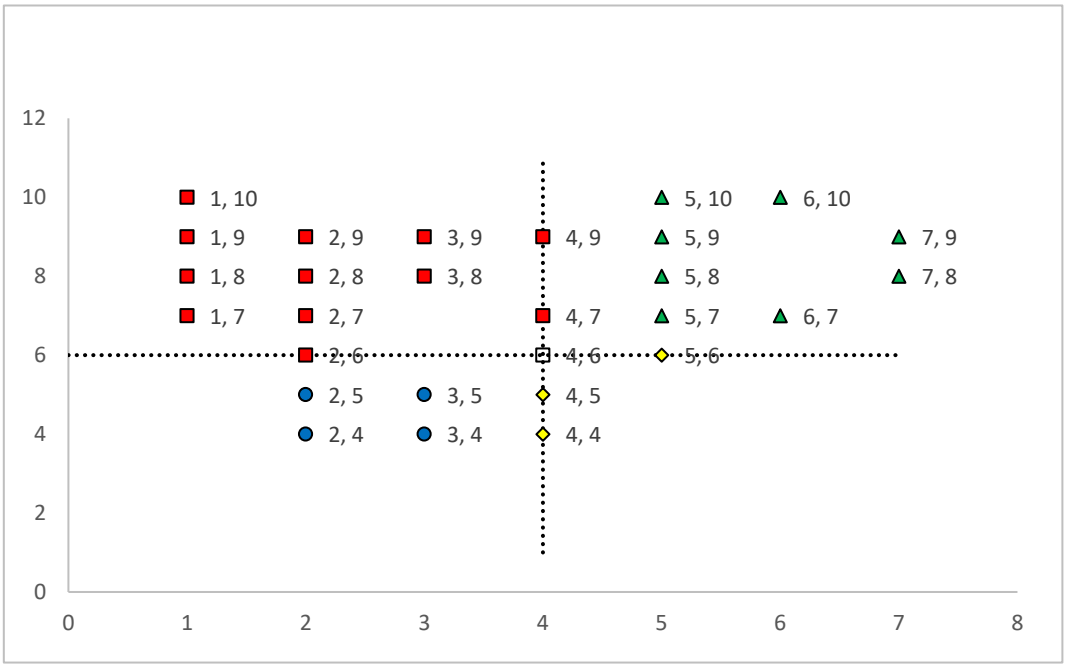


图 3-2 四分区域图

定理 3.2：(四分区域性质) 对任意 $v_0 \in H(\Gamma)$，有以下性质：①$v_0 \subseteq u_0 \Leftrightarrow u_0 \in LU[4,6]$；②$u_0 \subseteq v_0 \Leftrightarrow u_0 \in RD[4,6]$；③$u_0 \in RUP(v_0) \Rightarrow v_0 \cap u_0 = \emptyset$；④$\neg (u_0 \subseteq v_0) \vee \neg (v_0 \subseteq u_0) \Leftrightarrow u_0 \in RU(v_0) \vee u_0 \in LD(v_0)$。

3.2 线序划分和分支

定义 3.3：（线序划分）通过算法 3.1 得到序列的 $S(\Gamma)=<L_1,L_2,\ldots,L_m>$满足以下性质：对于$\forall L_i$，$L_j \in S \wedge i \neq j \Leftrightarrow L_i \cap L_j = \emptyset$,且$\cup L_i = \Gamma (1<=i,j<=m)$。这样的 S(Γ)称为 Γ 上一个线序划分 LOP(Linear Order Partition)，即拟序关系时态数据结构 QOTDS(Quasi-Order Temporal Data Structure)，记为 LOP(Γ)=S(Γ)，而其中的每个 L_i 称为这个 LOP 的一个线序分支 LOB(Linear Order Branch)

$\forall u_0 \in LOB$，LOB 序列中所有 u_0 前驱坐标集（包括 u_0 在内）构成的子序列记为 $LP(u_0)$，相似的，所有 u_0 后继坐标集（包含 u_0 在内）组成子序列记为 $LS(u_0)$。有多种方式可构建 LOP，考虑到利用线序划分的特性，LOB 个数越少，即平均每个 LOB 中包含元素较多时可提高实际操作效率。

定义 3.4：(最小线序划分)$\exists LOP_0 \subseteq \Gamma$，对$\forall LOP \subseteq \Gamma$，满足$|LOP_0| <= |LOP|$，则 LOP_0 为 Γ 上最小线序划分，记为 MinLOP(Minimum Linear Order Partition)。

定理 3.3：(MinLOP 可得性)通过算法 3.1 在 H(Γ)平面上获得的线序 LOP 即为 MinLOP。

3.3 时态数据索引

对于 $L \in LOP(\Gamma)$，记 L 的首结点为 max(L)，尾结点为 min(L)。记 Γmax 为 LOP(Γ)中所有“max(L)”的集合，通过算法 3.1 在 Γmax 上进行线序划分，将得到序列记为 $LOP(\Gamma max)=\{L_i(\Gamma max)\}(1<=i<=|LOP(\Gamma)|)$。类似地，定义 Γmin 为 LOP(Γ)中所有“min(L)”的集合，线序划分记为 $LOP(\Gamma min)=\{L_r(\Gamma min)\}(1<=r<=|LOP(\Gamma)|)$。称 $L_i(\Gamma max)$和 $L_r(\Gamma min)$为“端点 LOB”，并称定义中的线序分支为“元素 LOB”。

定义 3.5：(时态索引)关于 H(Γ)上时态索引满足如下定义，如图 3-3 所示

①Lroot 根结点层$<R_1,R_2,\ldots,R_m>$,$R_i=max(L_i(\Gamma max))(1<=i<=m)$。

②LOP(Γmax)层：在 LOP 上构建

LOP(Γmax)={L_i(Γmax)}(1<=i<=|LOP(Γmax)|)，该层结点为 L_i(Γmax)。

③LOP(Γmin)层：上层结点 L_i(Γmax)中对应元素 LOB 集合构建 LOP(Γmin(L_i(Γmax)))(1<=r<=|LOP(Γmin(L_i(Γmax)))|)，该层结点为 LOP(Γmin(L_i(Γmax)))中 L_r(Γmin(L_i(Γmax)))。

④LOB 层：本层结点为 L_r(Γmin(L_i(Γmax)))对应元素 LOB。

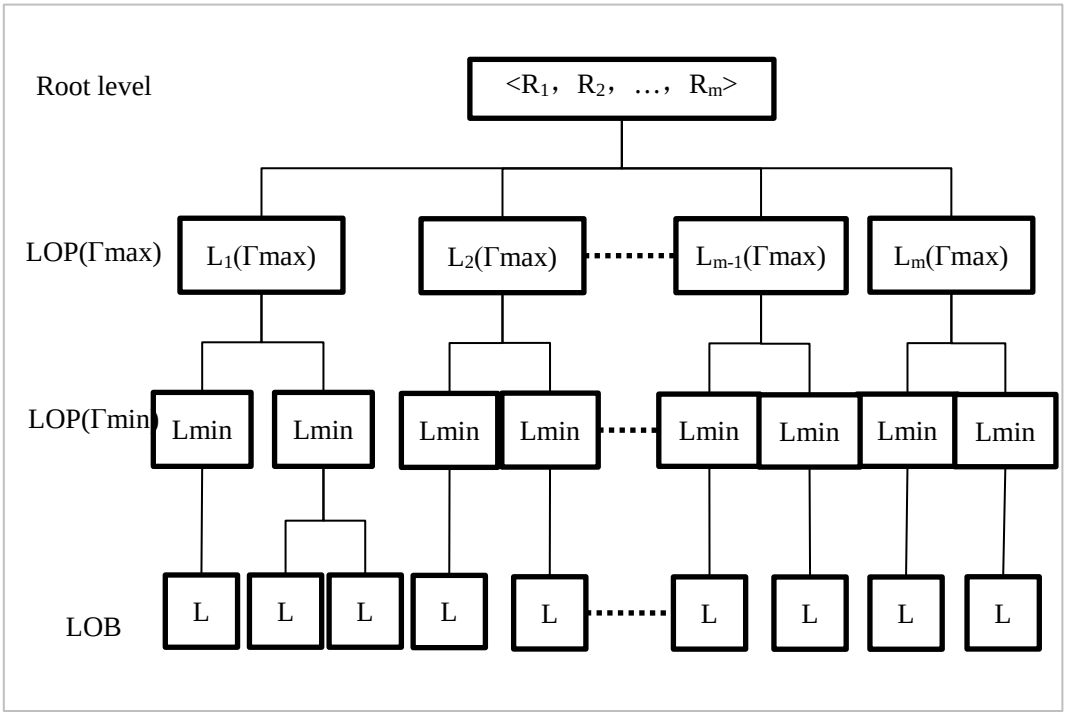


图 3-3 时态索引树形结构图

例 3.3：给定 Γ 上 LOP(Γ)=<L_1,L_2,L_3,L_4,L_5>，L_1=<[1,10],[1,9],[1,8],[1,7],[2,7],[2,6],[2,5],[2,4],[3,4],[4,4]>}，L_2=<[2,9],[2,8],[3,8],[3,5],[4,5]>，L_3=<[3,9],[4,9],[4,7],[5,7],[5,6]>，L_4=<[5,10],[5,9],[5,8],[6,7]>，L_5=<[6,10],[7,9],[7,8]>。LOP(Γ)则如图 3-4 所示。

通过算法 3.1 可得，Γmax=<Max(L_1),Max(L_2),Max(L_3),Max(L_4),Max(L_5)>=<[1,10],[2,9],[3,9],[5,10],[6,10]>，Γmin=<Min(L_1),Min(L_2),Min(L_3),Min(L_4),Min(L_5)>=<[4,4],[4,5],[5,6],[6,7],[7,8]>，将分别进行线序划分可得，如图 3-5 所示。从 Γmax 可获得 LOP(Γmax)=<L_1(Γmax),L_2(Γmax)>，其中 L_1(Γmax)={Max(L_1),Max(L_2),Max(L_3)}=<[1,10],[2,9],[3,9]>，L_2(Γmax)={Max(L_4),Max(L_5)}=<[5,10],[6,10]>。

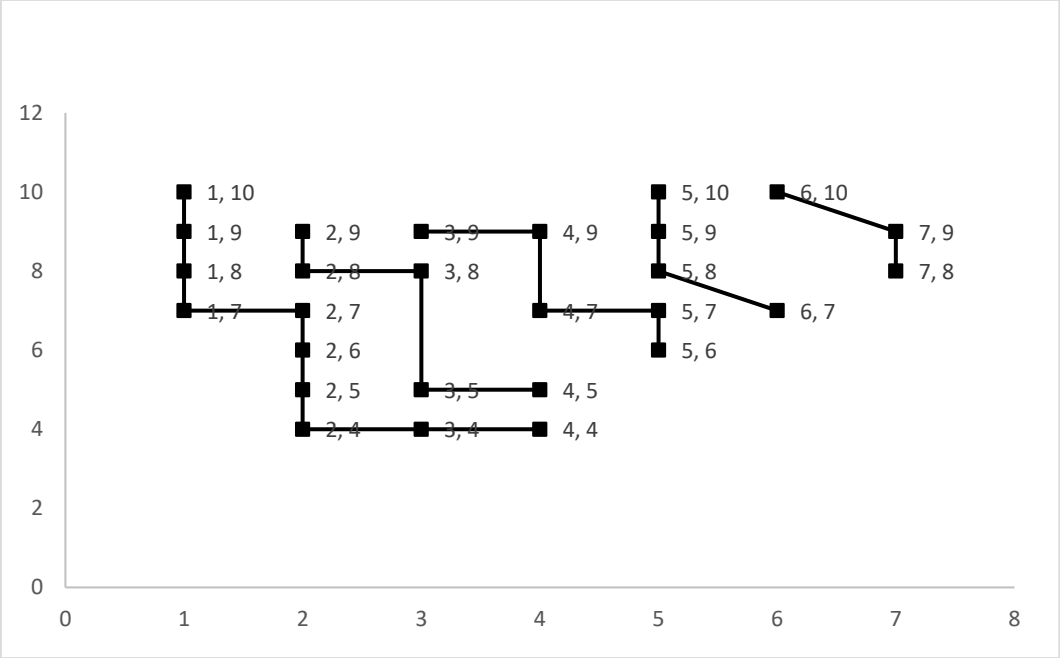


图 3-4 LOB 分支

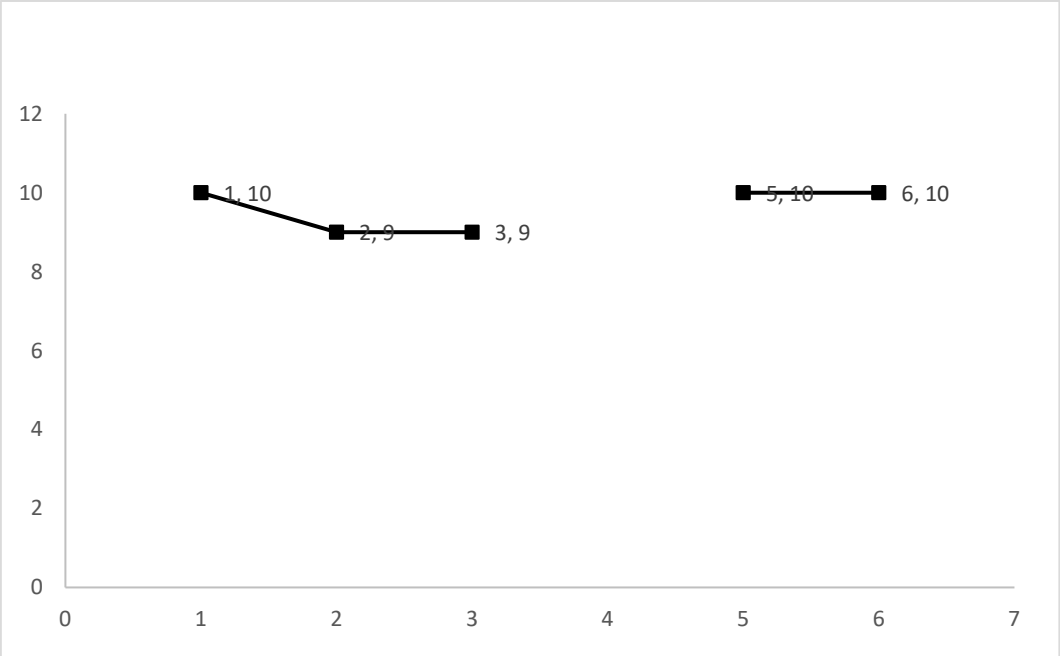


图 3-5 LOP(Γmax)的 LOB 分支

从 Γmin 可获得 LOP(Γmin)=<L_1(Γmin),L_2(Γmin),L_3(Γmin),L_4(Γmin)>，其中 L_1(Γmin)={Min(L_1),Min(L_2)}=<[4,5],[4,4]>，L_2(Γmin)={Min(L_3)}}=<[5,6]>，L_3(Γmin)={Min(L_4)}}=<[6,7]>，L_4(Γmin)={Min(L_5)}}=<[7,8]>，如图 3-6 所示。

将上述线序构成树形结构图，如图 3-7 所示。

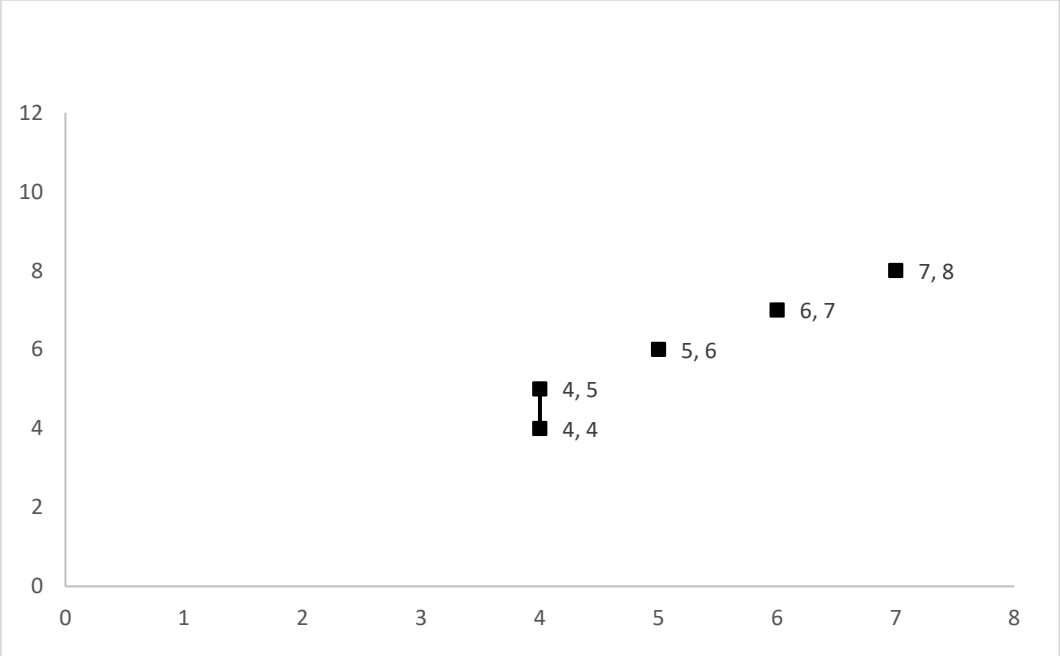


图 3-6 LOP(Γmin)的 LOB 分支

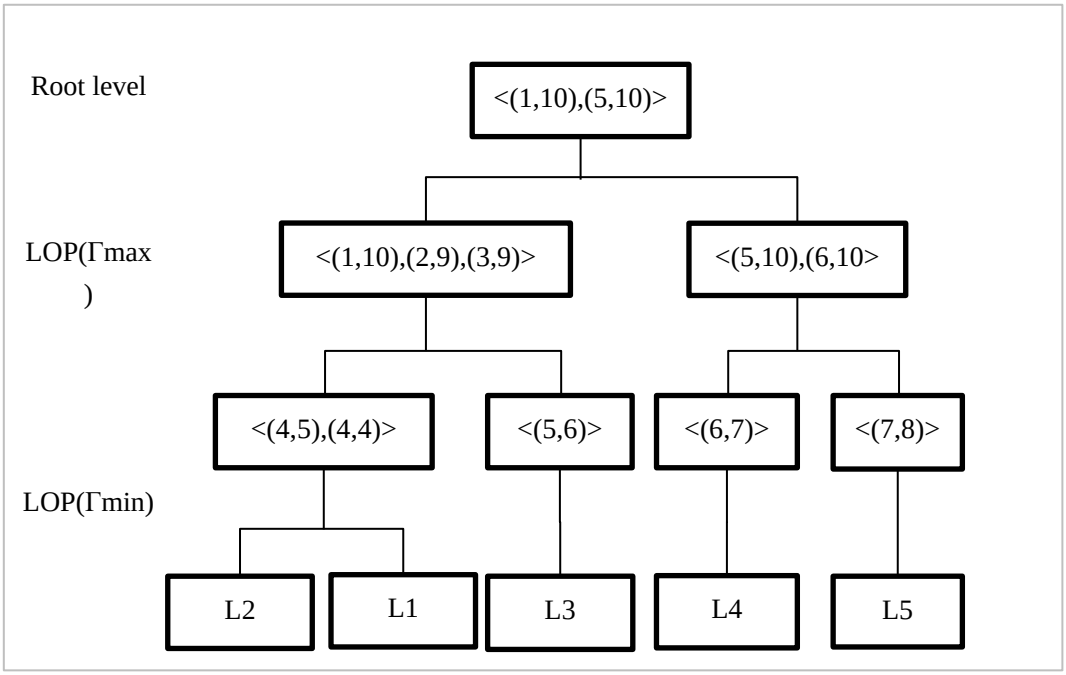


图 3-7 例子的树形结构图

3.4 时态数据查询

在 LOB 中，前一个时间期间一定包含后一个时间期间的特点，可以有如下的基于 LOB 的二分查询算法。给定时间数据集 Γ，根据 3.3 节的内容，建立好

索引结构，现在给一个查询期间 Q，其中 3.4.1 节在该时间数据集 Γ 中找出包含查询期间 Q 的所有时态数据，3.4.2 节在该时间数据集 Γ 中找出被查询期间 Q 包含的所有时态数据。前者称为包含 Q 的查询，后者称为被 Q 包含的查询，下文中如未明确指出时，都为前者的包含 Q 的查询。

3.4.1 时态数据的包含查询

算法 3.2：（基于 LOB 的简单二分包含查询算法）设 LOB=[loc_{max}(LOB),loc_{min}(LOB)]，查询数据为 Q，查询结果集为 R。

Step1：在 LOB 中找二分查找中点 loc_{mid}(LOB)；

Step2：当 VT(Q)⊆loc_{mid}(LOB)，将 loc_{max}(LOB)和 loc_{mid} 之间的数据放入结果集 R 中（即设 n,(max>=n>=mid),R=R∪ Σloc_n(LOB)）。若 loc_{max}(LOB)≠loc_{min}(LOB)时，max=mid+1，返回 step1；否则转 step4。

Step3：当 VT(Q)⊈loc_{mid}(LOB)，若 loc_{max}(LOB)≠loc_{min}(LOB)时，min=mid-1，返回 step1；否则转 step4。

Step4：返回结果集 R。

该算法亦可扩展为适用于基于 LOB 的二分相交查询算法，设 n 为每个 LOB 分支结点的个数，即 n=min-max+1，则算法时间复杂度为 O(logn)。

例 3.4：有查询期间 Q=[4,5],LOB=<[1,10],[1,9],[1,8],[1,7],[2,7],[2,6],[2,5],[2,4],[3,4],[4,4]>，则包含查询结果集 R=<[1,10],[1,9],[1,8],[1,7],[2,7],[2,6],[2,5]>，查询步骤见图 3-8 所示，得每个中点始末点与 Q 的始末点比较次数为 4*2=8 次。

算法 3.3：（基于 LOB 的始末序列二分包含查询算法）由定理 3.1 可知 VTs(LOB)的单增性和 VTe(LOB)的单减性，设 Ls 为当前 LOB 的 VTs(LOB)序列，序号 i(max<=i<=min)，Le 为当前 LOB 的 VTe(LOB)序列，序号 j(max<=j<=min)，查询数据为 Q，查询结果集为 R。

Step1：当 VTs(Q)<Ls_{max} 或者 VTe(Q)>Le_{max}，则 R=∅，转 step14。

Step2：在 Ls 中折半查找中点 Ls_{mid}，在 Le 中折半查找中点 Le_{mid}。

Step3：当 VTs(Q)<Ls_{mid} 且 VTe(Q)>Le_{mid} 时，若 max<min 时，min=mid-1，返回 step2；否则转 step13。

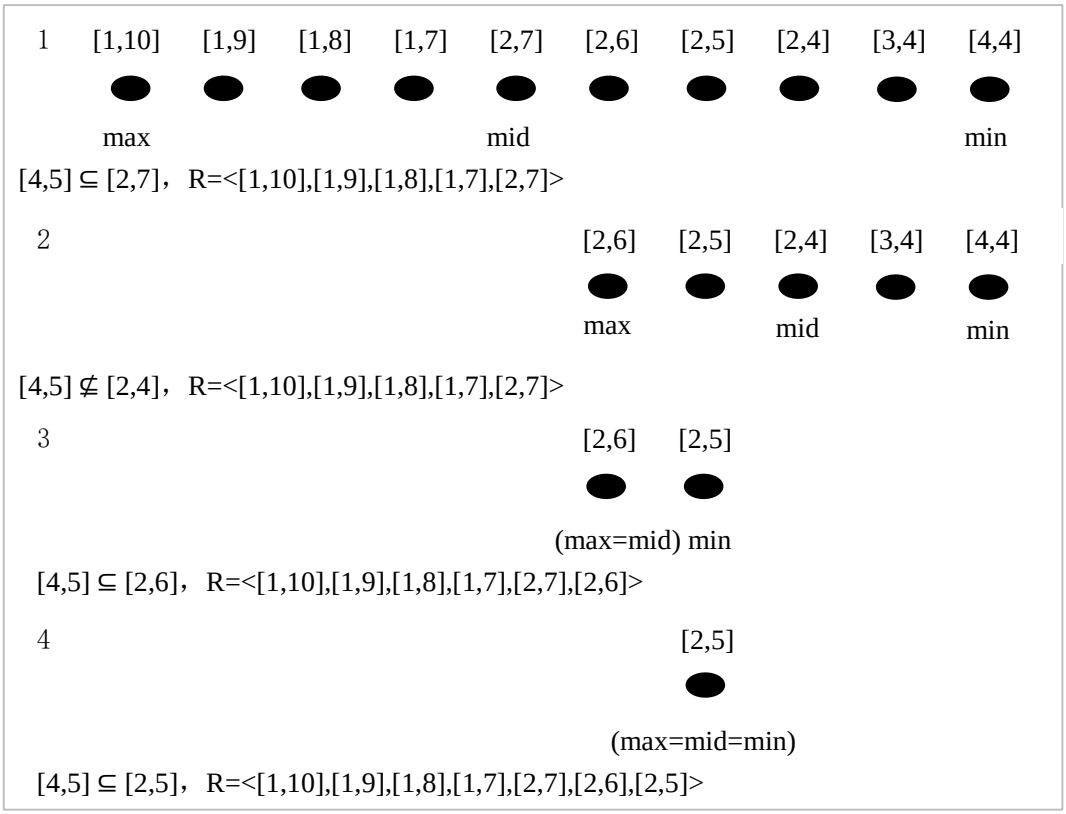


图 3-8 基于 LOB 的二分包含查询算法实例

Step4：当 VTs(Q)>=Ls_{mid} 且 VTe(Q)<=Le_{mid} 时，若 max<min 时，max=mid+1，返回 step2；否则转 step13。

Step5：当 VTs(Q)<Ls_{mid} 且 VTe(Q)<=Le_{mid} 时，min=mid-1，转 step7。

Step6：当 VTs(Q)>=Ls_{mid} 且 VTe(Q)>Le_{mid} 时，min=mid-1，转 step10。

Step7：只需在 Ls 中折半查找中点 Lsmid。

Step8：当 VTs(Q)<Lsmid，若 max<min 时，min=mid-1，返回 step2；否则转 step7。

Step9：当 VTs(Q)>=Lsmid，若 max<min 时，max=mid+1，返回 step2；否转 step7。

Step10：只需在 Le 中折半查找中点 Le_{mid}。

Step11：当 VTe(Q)>Le_{mid}，若 max<min 时，min=mid-1，返回 step5；否则转 step10。

Step12：当 VTe(Q)<=Le_{mid}，若 max<min 时，max=mid+1，返回 step5；否则转 step10。

Step13：将 LOB 从头到 mid 的所有结点放入结果集 R 中。

Step14：返回结果集 R。

设 n 为每个 LOB 分支结点的个数，即 n=min-max+1，且算法时间复杂度也为 O(logn)。但发现，当待查询数据 Q 的开始有效时间小于中点的开始时间且 Q 的结束时间小于或等于中点的结束时间，或者 Q 的开始有效时间大于或等于中点的开始时间且 Q 的结束时间大于中点的结束时间时这两种特殊情况时，根据始点的单增性与结束点的单减特性，可以只在始点序列或者只在终点序列中做比较即可，能减少结束有效时间序列的比较次数，总体的比较次数会少于算法 3.2。

例 3.5：对于查询期间 Q=[4,5],LOB=<[1,10],[1,9],[1,8],[1,7],[2,7],[2,6],[2,5],[2,4],[3,4],[4,4]>，则包含查询结果集 R=<[1,10],[1,9],[1,8],[1,7],[2,7],[2,6],[2,5]>，查询步骤见图 3-9 所示，由于在第 8 个点满足 VTs(Q)=4>=2=Ls7 且 VTe(Q)=5>4=Le7，根据单调特性，始点序列不需要进行比较，即对中点的比较次数为 6 次，少于例 3.4 的 8 次。

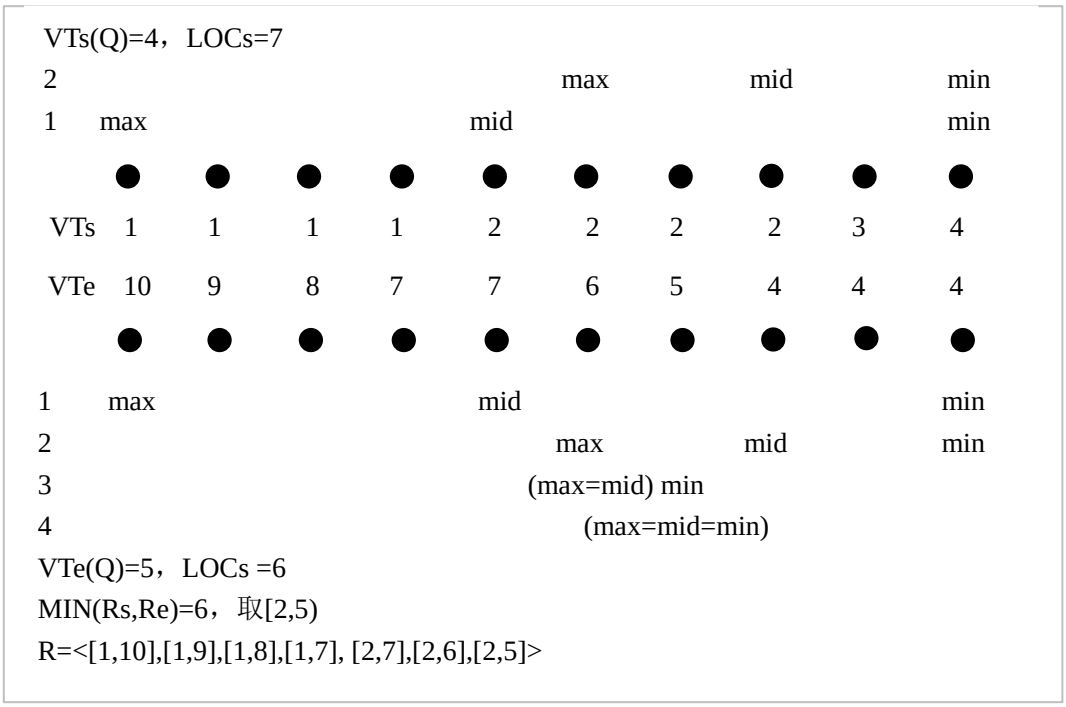


图 3-9 基于 LOB 结点起始序列包含查询算法实例

算法 3.4：（基于时态索引的 LOB 包含查询算法）设查询数据为 Q，查询结果集为 R，待查分支集 RW1、RW2、RW3。

Step1：$R_k \in Lroot(1<=k<=m)$，当 $Q \cap R_k=\emptyset$时，说明 R_k 的结果都不是，否则将 R_k 放入待查分支集 RW1 中。$Lroot=Lroot\backslash\{R_k\}$，若 $Lroot=\emptyset$，转 Step2，否则

返回 Step1。

Step2：依次取出待查分支集 RW1 中的 R_k，对每个 R_k 中的 L(Γmax)调用算法 3.2 包含查询算法，得出的结果放入待查分支集 RW2 中。

Step3：依次取出待查分支集 RW2 中的 LOB，对每个 LOB 的 L(Γmin(LOB))调用算法 3.2 或算法 3.3 包含查询算法，将包含查询期间 Q 的整条 LOB 中的所有结点放入结果集 R 中，将不包含查询期间 Q 的 LOB 放入待查分支集 RW3 中。

Step4：依次取出待查分支集 RW3 中的 LOB，对每个 LOB 调用算法 3.2 或算法 3.3 包含查询算法，将包含查询期间 Q 的结点放入结果集 R 中。

Step5：返回结果集 R。

根据算法 3.2 和算法 3.3，对于包含算法 3.2 的基于算法 3.4 的算法称为基于时态索引 LOB 的简单二分包含查询算法，对于包含算法 3.3 的基于算法 3.4 的算法称为基于时态索引 LOB 的始末序列二分包含查询算法。算法时间复杂度都为 O（logn）。

例 3.6：给定例 3.3 的索引，设查询期间为[4,5)，则查询结果集 R=<[1,10],[1,9],[1,8],[1,7],[2,7],[2,6],[2,5],[2,9],[2,8],[3,8],[3,5],[4,5],[3,9],[4,9],[4,7]>。如图 3-10 所示。

3.4.2 时态数据的被包含查询

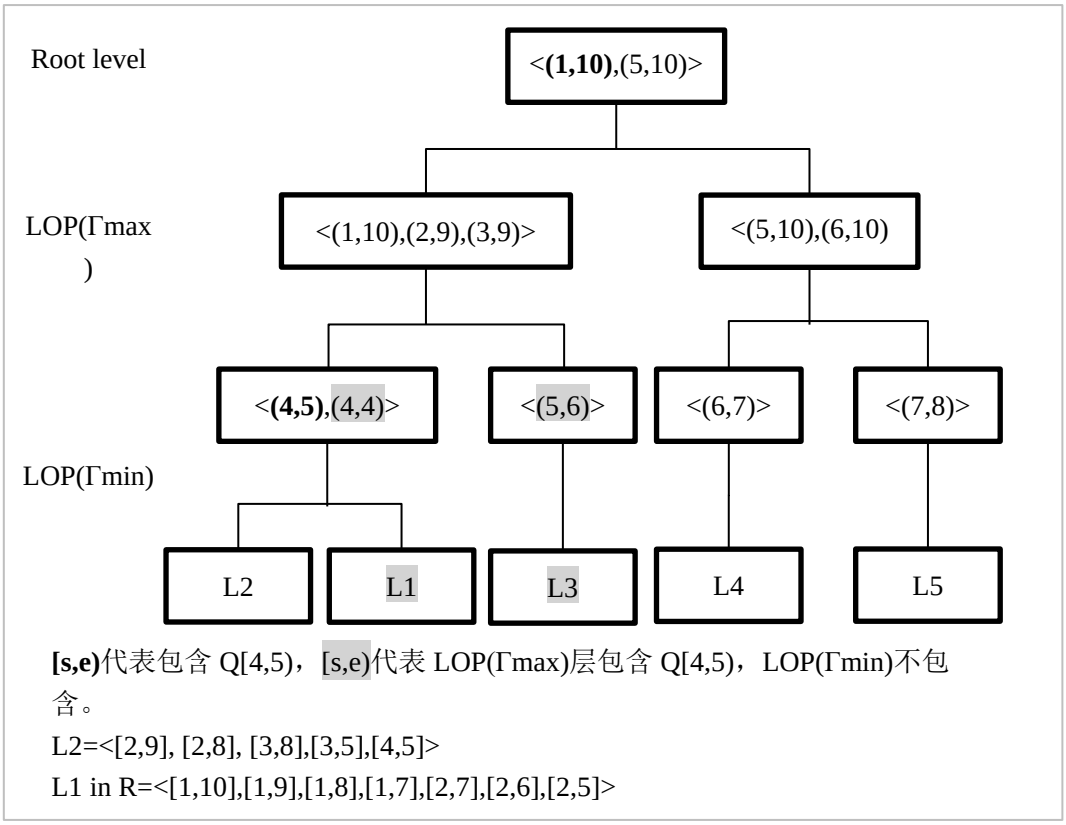
算法 3.5：（基于 LOB 的二分被包含查询算法）设 LOB=[loc_{max}(LOB),loc_{min}(LOB)]，查询数据为 Q，查询结果集为 R。

Step1：在 LOB 中找二分查找中点 loc_{mid}(LOB)；

Step2：当 VT(Q)⊇loc_{mid}(LOB)，将 loc_{mid}(LOB)和 loc_{min} 之间的数据放入结果集 R 中（即设 n,(mid>=n>=min),R=R∪Σloc_n(LOB)）。若 max<min 时，min=mid-1，返回 step1；否则转 step4。

Step3：当 VT(Q)⊉loc_{mid}(LOB)，若 loc_{max}(LOB)≠loc_{min}(LOB)时，max=mid+1，返回 step1；否则转 step4。

Step4：返回结果集 R。



[s,e)代表包含 Q[4,5)，[s,e)代表 LOP(Γmax)层包含 Q[4,5)，LOP(Γmin)不包含。

L2=<[2,9], [2,8], [3,8],[3,5],[4,5]>

L1 in R=<[1,10],[1,9],[1,8],[1,7],[2,7],[2,6],[2,5]>

图 3-10 基于时态索引的包含查询算法实例

设 n 为每个 LOB 分支结点的个数，即 n=min-max+1，则算法时间复杂度为 O(logn)。

例 3.7：有查询期间 Q=[5,10],LOB=<[3,9],[4,9],[4,7],[5,7],[5,6]>，则包含查询结果集 R=<[5,7],[5,6]>，查询步骤见图 3-11 所示。

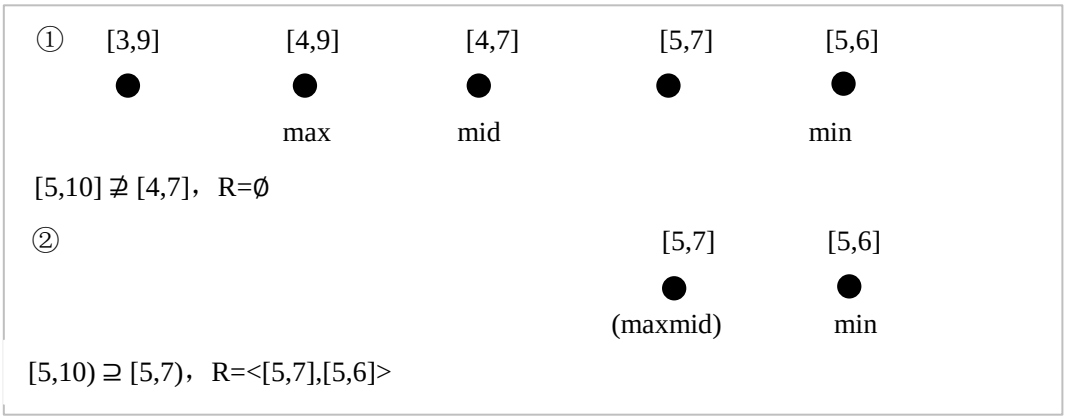


图 3-11 基于 LOB 的二分被包含查询算法实例

算法 3.6：（基于时态索引的被包含查询算法）设查询数据为 Q，查询结果集为 R，待查分支集 RW1、RW2、RW3。

Step1：$R_k \in Lroot(1<=k<=m)$，当 $Q \cap R_k=\emptyset$时，说明 R_k 的结果都不是，否则

将 R_k 放入待查分支集 RW1 中。Lroot=Lroot\{R_k}，若 Lroot=∅，转 Step2，否则返回 Step1。

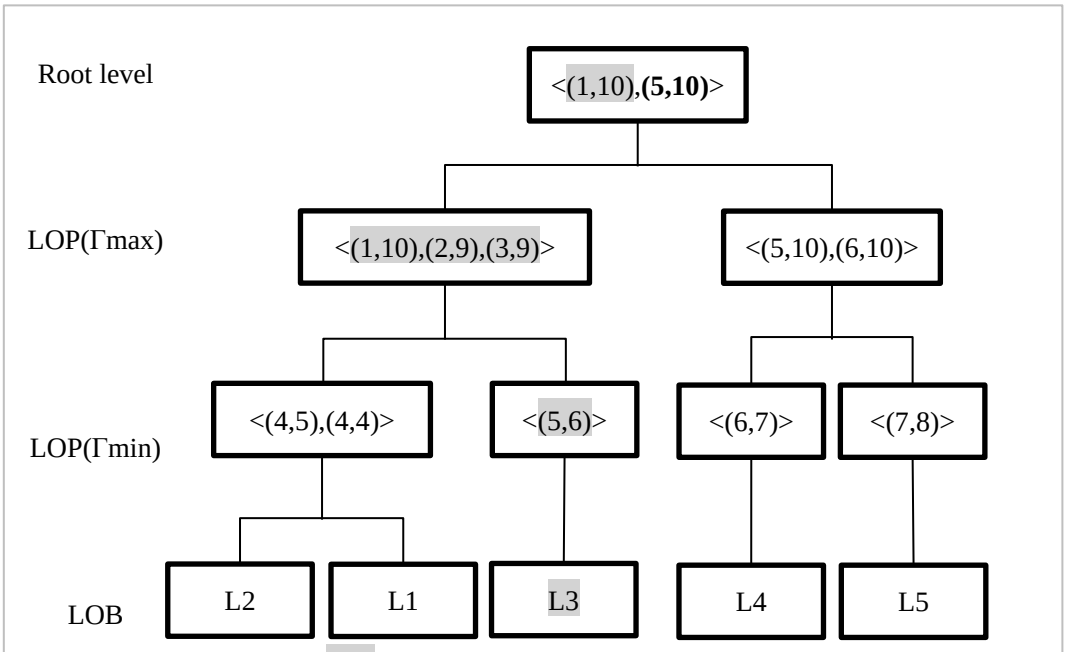
Step2：依次取出待查分支集 RW1 中的 R_k，对每个 R_k 中的 L(Γmax)调用算法 3.5 被包含查询算法，将被查询期间 Q 包含的底层整条 LOB 中的所有结点放入结果集 R 中，将不包含查询期间 Q 的 LOB 放入待查分支集 RW2 中。

Step3：依次取出待查分支集 RW2 中的 LOB，对每个 LOB 的 L(Γmin(LOB))调用算法 3.5 被包含查询算法，将被查询期间 Q 包含的 LOB 放入待查分支集 RW3 中。

Step4：依次取出待查分支集 RW3 中的 LOB，对每个 LOB 调用算法 3.5 被包含查询算法，将被查询期间 Q 包含的结点放入结果集 R 中。

Step5：返回结果集 R。

例 3.8：给定例 3.3 的索引，设查询期间为[5,10)，则查询结果集 R=<[5,7],[5,6],[5,10],[5,9],[5,8],[6,7],[6,10],[7,9],[7,8]>。如图 3-12 所示。



[s,e)代表被 Q[4,5)包含，[s,e)代表 LOP(Γmax)层不被 Q[4,5) 包含，LOP(Γmin)被 Q[4,5)包含。

L4=<[5,10], [5,9], [5,8], [6,7]>

L5=<[6,10],[7,9],[7,8]>

L3 in R =< [5,7],[5,6]>

图 3-12 基于时态索引的被包含查询算法实例

3.5 时态数据更新（插入和删除）

时态数据更新包括对数据的插入和删除，数据变化，则相对应的索引也会变化，索引的变化主要有两种，一种是全量式更新，另一种是增量式更新。在 3.3 节的时态数据索引为树形结构，对树的更新这里可以参考 B-tree 的更新算法。而本节主要讨论底层 LOB 线序的更新问题，包括插入和删除，本节没考虑修改，因为对于时态数据的历史特性来讲，历史的数据一般不允许有修改操作。当然如果实在需要修改，可以采用本节修的删除加插入的两种操作结合的方式来做，也留给今后进行研究。

3.5.1 时态数据插入

插入更新分为全量式更新和增量式更新两种。这里考虑的是增量式更新，即当数据元组增加时，索引会相应的变化。对于全量式更新而言，需要对现有的数据重新建立索引，效率低且原有建立好的索引不能很好的重复利用，针对于大量数据时，更新极其少量的数据来说，是不可取的。如图 3-13 所示为已经建好了的线序分支。

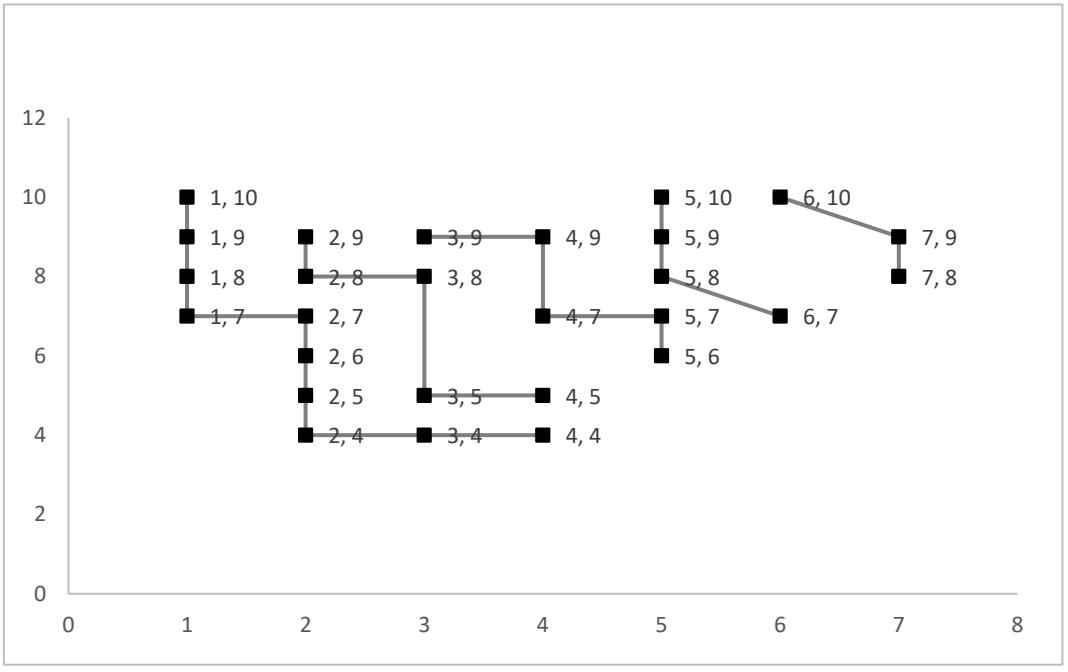


图 3-13 未更新前的数据

定义 3.6：（关于 LOP 和 LOB 的直接前驱与直接后继）设 u_0 为时间期间。

①u_0 关于线序划分 LOP 的直接前驱 $DP(u_0)$为 LOP 中满足条件 $max(Li)\notin RU(u_0)\wedge min(Li)\notin RU(u_0)\wedge VTs(max(Li))<VTs(u_0)$的 Li 中标号最大者。

例：u_0=[5,5]，max(L1)、max(L2)、max(L3)=[1,10]、[2,9]、[3,9]$\notin$RU(u_0)，min(L1)、max(L2)=[4,4]、[4,5]}$\notin$RU(u_0)，VTs(max(L1)、VTs(max(L2)=1、2<VTs(u_0)=5，其中标号最大为 2，所以 $DP(u_0)$=L2。

②u_0 关于线序划分 LOP 的直接后继 $DS(u_0)$为 LOP 中满足条件 $max(Li)\notin LD(u_0)\wedge min(Li)\notin LD(u_0)\wedge VTs(u_0)<=VTe(min(Li))$的 Li 中标号最小者。

例：u_0=[5,5]，max(L1)、max(L2)、max(L3)、max(L4)、max(L5)=[1,10]、[2,9]、[3,9]、[5,10]、[6,10]$\notin$LD(u_0)，min(L3)、min(L4)、min(L5)=[5,6]、[6,7]、[7,8]$\notin$LD(u_0)，VTs([5,5])=5<=VTe(min(L3)、VTe(min(L4)、VTe(min(L5))=6、7、8，其中标号最小为 3,所以 $DS(u_0)$=L3。

③u_0 关于线序分枝 L 的（直接）前驱 $DP_L(u_0)$是指 L 中包含 u_0 的“最小元素”，关于 L 的（直接）后继 $DS_L(u_0)$是指 L 中被 u_0 包含的“最大元素”。

算法 3.7：（插入更新算法）

Case1：以当前插入结点新建立分支，且该分支不影响其他的线序分支的变更。

①LOP$\subseteq$RU(u_0)，插入 u_0=[0,3]，作为新的分支 L0。如图 3-14 所示。

②LOP$\subseteq$LD(u_0)，插入 u_0=[8,11]，作为新的分支 L6。如图 3-15 所示。

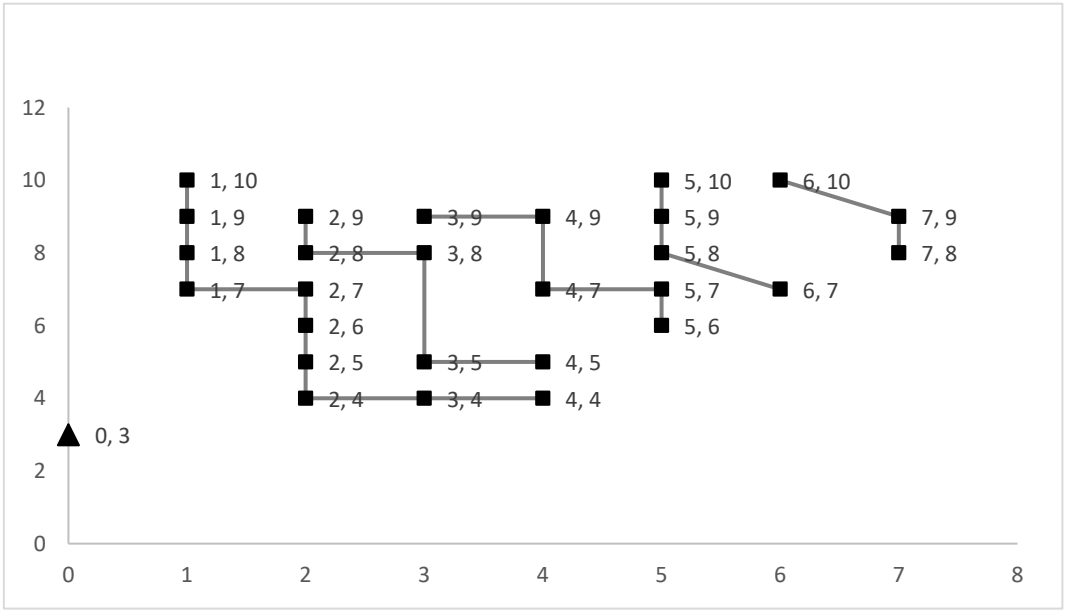


图 3-14 插入[0,3]

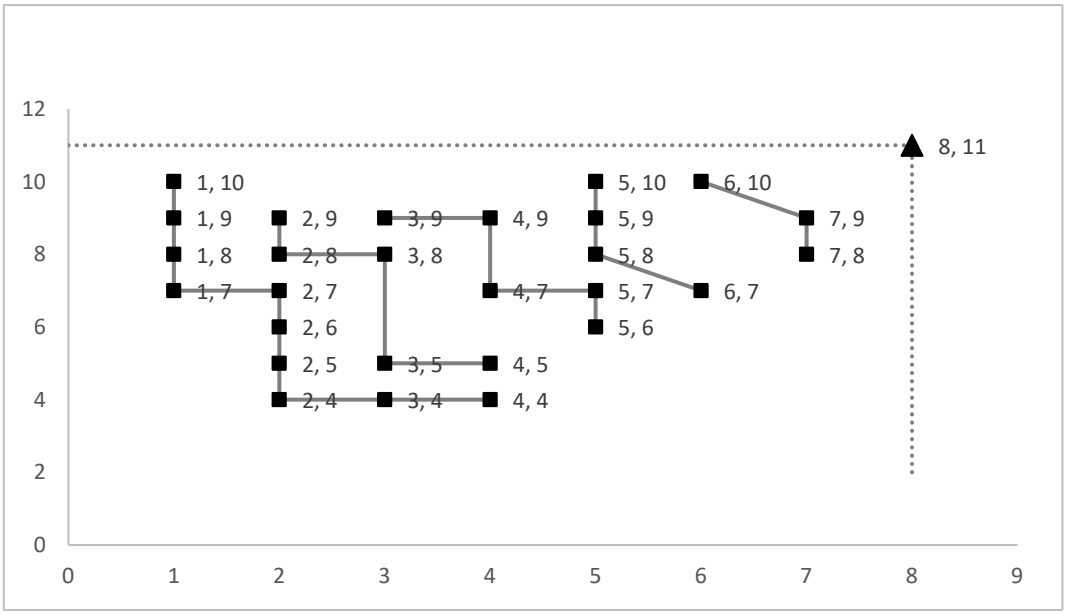


图 3-15 插入[8,11]

Case2：在单线序分支的头部和尾部插入，且该分支不影响其他的线序分支的变更。

①LOP⊆LU(u_0)，插入 u_0=[8,3]，作为分支 L1 的尾部，如图 3-16 所示。

②LOP⊆RD(u_0)，插入 u_0=[0,11]，作为分支 L1 的头，如图 3-17 所示。

③VTs(max(DS(u0)))>=VTs(u0),插入 u0=[4,11]。max(DS(u0))=[5,10]，VTs([5,10])>=VTs([4,11])，接到 max(DS(u0))前面，如图 3-18 所示。

④VTs(min(DP(u_0)))=VTs(u_0),插入 u_0=[4,3]。min(DP(u_0))=[4,4]，

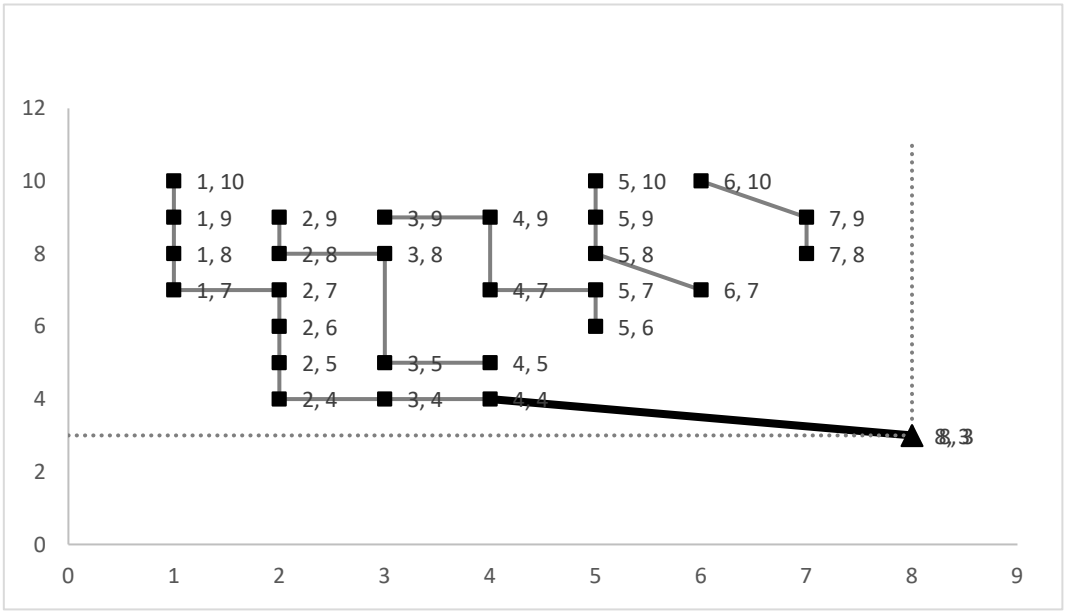


图 3-16 插入[8,3]

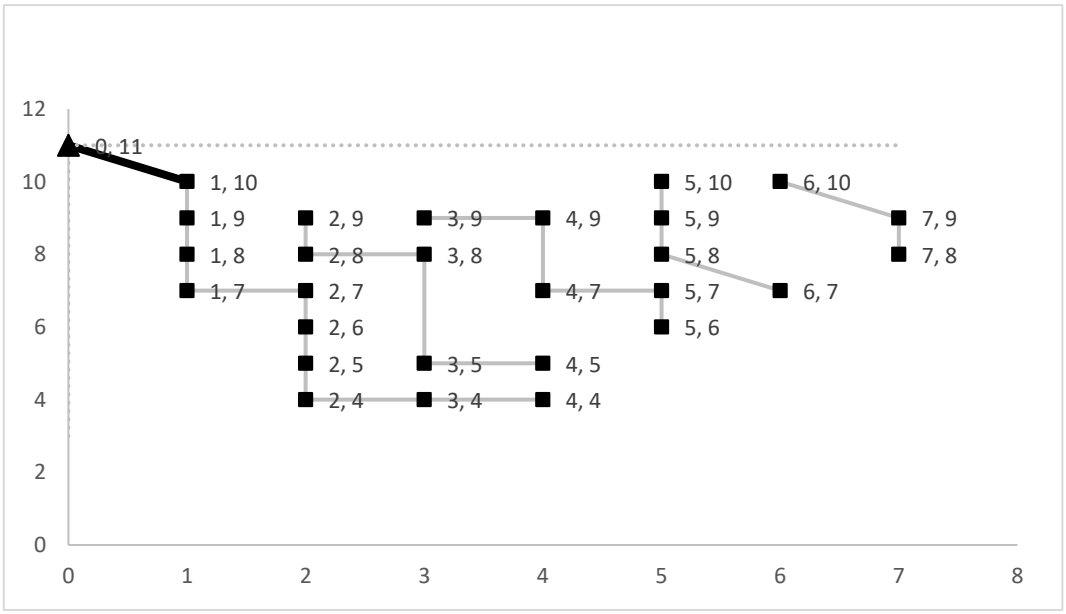


图 3-17 插入[0,11]

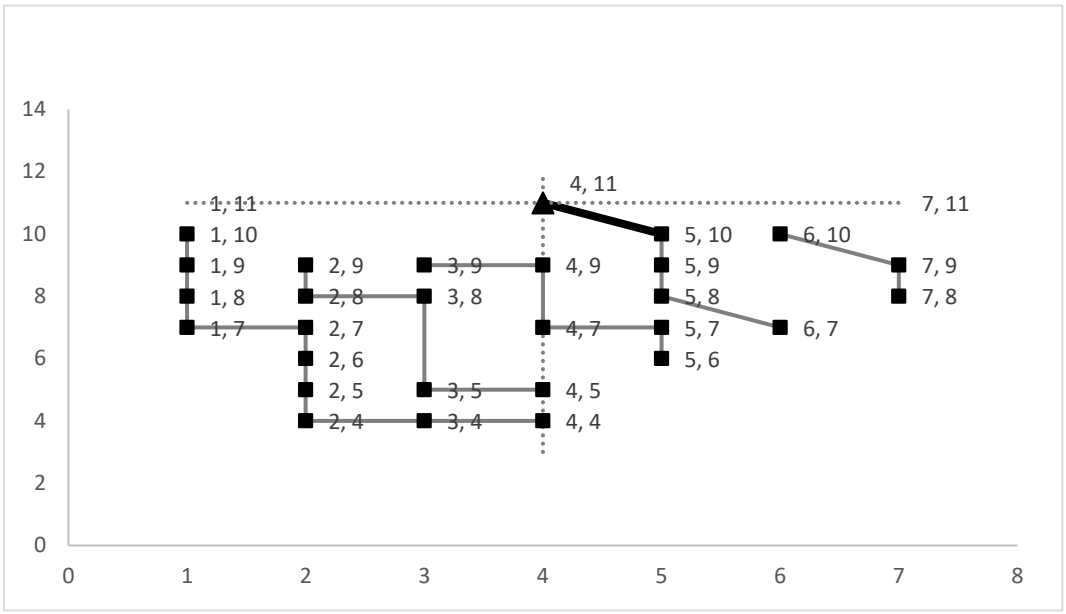


图 3-18 插入[4,11]

VTs([4,3])=VTs([4,4])，接到 min(DP(u_0))后面，如图 3-19 所示。

⑤VTe(min(DP(u_0)))=VTe(u_0)，插入 u_0=[7,7]。min(DP(u_0))=[6,7]，VTe([7,7])=VTe([6,7])，接到 min(DP(u_0))后面，如图 3-20 所示。

Case3：在某个线序分支的中插入

①对<VTs(DS(u_0))>和<VTe(DS(u_0))>通过二分查找，在 L=DS(u_0))确定 v_0=$DP_L(u_0)$和 w_0=$DS_L(u_0)$，(VTs(w_0)=VTs(v_0)=VTs(u_0))∧(VTe(w_0)<VTe(v_0))，将 u_0 插入到 v_0 和 w_0 之间即可。插入 u_0=[3,7]，$DP_L(u_0)$=v_0=[3,8]，$DS_L(u_0)$=w_0=

[3,5]，$VTs(v_0)=VTs([3,8])=VTs([3,5])=VTs(w_0)\wedge VTe(w_0)=5<VTe(v_0)=8$，[1,7]直接接到 v_0=[3,8]，如图 3-21 所示。

②否则，u_0 接入后会导致原先 $L=DS(u_0)$分裂，将分裂的结点，再进行一次插入更新即可。插入 u_0=[4,6]，$DP_L(u_0)=v_0$=[4,7]，$DS_L(u_0)=w_0$=[5,6]，u_0 接入在 v_0 和 w_0 之间，原 $L=DS(u_0)$元素[5,7]被新 LOB “抛离”，将[5,7]作为新插入元素，继续插入操作，如图 3-22 所示。

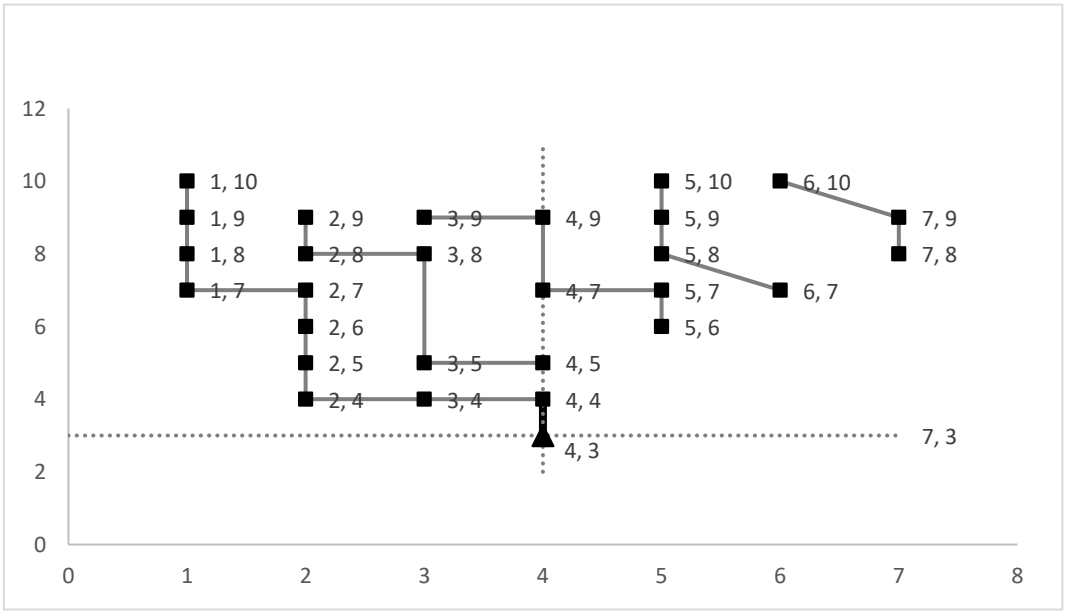


图 3-19 插入[4,3]

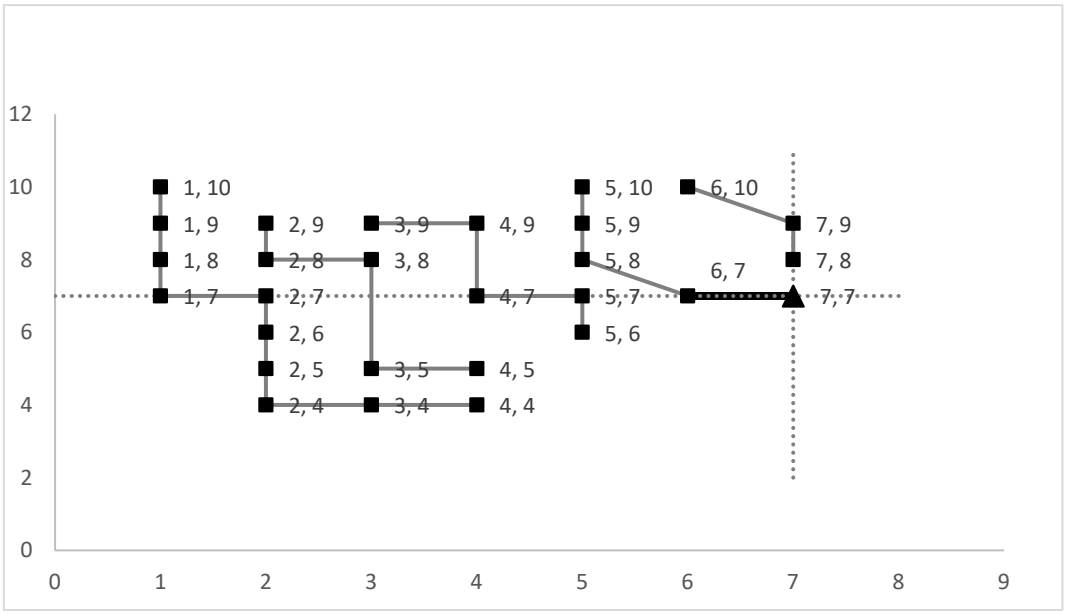


图 3-20 插入[7,7]

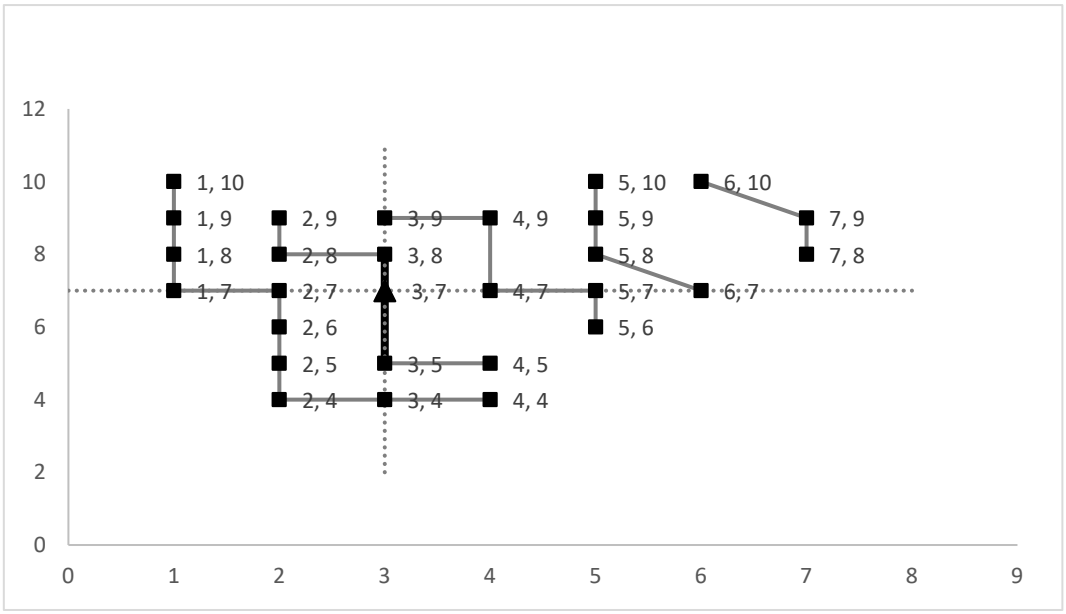


图 3-21 插入[3,7]

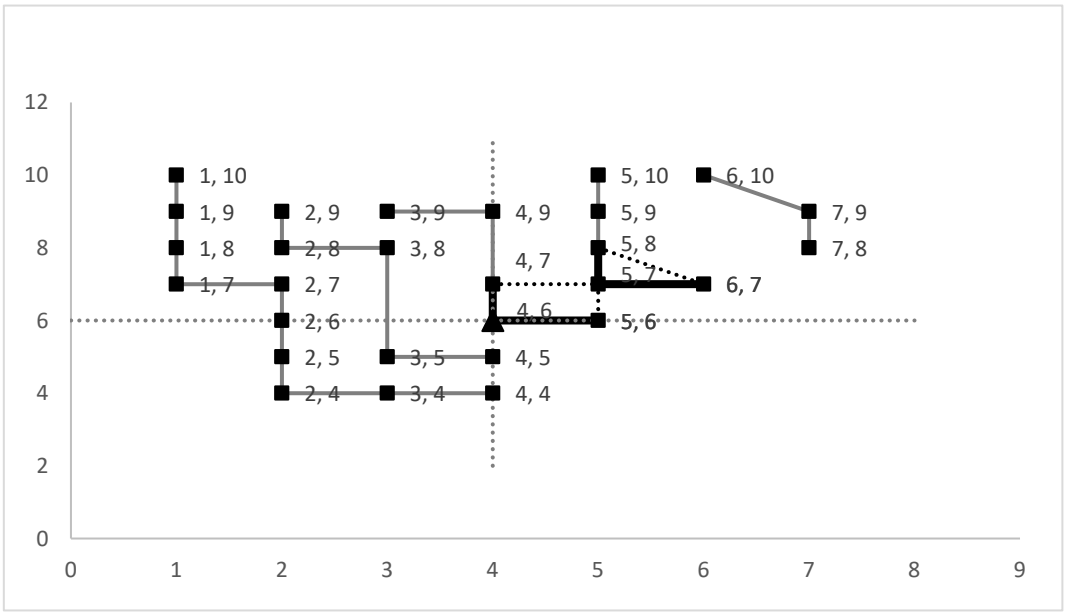


图 3-22 插入[4,6]

3.5.2 时态数据删除

算法 3.8：（删除更新算法）

删除更新分为以下三种情形。

Case1：

①（VTs（v_0）=VTs（u_0）=VTs（w_0））∨（VTe（v_0）=VTe（u_0）=VTe

（w_0））

②(VTs（u_0）<=VTs（v_0))∧(VTe（w_0）<VTe（u_0）),设 y_0=[VTs（v_0),VTe（w_0））

若 y_0=[VTs（w_0),VTe（v_0））∈L_{i0+1}，此时，删除 u_0 后，直接将 v_0 和 w_0 接起即可。如图 3-23 所示，L_{i0}=<[1,10],[1,9],[1,8],[1,7],[2,7],[2,6],[2,5],[2,4],[3,4],[4,4]>，L_{i0+1}=<[2,9],[2,8],[3,8],[3,5],[4,5]>，需在 Li0 删除 u_0=[1,8]，此时 v_0=[1,9]，w_0=[1,7]，删除 u_0=[1,8]，连接 v_0 和 w_0 即可得到新 LOB。

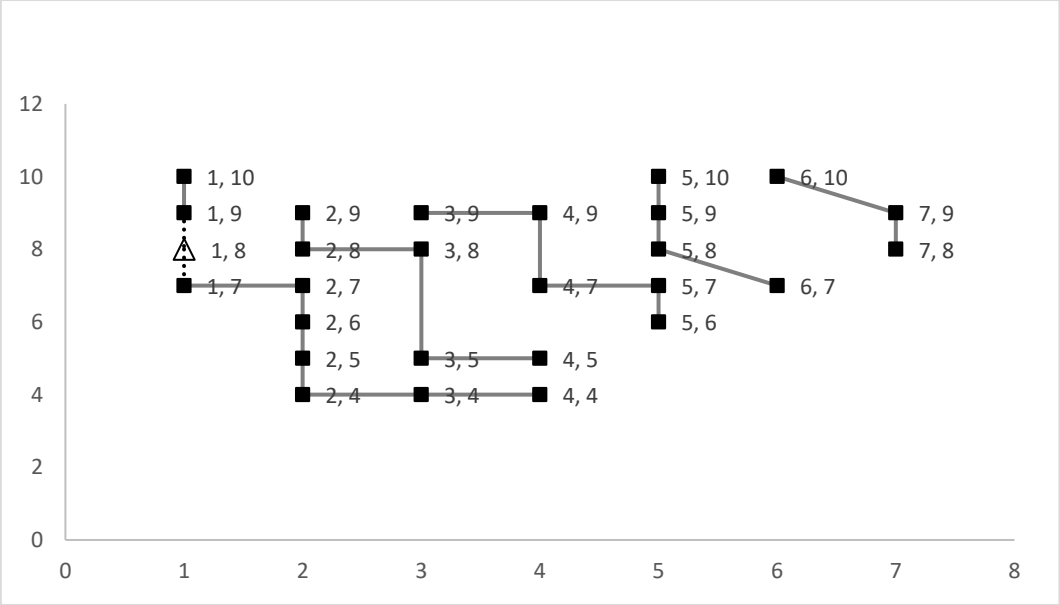


图 3-23 删除[1,8]情形

Case2：(VTs（u_0）<=VTs（v_0))∧(VTe（w_0）<VTe（u_0）),设 y_0=[VTs（v_0),VTe（w_0））。

若 y_0=[VTs（w_0),VTe（v_0））∈L_{i0+1},则删除 u_0 后将 v_0 和 w_0 通过 y_0 连接即的新的 LOB，同时对于 L_{i0+1} 来说，相当与删除了其中的 y0，继续删除算法组建新的 LOB。如图 3-24 所示，L_{i0}=<[1,10],[1,9],[1,8],[1,7],[2,7],[2,6],[2,5],[2,4],[3,4],[4,4]>，L_{i0+1}=<[2,9],[2,8],[3,8],[3,5],[4,5]>，需在 L_{i0} 删除 u_0=[1,7]，此时 v_0=[1,8]，w_0=[2,7]，y_0=[VTs（w_0）,VTe（v_0））=[VTs（[2,7]）,VTe（[1,8]）=[2,8]∈L_{i0+1}，由于删除 u_0=[1,7]后组建新的 LOB 占用了 L_{i0+1} 中的 y_0=[2,8]，需要对 L_{i0+1} 继续进行删除 y_0=[2,8]的操作。

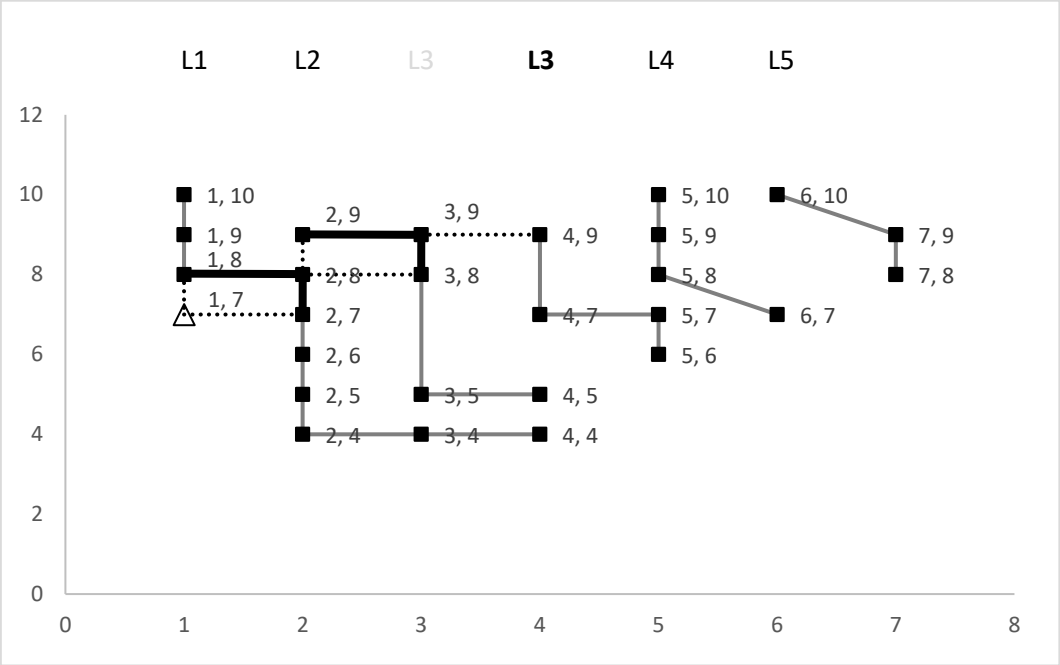


图 3-24 删除[1,7]情形

3.6 本章小结

本章阐述了时态数据结构，线序分支建立，时态索引构建、查询、插入和删除的算法细节。

第4章 基于时态索引的 TempMT_Index 平台

本章主要实现了在 TempMT_Index 中的时态语句到非时态语句的中间件，便于承接现有的数据库系统，接着讨论了时态选择、投影和连接三种关系，最后对于 TDIndex 的查询做了详细的多线程优化模型且提出了四种分支划分策略。并在 TempMT_Index 系统中实现了完整的基于多线程的时态数据查询操作。

4.1 TempMT_Index 中的时态关系运算

根据传统关系数据的选择、投影和连接三种运算，也可以对于时态数据库分为时态选择、时态投影和时态连接三种时态关系运算

4.1.1 时态选择运算

非时态数据选择查询是在 WHERE 子句中放入选择的条件，对查询的结果用相应的谓词做出选择，并得出最终结果。时态选择查询的操作符与非时态选择查询的操作符一致，将时态期间的 equals、befores，overlaps，meets、contains 五种谓词转换为 WHERE 子句中的用相应时间点表示的相应的选择条件，并得出最后结果。时态选择包括三种方式：第一种方式是将时态部分加入 where 子句中与非时态部分一次性通过生成的 SQL 语句在非时态数据库中选出；第二种方式则是先对非时态部分进行选出，对选出结果事先采用本文中提到的时态结构建立对应的索引，最后通过相应的查询算法选出对应的时态数据结果；第三种方式则是首先为整张时态表建立时态数据索引，接着通过相应的查询算法选出对应的未进行非时态选择的数据，最后对非时态数据部分选出最后的时态数据结果。三种时态选择方案如图 4-1 所示。方案 1 为时态数据库在传统数据库中的选择，十分方便，其利用现有的数据库就可以实现，不需要额外的算法。方案 2 与方案 3 都采用了时态索引，效率更高。区别在于非时态部分与时态部分选择的先后顺序。实际效率根据不同实际查询而知。

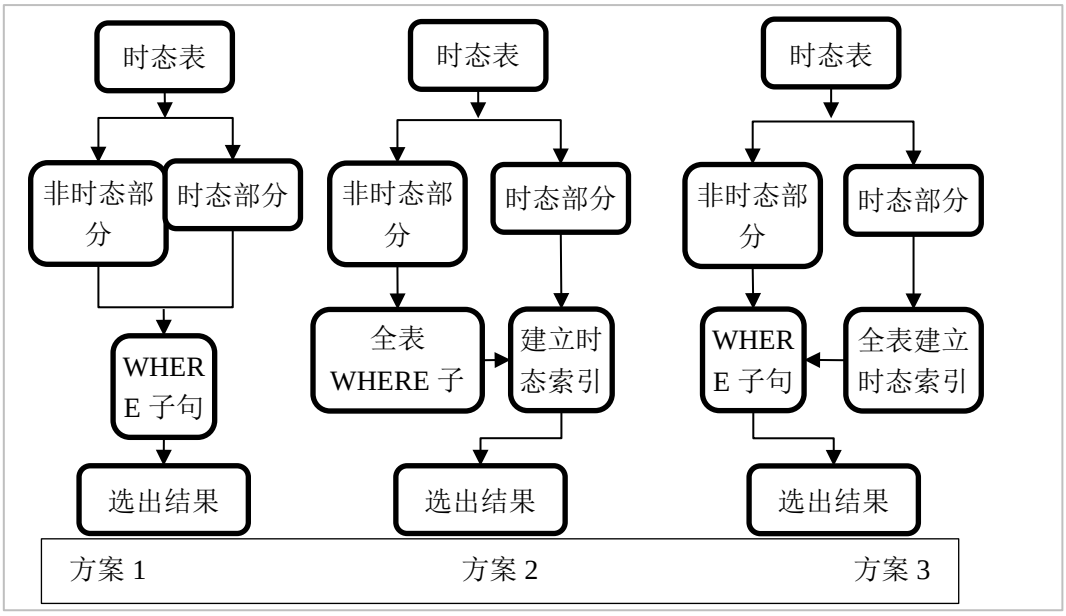


图 4-1 时态选择的三种设计方案

4.1.2 时态投影运算

投影运算是从关系表中选择若干属性组成新的关系。

在非时态数据关系表中有如表 4-1 所示，如果对该表投影 course_id，结果为 100070 和 100090 等。如果对该表投影 name，结果为 Chery1 Wagner、Dan Elliott、Marion Ross、Hazel Anderson 等所有姓名。需要注意的是其取消了原有的某些列，如果剩下列的值完全相同时，需进行去重处理

表 4-1 student 学生信息表

sid	name	e_mail	course_id
0	Chery1 Wagner	Chery1Wagner@21cn.com	100070
1	Dan Elliott	DanElliott@sohu.com	100090
2	Marion Ross	MarionRoss@263.net	100070
3	Hazel Anderson	HazelAnderson@mail.com	100090
…	…	…	…

时态投影是在带有时态区间的时态表中，属性值分为时态部分和非时态部分，在做投影时是对非时态部分的某一属性值进行操作，如果不同元组的同一属性值相同，在普通投影时，只输出单一的属性值，而时态表中增加了时态区间部分，所以还需要考虑时态区间。当时态区间相同时，则结果为单一属性和相同的时态区间。当时态区间不同时，要把所有同一属性值相同的不同元组的

时态区间做集合并运算，得到新的元组，则结果为单一属性和不同时态区间的并集，如果并集有多个区间结果也为多个区间。

算法 4.1：时态期间并集算法

做两个线性表，一个数组 Q，一个为已排数组 R

Step1：将时间期间序列按开始时间从小到大的快速排序算法得 Q，选取第一个时间期间作为 T=Q0，i=1。

Step2：当 Te<Qis 时，将 Te 放入已排数组 R 中;否则 Ts=MIN(Ts,Qis)，Te=MAX(Te,Qie)。再令 T=Qi，i++转 Step2。直到 i=Q.size()，将 Te 放入已排数组 R 中，转 Step3。

Step3：返回结果 R。

在时态数据关系表中有表 4-2 带有时态区间的 student 学生信息表所示，如果对该表投影 course_id 时，结果为

[100089,2000-12-19,2002-06-24],[100084,2001-10-10,2002-01-19],

[100084,2000-11-16,2001-06-09],[100055,2001-08-18,2008-02-05],

[100055,2000-12-28,2001-07-18]。

如果对该表投影 sid 时，结果为

[1,2001-10-10,2002-06-24],[2,2001-08-18,2008-02-05],[2,2000-12-28,2001-07-18]。

表 4-2 带有时态区间的 student 学生信息表

sid	name	e_mail	course_id	vts_timeDB	vte_timeDB
1	Dan Elliott	DanElliott@sohu.com	100089	2001/7/24	2002/6/24
1	Dan Elliott	DanElliott@sohu.com	100089	2000/12/19	2001/8/2
1	Dan Elliott	DanElliott@sohu.com	100084	2001/10/10	2002/1/19
1	Dan Elliott	DanElliott@sohu.com	100084	2000/11/16	2001/6/9
2	Marion Ross	MarionRoss@263.net	100055	2005/5/12	2008/2/5
2	Marion Ross	MarionRoss@263.net	100055	2001/8/18	2006/10/1
2	Marion Ross	MarionRoss@263.net	100055	2000/12/28	2001/7/18
…	…	…	…	…	…

4.1.3 时态连接运算

连接运算是从两个关系的笛卡尔积中选择属性间满足一定条件的元组。

时态连接运算指的是既对非时态数据做传统的关系连接运算，又要对两个关系中的两个时态部分取共有的部分，即对两个时态部分做集合交运算，取交集。

算法 4.2：时间期间序列交集算法

Step1：将时间期间序列按开始时间从小到大的快速排序算法得 VT，选取第一个时间期间作为 R=VTi，i=0。

Step2：当 Re<VT(i+1)s 时，R=⦰，执行 Step3;否则得交集 Rs=MAX(Rs,VT(i+1)s)，Re=MIN(Re,VT(i+1)e)，i++，直到 i=VT.size()，执行 Step3。

Step3：返回结果 R。

4.2 Atsql 与 SQL 的中间件

ATSQL 和 SQL 的中间件，主要功能是接收用户的输入时态查询和非时态查询请求语句，转换成标准的 SQL 语句，并在底层数据库中执行，返回输出结果，流程图如图 4-2 所示。由于研究的需要，仅先实现了 DQL 语句的中间件，和 DDL 语句中 CREATE TABLE 的中间件，预留其他语句的接口。

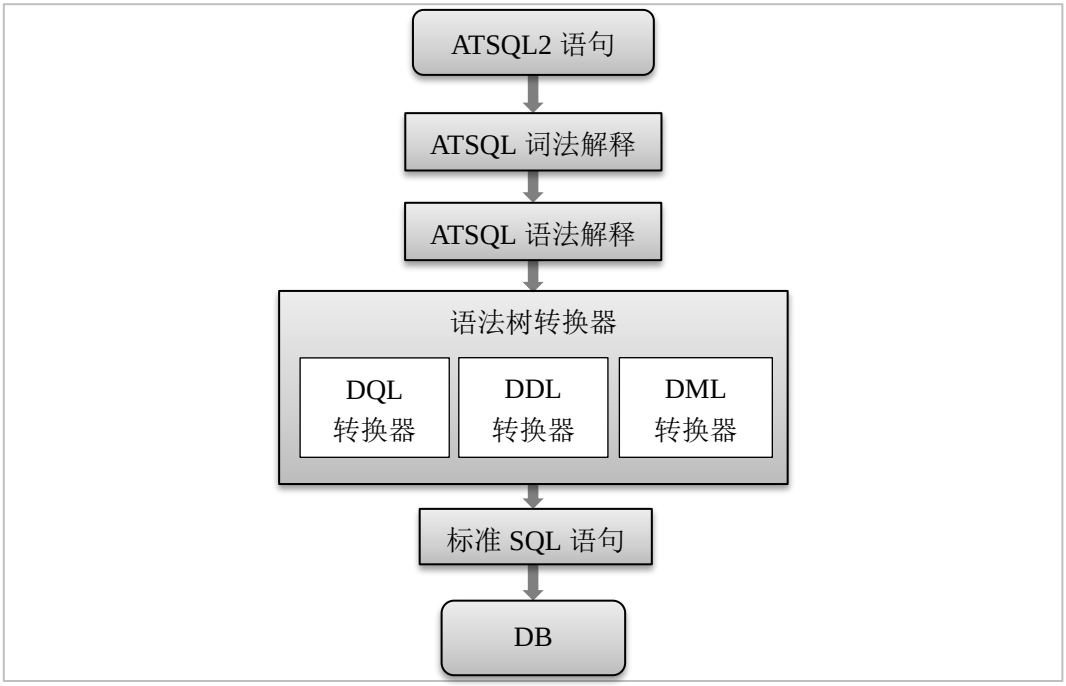


图 4-2 ATSQL 与 SQL 的转换流程图

4.2.1 时态表 DDL 的转换模块

在 ATSQL2 中包含时态的定义表的语句一般有三种，一种是建立带有有效时间的时态表，第二种是建立带有事务时间的时态表，第三种是建立同时带有有效时间和事务时间的时态表。前面说过，在 TempMT_Index 等系统中都是先建立有效时间的时态表，等有效时间的时态表完善了，再考虑建立具有事务时间的时态表和双时态的事务表。

ATSQL2 带有有效时间的时态表的创建语句如下：

CREATE TABLE <tablename> (<datatype1> <columnname1> , … ,) AS VALIDTIME ;

经过语义转化为标准的 SQL 语句为：

CREATE TABLE <tablename> (<datatype1> <column_name1> , … , vts_timeDB timestamp , vte_timeDB timestamp) ;

例 4.1：带有效时间的时态表 student 创建语句

CREATE TABLE student (int id , varchar(255) name , varchar(255) email , varchar(255) course_id) as validtime;

经过语义的转换变成：

CREATE TABLE student (int id , varchar(255) name , varchar(255) email , varchar(255) course_id, vts_timeDB timestamp , vte_timeDB timestamp);

这是标准的 SQL 语句。将 as validtime 字段识别出来，在数据表中增加两列有效时间的形式，来建立时态表。这里预留增加定义时态索引的模块的语句。本论文的系统采用的是独立的操作来增加索引的。

4.2.2 时态表 DQL 的转换模块

根据对 ATSQL2 的语义解释可以知道，其分为三部分：时态标识（time flag）、SFW 语句、以及 SFW 语句之间的并交差操作（这里主要为时态连接操作）。

① 时态快照的转换模块

时态快照查询，字段为 VALIDTIME SNAPSHOT，给出所要查找一个有效

时间字段，默认不给有效时间字段为查询当前时间的记录。当无 date 字段时，时间为 now()。

ATSQL2 带有有效时间的时态表的快照查询语句：

VALIDTIME SNAPSHOT [<date>] SELECT <column_name>… FROM <tablename>;

转化为标准的 SQL 查询语句为：

SELECT distinct <column_name(noVT)>… FROM (SELECT <column_name>… FROM <tablename> WHERE vts_timeDB <= <date> AND vte_timeDB >= <date>) temptable ;

例 4.2：ATSQL2 的 student 时态表的快照查询语句

VALIDTIME SNAPSHOT '2017-03-16' SELECT * FROM <tablename>;

转化为 SQL 查询语句为：

SELECT DISTINCT sid,name,e_mail,course_id FROM (SELECT * FROM student where vts_timeDB <= '2017-03-16' AND vte_timeDB >= '2017-03-16') a ;

② 时态期间查询的转换模块

期间查询，字段为 PRIEOD，对有效时间期间做包含，相交，不相交等关系的选择运算。

ATSQL2 带有有效时间的时态表的期间查询语句：

VALIDTIME PRIEOD [DATE <begin_date> - DATE <end_date>] SELECT <column_name>… FROM <tablename> [WHERE <column_name> <operator> <value>…;

转化为标准的 SQL 查询语句为（此处为采用底层 SQL 自带的查询，不加入时态索引）：

SELECT <column_name>… FROM <tablename> [WHERE <column_name> <operator> <value>… AND vts_timeDB <= <begin_date> AND <end_date> <= vte_timeDB ;

例 4.3：有效时间的时态表的时态选择之期间查询 student 表

VALIDTIME PERIOD [DATE '2013-1-1' - DATE '2014-7-1'] SELECT * FROM student WHERE course_id = 100070;

转化后的标准 SQL 查询语句：

SELECT * FROM student WHERE course_id = 100070 AND vts_timeDB <= '2013-1-1' AND '2014-7-1' <= vte_timeDB 。查询结果如表 4-3 所示。

表 4-3 时态选择期间查询结果示例表

sid	name	e_mail	course_id	vts_timeDB	vte_timeDB
64	Ana Barnes	CarolMyers@sogou.com	100070	1986/12/23	2019/05/26
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	1991/12/17	2026/02/28
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	1994/01/09	2016/05/31
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	1998/12/23	2025/01/17
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	2001/06/23	2026/07/31
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	2004/08/14	2026/06/19
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	2005/06/24	2024/09/04
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	2005/08/14	2023/09/07
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	2008/05/16	2017/03/29
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	2009/03/29	2025/10/31
64	Ana Barnes	CarolMyers@sogou.com	100070	2012/03/28	2016/06/03

③ 时态跨度查询的转换模块

跨度查询，字段为 INTERVAL，计算有效时间开始与结束时间的差值，根据不同时间粒度来进行选择运算。

ATSQL2 带有有效时间的时态表的跨度查询语句：

VALIDTIME INTERVAL <time_granularity> <operator> <value> SELECT <column_name>… FROM <tablename> [WHERE <column_name> <operator> <value>…;

转化为标准的 SQL 查询语句为：

SELECT <column_name>… FROM <tablename> [WHERE <column_name> <operator> <value>… AND timestampdiff(vts_timeDB , vte_timeDB) <operator> <value>;

例 4.4：带有有效时间的时态表的跨度查询语句

VALIDTIME INTERVAL month > 24 SELECT * FROM student;

转化为标准的 SQL 查询语句为：

SELECT * FROM student where timestampdiff (year , vts_timeDB , vte_timeDB) > 24 。查询结果如表 4-4 所示。

④ 时态投影的转换模块

时态投影查询，字段为 PROJECTION，第一步去除有效时间字段，先对非

时态部分进行投影，将带有相同非时态部分的有效时间进行期间并集。

ATSQL2 带有有效时间的时态表的投影查询语句：

VALIDTIME PROJECTION SELECT <column_name>… FROM <tablename>;

转化为标准的 SQL 查询语句+(第三方程序处理)为：

SELECT * FROM <tablename>;

表 4-4 时态选择跨度查询示例表

sid	name	e_mail	course_id	vts_timeDB	vte_timeDB
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	2020/04/04	2023/10/03
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	2004/08/14	2026/06/19
0	Cheryl Wagner	AnnaJordan@21cn.com	100051	2019/05/25	2024/07/07
0	Cheryl Wagner	AnnaJordan@21cn.com	100051	1999/02/26	2015/06/20
1	Dan Elliott	LeonardWillis@sohu.com	100089	2011/08/29	2025/04/10
1	Dan Elliott	LeonardWillis@sohu.com	100089	1992/10/26	2000/08/28
1	Dan Elliott	LeonardWillis@sohu.com	100089	2010/08/12	2024/09/23
1	Dan Elliott	LeonardWillis@sohu.com	100089	1991/03/14	1999/12/06
1	Dan Elliott	LeonardWillis@sohu.com	100089	1986/05/28	2017/10/17

⑤ 时态连接的转换模块

时态连接查询，字段为 VALIDTIME ，主要作用于为多个表同时查询时，保持非时态字段按非时态数据库进行连接，而对于时态部分（这里是有效时间段），需要找出公有的有效时间段，即时态表 A 与时态表 B 连接查询得出的时态结果为 R（VT）=A（VT）∩B（VT）。

ATSQL2 带有有效时间的时态表的连接查询语句：

VALIDTIME SELECT <column_name>… FROM <tableAname>,<tableBname> WHERE <tableAname.tableBid> = <tableBname.id>;

转化为标准的 SQL 查询语句+(第三方程序处理)为：

SELECT <column_name>… FROM SELECT <column_name>… FROM <tableAname>,<tableBname> WHERE <tableAname.tableBid> = <tableBname.id> ;

第三方程序主要对得到的结果，对于相同的非时态部分，将时态部分调用算法 4.2 进行求时间期间的交集。

4.2.3 时态表 DML 的转换模块

DML 语句最基本的包含有三种：insert，update，delete。在 ATSQL2 中时态表的操作语句 insert，与一般数据表的插入类似，只是在插入非时态部分数据

时，同时将有效时间字段插入即可。因为对于有效时间字段，一般默认为录入系统时则为准确的时间，而且对于历史数据库而言，历史记录始终都要保存着，所以不存在对于时态数据的时态部分进行物理意义上的删除和更新。若要对时态数据的非时态部分进行更新和删除，则使用一般的非时态语句即可。

ATSQL2 带有有效时间的时态表的插入语句如下：

VALIDTIME PRIEOD [DATE <begin_date> - DATE <end_date>] INSERT INTO <tablename> VALUES (<value>…) ;

经过语义转化为标准的 SQL 语句为：

CREATE TABLE <tablename> (<datatype1> <column_name1> , … , vts_timeDB timestamp , vte_timeDB timestamp) ;

例 4.5：带有效时间的时态表 student 插入语句

VALIDTIME PRIEOD [DATE ‘2014-09-01’ – DATE ‘2017-07-01’] INSERT INTO student VALUES (‘67’ ,’Chi Xuehui’,’qazcxh@163.com’,’100070’) ;

经过语义的转换变成：

INSERT INTO student VALUES ('67' , 'Chi Xuehui' , 'qazcxh@163.com' , '100070' , '2014-09-01' , '2017-07-01') ;

这是标准的 SQL 语句。将有效时间字段放 VALUES 后面增加两列有效时间的形式，来做插入。

4.3 多线程优化的 TDIndex 查询

本节采用多线程算法对 TDIndex 索引查询时进行优化。

4.3.1 多线程优化模型

查询元素 Q，n 个线序分支，t 个线程，将 n 个线序分支分成 t 块，保证块内有序，分别对每一块调用第 3 章的时态索引 LOB 的二分包含查询算法，得到每一块的结果。最后等所有 t 块的结果得到以后，将所有的结果合并到一起。如图 4-3 所示。

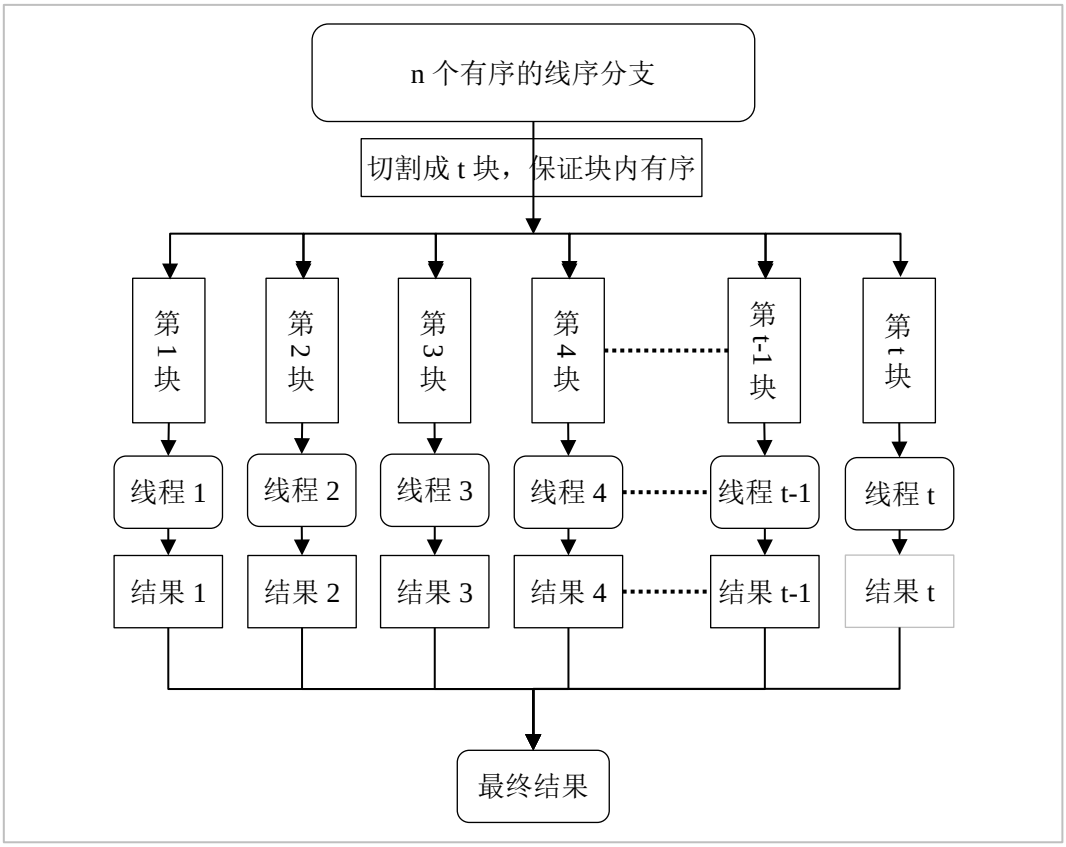


图 4-3 多线程优化模块图

4.3.2 分支策略划分

所有的计算过程都需要等待为最后完成的那个计算，即多线程查询运行的时间开销，主要取决于最慢的线程的开销，由于总体的查询算法一致，所以保证每一个线程的查询开销一致，即可以提升最慢线程的效率。对于多线程处理的问题，根据前一节的优化模型我们可以知道，每个线程里分别应该处理哪些线序分支，这是一个比较棘手的问题。这里呢，我们想出了至少四种解决方法，分别是连续划分，交叉划分，前部连续分支划分和前部交叉划分四种策略。采用策略划分完成后，对每一个线程采用基于时态索引 LOB 的二分包含查询算法，将每个线程得到的结果进行合并即可。为了方便的进行图形理解，这里的例子统一以 3 个线程，11 条分支分支为例。如图 4-4 所示。

① 连续策略划分

这是第一种，也是我们最容易想到的划分方法，平均所有的线序分支，并将相邻的线序分支放到同一个线程里。也即已知计算机开设有 t 个线程，n 个线

序分支，则前 t-1 个线程处理的分支个数为 $m_i=(n-1)/t+1$（其中 m_i 向下取整数，i 取 1 至 t-1）个，第 t 个线程处理的分支个数为 $m_t=n-(m_1x(t-1))$。则第 1 个线程处理第 1 至第 m_1 个分支，第 2 个线程处理第 m_1+1 至 m_2 个分支，以此类推，第 t 个线程处理第 m_{t-1} 至 m_t 个分支，如下图 4-5 所示。

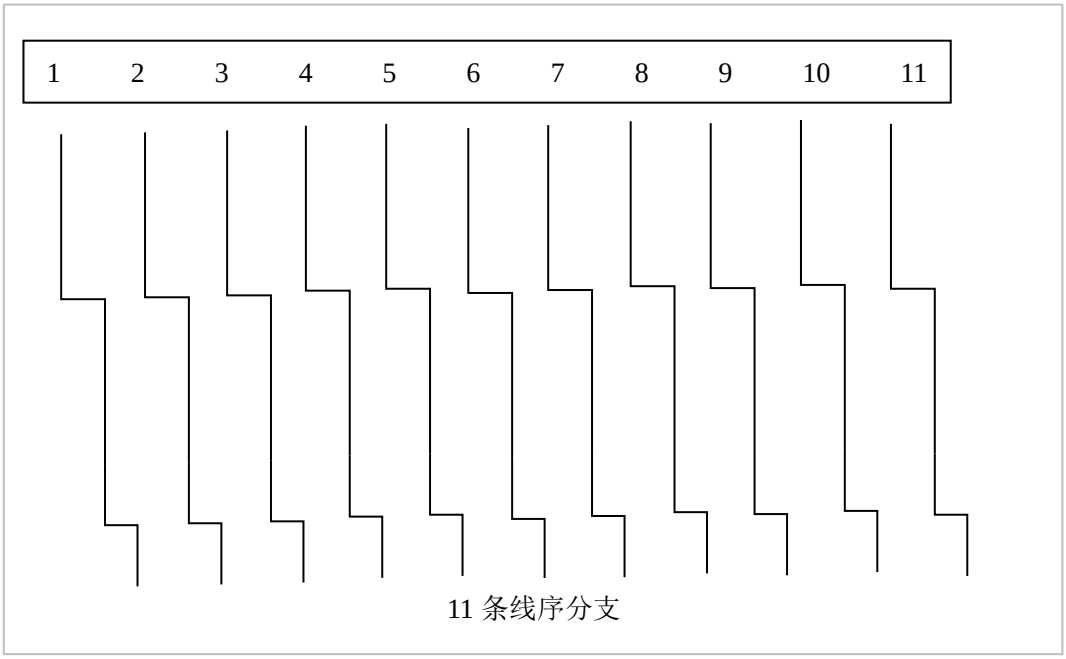


图 4-4 线序分支划分前

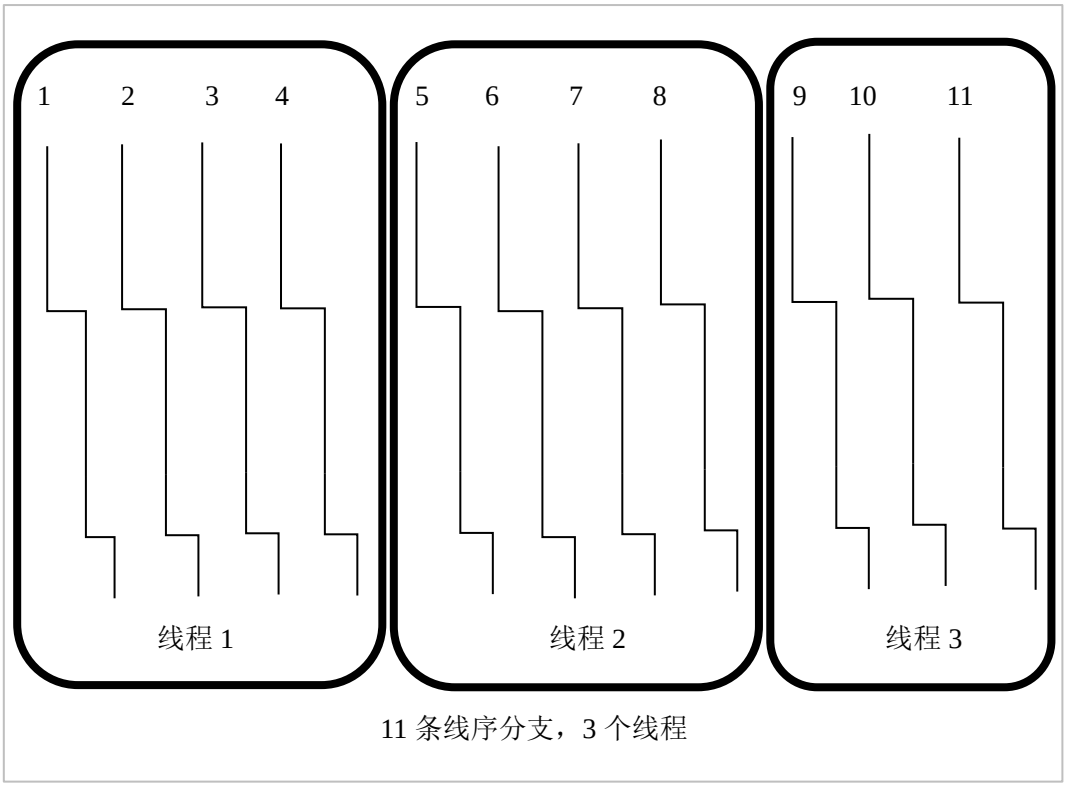


图 4-5 连续策略划分过程

② 交叉策略划分

这种划分方法和其实类似连续策略划分，与连续策略划分不同之处在于，连续策略划分采用的是将相邻的线序分支放到同一个线程里，保持连续性，而交叉策略划分的方法是，将连续的分支，顺序的放入线程中，直到最后一个线程，如此反复循环，直到最后一个线序分支放慢。也即已知计算机开设有 t 个线程，n 个线序分支，先取 y=mod(n,t)（mod 为取 n/t 的余数），则前 y（1<=y<=t）个线程处理的分支个数为 $m_i=(n-y)/t+1$（其中 i 取 1 至 y）个，第(y+1)至第 t 个线程处理的分支个数为 $m_i=(n-y)/t$（其中 i 取 y+1 至 t）。则第 i（其中 i 取 1 至 y）个线程处理线序（$\Sigma L_{((j-1)xt+i)}$（其中 j 取 1 至 y+1））分支，则第 i（其中 i 取 y+1 至 y）个线程处理线序（$\Sigma L_{((j-1)xt+i)}$（其中 j 取 1 至 y））分支，如图 4-6 所示。

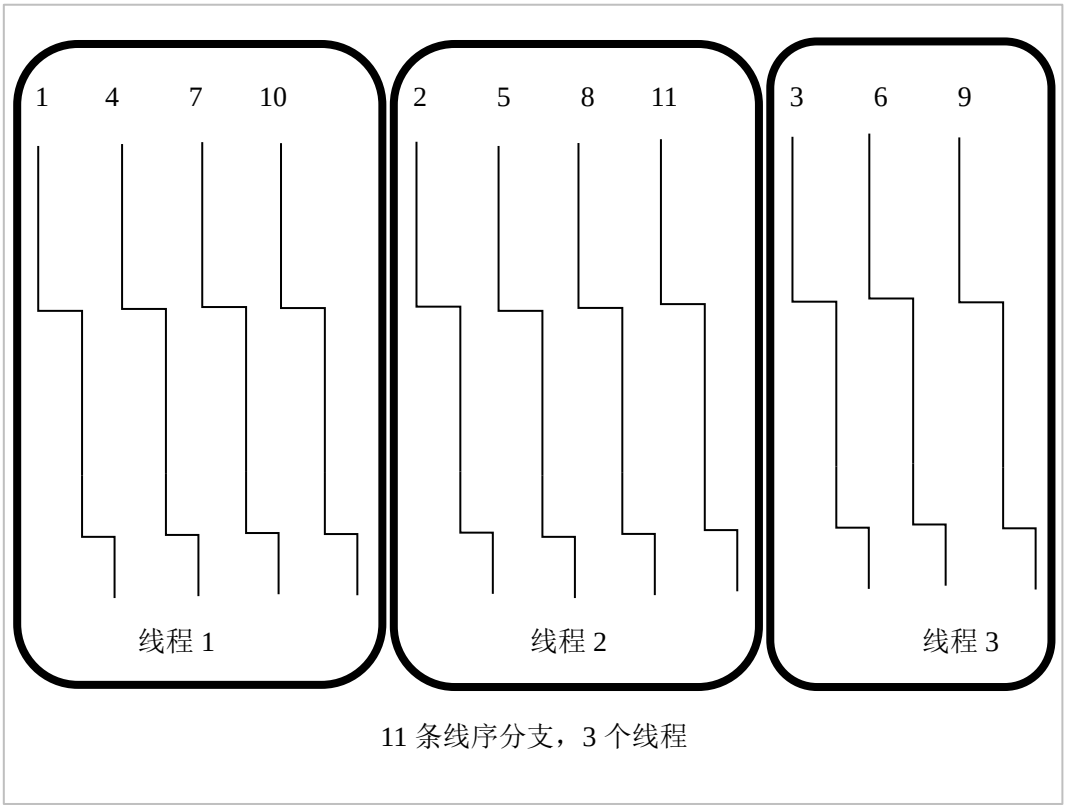


图 4-6 交叉策略划分过程

③ 前部连续策略划分

这种划分方法是先做每个一线序分支头部的比较，看查询区间是否在该分支里，当比较完所有线序分支后，将查询区间落在该分支的分支重新组合成一个组，然后对该组的查询再做线序分支内部的查询，这里的策略再采用连续策

略划分。这种划分的想法来源于，我们观测到实际有些分支不需要进行分支内部的查询，此时我们通过筛选需要的以减轻线程的开销。而实际中我们根据大量的数据知道，查询的区间往往在所有线序分支的前部，此时只需要用二分查找法快速找到前面的线序分支，即可大大提高效率。也即已知计算机开设有 t 个线程，n 个线序分支，通过查询区间的头部采用二分查找法比较得到实际需要查找的线序分支个数 N 个，N<n 且 n 个线序分支包含这 N 个线序分支，得到则前 t-1 个线程处理的分支个数为 $m_i=(N-1)/t+1$（其中 m_i 向下取整数，i 取 1 至 t-1）个，第 t 个线程处理的分支个数为 $m_t=N-(m_1x(t-1))$。则第 1 个线程处理第 1 至第 m_1 个分支，第 2 个线程处理第 m_1+1 至 m_2 个分支，以此类推，第 t 个线程处理第 m_{t-1} 至 m_t 个分支。如下图所示，这里采用二分查找法找到线序分支，1，2，3，4，5，6，7 这些分支在查询区间里，现在即可在这 7 个分支采用连续分配到每个线程中即可。如图 4-7 所示。

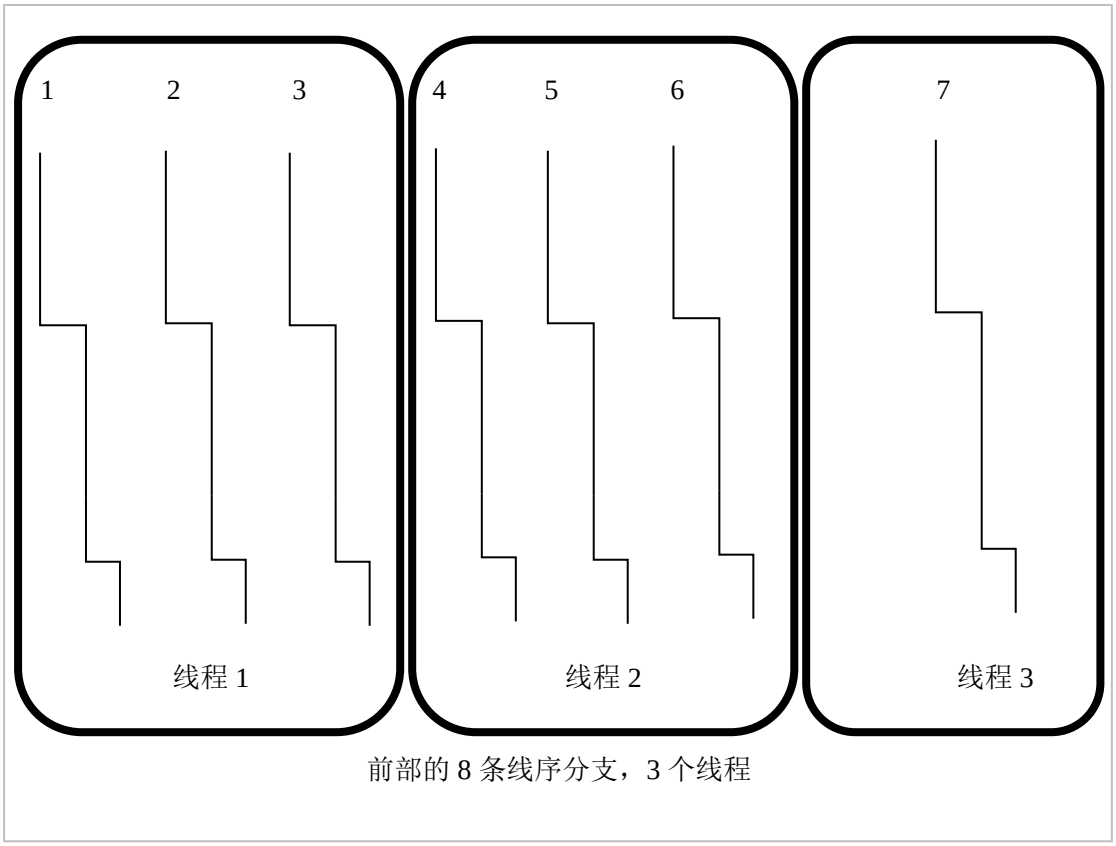


图 4-7 前部连续策略划分过程

④前部交叉策略划分

这种划分方法是先做每个一线序分支头部的比较，看查询区间是否在该分

支里，当比较完所有线序分支后，将查询区间落在该分支的分支重新组合成一个组，然后对该组的查询再做线序分支内部的查询，这里的策略再采用连续策略划分。也即已知计算机开设有 t 个线程，n 个线序分支，通过查询区间的头部采用二分查找法比较得到实际需要查找的线序分支个数 N 个，N<n 且 n 个线序分支包含这 N 个线序分支，再取 y=mod(N,t)（mod 为取 N/t 的余数），则前 y（1<=y<=t）个线程处理的分支个数为 $m_i=(N-y)/t+1$（其中 i 取 1 至 y）个，第(y+1)至第 t 个线程处理的分支个数为 $m_i=(N-y)/t$（其中 i 取 y+1 至 t）。则第 i（其中 i 取 1 至 y）个线程处理线序（$\Sigma L_{((j-1)xt+i)}$（其中 j 取 1 至 y+1））分支，则第 i（其中 i 取 y+1 至 y）个线程处理线序（$\Sigma L_{((j-1)xt+i)}$（其中 j 取 1 至 y））分支。如下图所示，这里采用二分查找法找到线序分支，1，2，3，4，5，6，7 这些分支在查询区间里，现在即可在这 7 个分支采用交叉分配到每个线程中，即可。如图 4-8 所示。

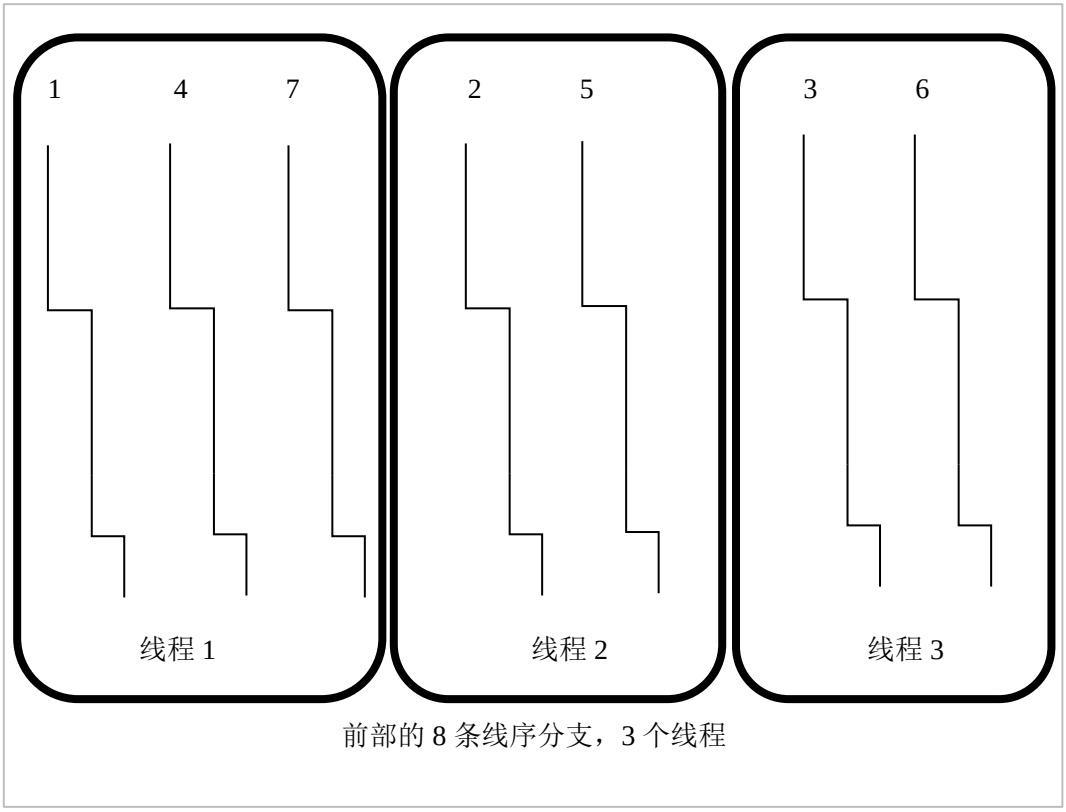


图 4-8 前部交叉策略划分过程

该方案，使得不同线程之间能做到较大的负载均衡，能将现有的 TDIndex 索引的查询运用到现有计算机环境中，大大提高查询效率。

4.4 本章小结

对于 TempMT_Index 中的时态关系进行了深入研究具有理论意义。本章中本论文研究的重点在于 TDIndex 索引的多线程优化的查询方案，并创新性的提出了四种分支策略优化方案，其中前部交叉策略划分是对于多线程目前的最优方案，具有重大意义。

第5章 TempMT_Index 系统结构

TempMT_Index 系统是在 tempDB 基础上建立起来的系统。其主要特点包括，采用 B/S 架构搭建的 Web 系统，便于研究者共享交流。更加轻量级，去除与时态数据不相关模块，增加效率，提高了准确性。服务器环境：Windows，Tomcat7.0，Mysql5.6，JDK1.8。

5.1 典型模块的设计与实现

Web 系统采用逻辑业务、控制、界面显示分离的方式组织，便于系统算法业务方面的修改，也便于系统功能扩展与移植等。如图 5-1 所示。

5.1.1 时态索引相关算法模块

包含有时态索引构建算法模块，采用时态索引的期间包含查询算法模块，采用多线程优化的期间包含查询算法模块（含四种分支策略）。 具体见第 3 章和第 4 章。

5.1.2 时态数据平台相关算法模块

时态投影查询算法模块，快照查询算法模块，时态连接查询算法模块，时态选择跨度查询算法模块，ATSQL2 中间件模块，时态数据元组模型，提示信息模块。具体见第 4 章。

5.1.3 期间选择策略控制模块

采用二进制标志位方式，设置参数，树的结点有两个孩子从左至右分别为 0，1，三个孩子时从左至右为 00，01，10。对于包含查询，共 8 种策略采用 4 位二进制表示。可知一次性采用 SQL 查的策略标志为 0000，高位为 1 时是采用了时态索引，高位为 10 采用单线程查，其中分支内部遍历标志为 1000，分支内部二分法为 1001，分支内部起始序列法为 1010，高位为 11 采用多线程策

略，其中连续策略为 1100，交叉策略为 1101，前部连续策略为 1110，前部交叉策略为 1111。如图 5-2 所示。

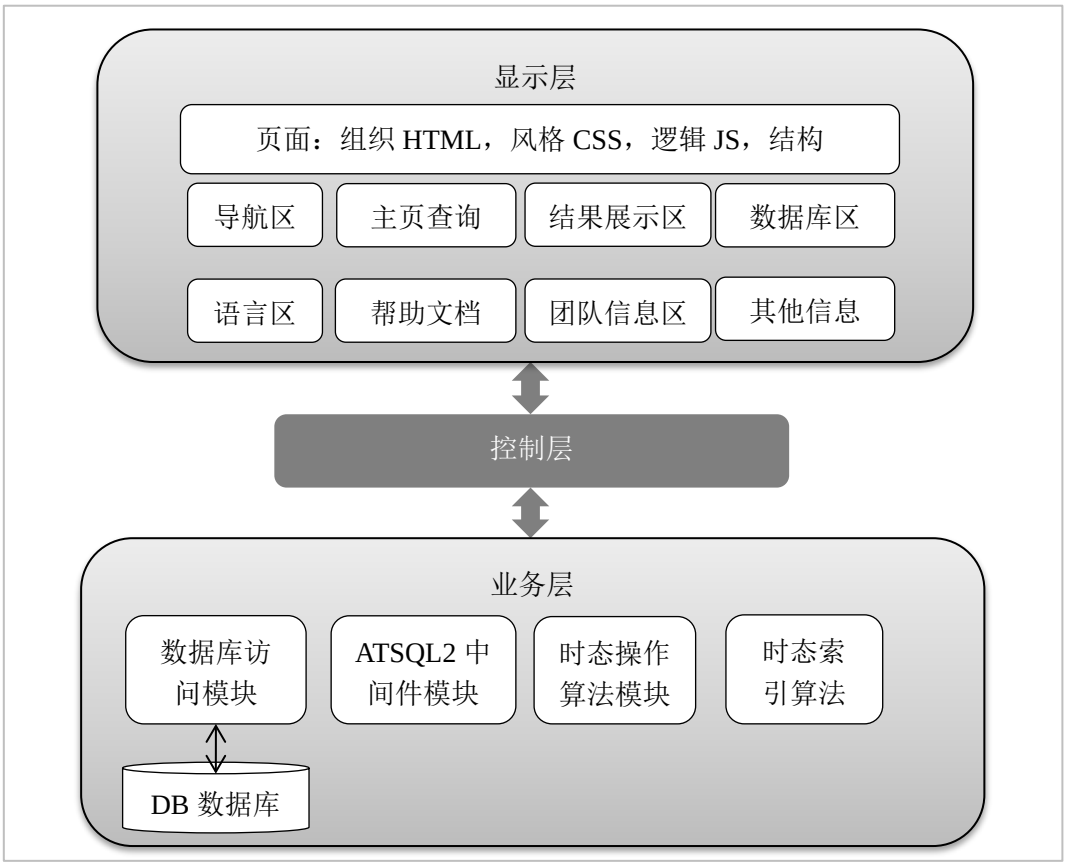


图 5-1 系统模块图

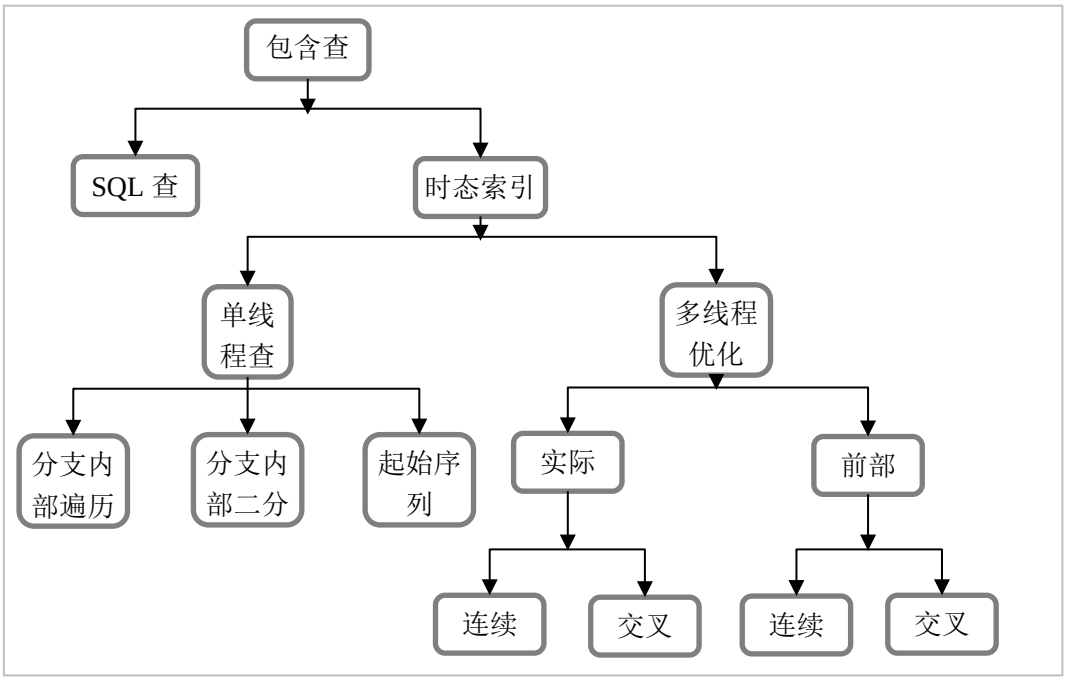


图 5-2 期间选择策略控制图

5.1.4 磁盘文件索引模块

为了便于索引能够及时方便的读取，在磁盘内以文件格式存放已经建立好的索引。系统中索引来自于磁盘文件，将上层索引采用单一文件以一定的格式存储，对每一个线序分支的数据都建立单独的文件进行存放。当进行查询时先检索上层结点的单一文件，对于线序分支内部需要进一步检索的结果，再读取对应的单独数据文件，由于文件独立存放和查询的部分性，可以减少访问磁盘的 IO 次数，减少查询时间开销，进而增加了查询效率，如图 5-3 所示。

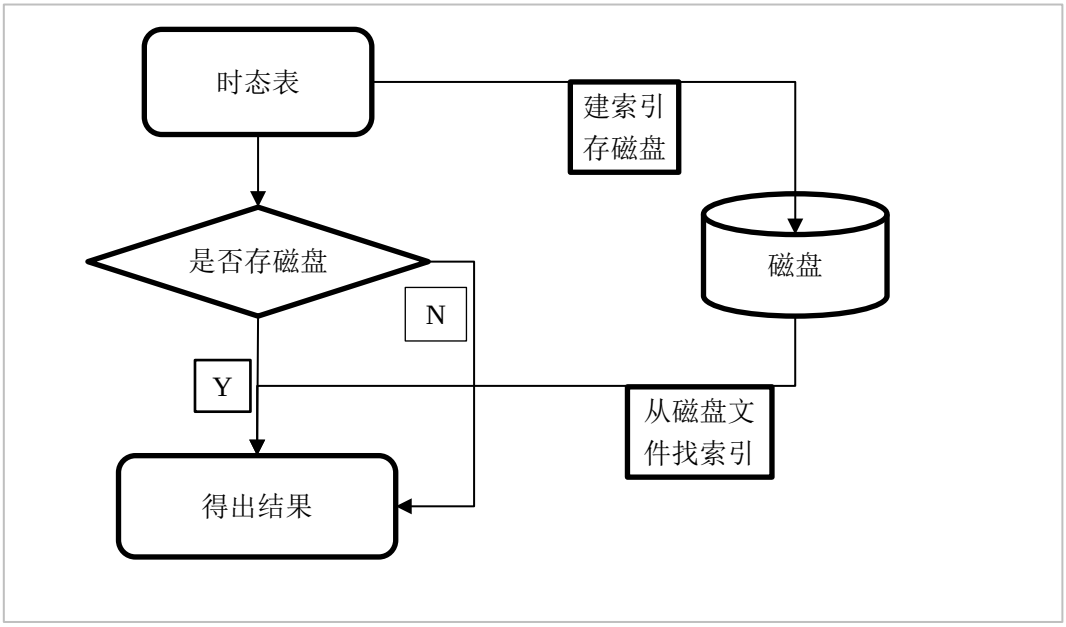


图 5-3 磁盘存放时态索引模块

5.1.5 底层数据库连接模块

采用 JDBC 包，底层数据库为 Mysql。通过 JAVA 的类加载器读取 mysql.properties 中的连接信息等，通过 Class.forName(driver)加载驱动，获取 Connection 连接，调用其 PreparedStatement 加载 SQL 语句，并进行 query，update 等方法的封装，其中 update 方法支持事务性，对异常信息做好抛出处理，并对异常与语句执行完毕后合理调用 close()方法，形成单例模式的数据访问接口 IDAO，统一对所有连接数据库的入口。

5.1.6 单页页面显示模块

为便于展示方便，系统采用单页下拉页面，便于更好的交互学习，也便于操作者便捷使用。页面内容主要分为八个区块：导航区，主页查询区，结果展示区，数据库区，语言区，帮助文档区，团队信息区，其他信息区。

导航区：系统中的展示页面的导航，主要包括 logo，系统名称，可导航至主页查询区，数据库区，语言区，帮助文档区，团队信息区等。起到单页面的页面定位作用。

主页查询区：系统中最重要的部分，包含有 logo，系统名称，查询输入框，查询策略选项，内置 ATSQL 语句块，磁盘存放索引选项等，还包括有外部链接和分享链接，便于系统共享。是系统中最核心的操作区域。

结果展示区：系统中结果回显模块调用的页面，包括提示信息与查询结果信息的展示。

数据库区：查看当前数据库的状态信息。

语言区：语言模块，和一句话模块增强系统的生动性，提供优质内容。

帮助文档区：系统中对于了解系统的重要资料部分。包含与时态索引相关的文献论文，ATSQL2 语句的介绍，时态数据平台的其他系统下载，实例数据库下载，本系统的部署文档等。便于系统共享交流。

团队信息区：介绍本系统相关理论和相关实践工作等的人员信息的情况。

其他信息区：系统的描述信息与页面相关的信息。

5.1.7 结果回显模块

结果回显模块调用包括提示信息和查询结果的回显。提示信息对于 DDL，DQL，DML 等语句执行都会有回显，而查询结果只针对 DQL 的执行后的结果展示，采用 table 表格分页的形式。当 html 提交 ATSQL2 语句后，通过调用 AJAX 异步更新模块发送给指定的语句 Servlet， 语句 Servlet 接收请求和请求的参数，将过程交给业务逻辑层的服务处理，服务根据参数调用对应的 DQL，DDL，DML 中间件和对应的业务查询功能得到结果和提示信息，该结果和提示信息又返回给语句 Servlet，语句 Servlet 将结果返回给 AJAX 的回应模块，通过

JS 采用分页的格式，将数据推送到 html 页面回显结果区域。执行流程也可如图 5-4 所示。

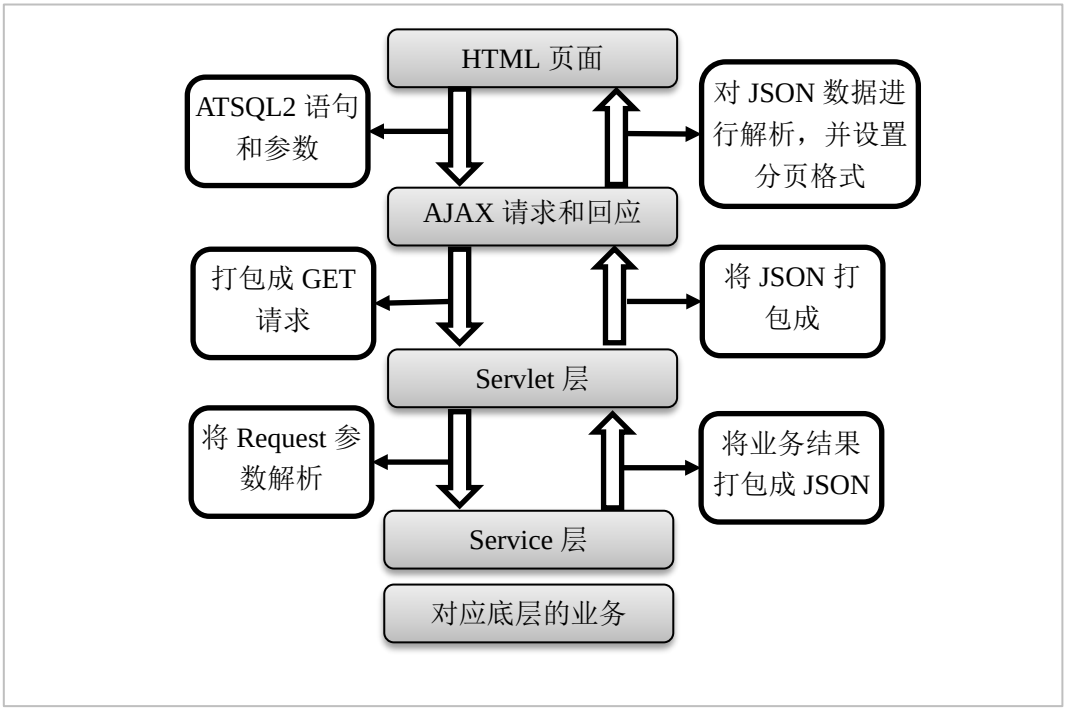


图 5-4 结果回显提示模块流程图

5.2 详细设计

数据库访问层 DAO：dao 包，配置文件目录：conf/。

业务层：service 包，所有的业务逻辑。

控制层：web.action 包，WebRoot/WEB-INF/web.xml。

视图层：在 WebRoot/目录下。

日志信息：log/。

实验仿真：实验逻辑：lab，实验数据：lab/。

如图 5-5 所示。

数据库访问层

IDAO：定义了 Dao 层可进行的操作，可扩展到多数据库平台。

MysqlDAO：IDAO 的 Mysql 数据库实现。

DAOFactory：getInstance()方法根据传入的 Dao 名，创建一个实现了 IDao 接口的实例。如图 5-6 所示。

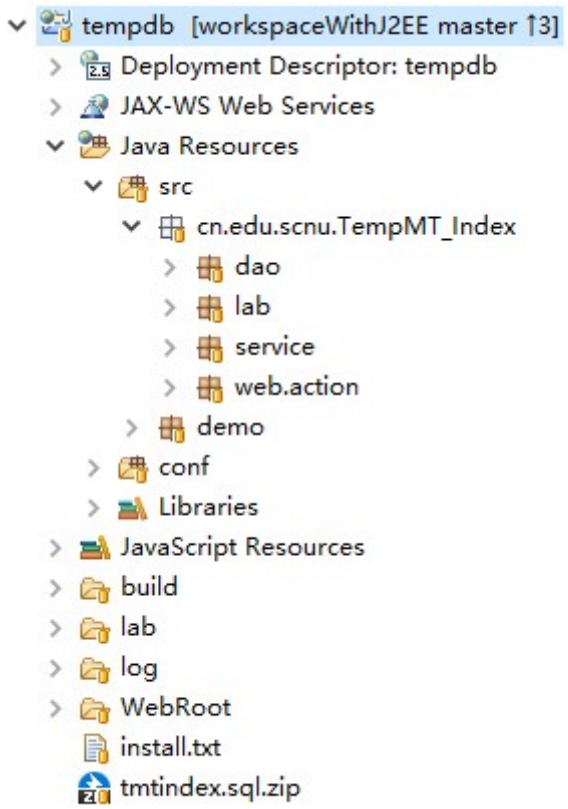


图 5-5 系统框架结构图

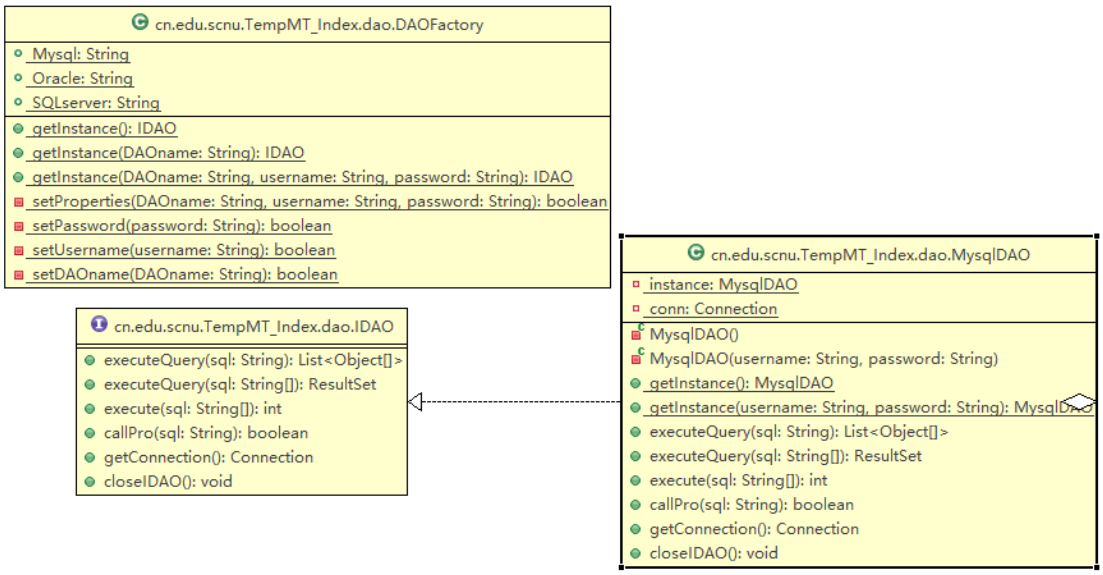


图 5-6 数据访问层及结构图

业务层设计包含 DDL,DQL,DML 的中间件，如图 5-7 所示。包含数据本体与时间标签，如图 5-8 所示。包含时态索引算法设计（含多线程），如图 5-9 所示。

视图层设计，为页面展示，包括 index.jsp，WEB-INF/MainFrame.html，js/，css/，img/，help/，font/，doc/，如图 5-10 所示。

详细的其他内容见附录开发文档。

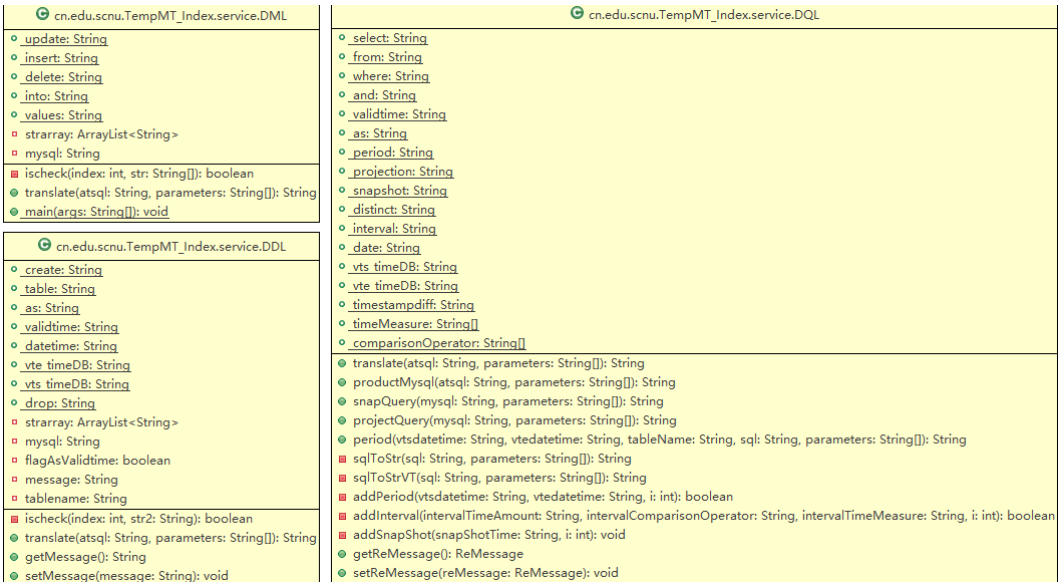


图 5-7 时态语句中间件结构图

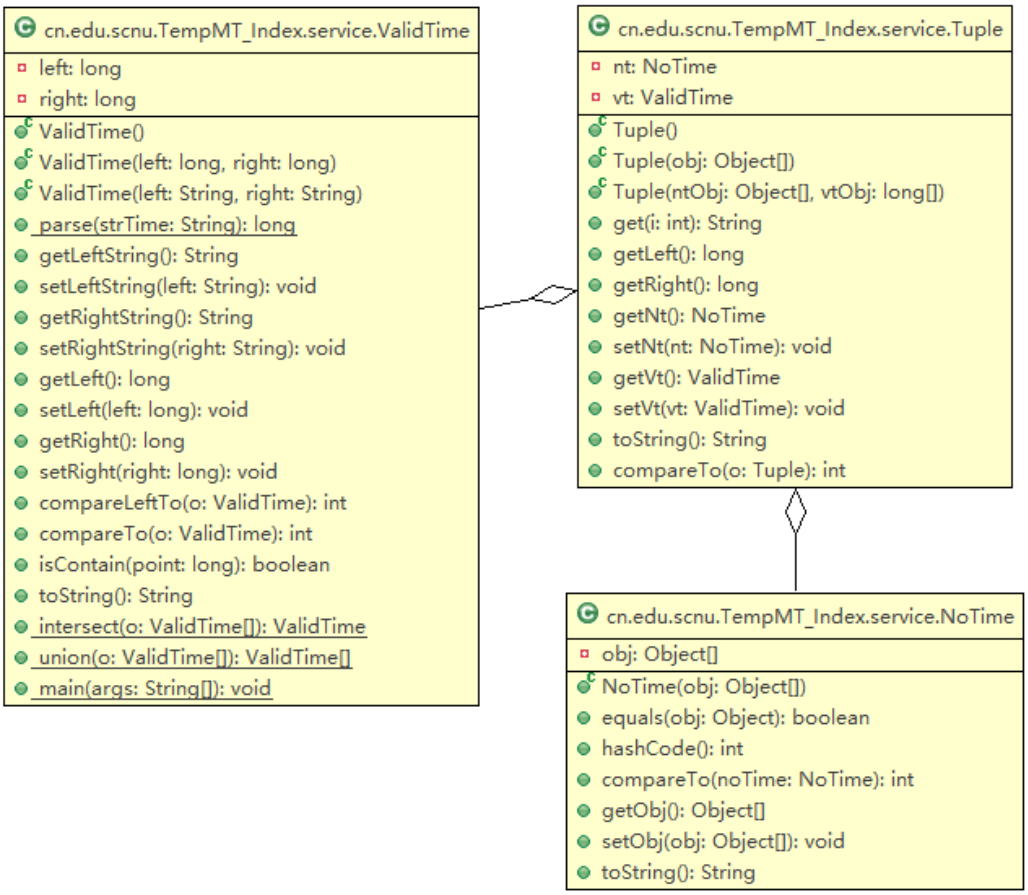


图 5-8 数据本体与时间标签

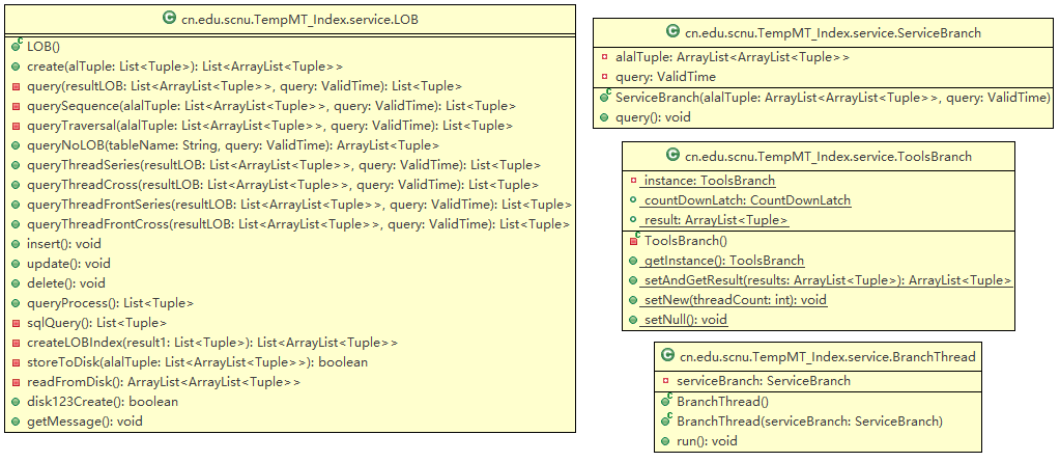


图 5-9 时态索引算法图

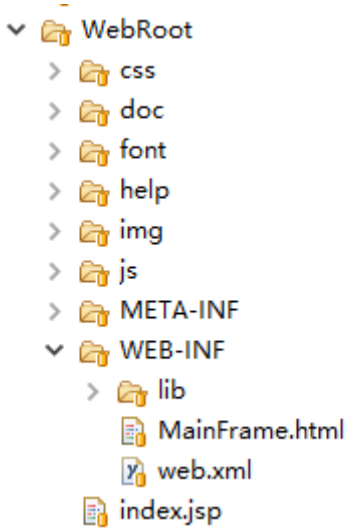


图 5-10 视图层结构图

5.3 系统编译及调试

为保证系统的可靠性与稳定，对系统进行用例的整体测试和分模块测试。

①系统整体调试

基本测试用例，student 数据实例包含 1000 条记录，course 表，teacher 表等对正确查询语句进行不同操作的情形测试。对错误查询语句进行测试。大数据量测试用例，student100w 数据表包含 1000000 条记录，测试系统的运行效率

②模块调试

时态索引构建测试，实例数据 1000 条，对算法 bug 的测试。实例数据 100w 条，对算法效率上的测试，并对内存进行对应的优化。时态查询测试，实

例数据 1000 条，对算法 bug 测试。实例数据 100w 条，对算法性能上的测试。对策略选择条件的所有满足条件方案的测试，修复针对出错情况的提示和修复结果显示不正确的情况。对 JSON 数据传输的测试，实例数据 100w，修复 JSON 传输数据量过大，服务器不响应的问题。

③BUG 修复

修复以下类型 bug，空指针异常，代码对有可能产生空指针的模块进行判定。类找不到异常。内存泄露，对内存合理的分配，在方法中合理调理各变量和对象的生命周期。SQL 语句异常，对字段严格控制，并增加判断条件。

5.4 运行以及案例

通过系统页面展示与一个查询实例，来查看系统。

5.4.1 系统页面展示

TempMT_Index 系统首页，包含系统 logo、标题、导航栏、背景等，其他所有模块如图 5-11 至图 5-18 所示。

查询结果显示区

sid	name	e_mail	course_id	vts_timeDB	vte_timeDB
56	Kyle Gomez	MarvinMurphy@163.net	100063	1990-01-04 13:38:39.0	2019-08-05 08:56:17.0
56	Kyle Gomez	MarvinMurphy@163.net	100063	1985-12-28 21:04:18.0	2018-04-20 16:23:22.0
56	Kyle Gomez	MarvinMurphy@163.net	100063	1990-04-23 15:00:09.0	2016-05-10 02:19:19.0
56	Kyle Gomez	MarvinMurphy@163.net	100063	1984-01-08 18:03:29.0	2021-05-06 09:08:12.0
56	Kyle Gomez	MarvinMurphy@163.net	100063	1993-05-09 20:04:29.0	2022-05-09 06:17:31.0
57	Diana Stevens	BradleyDuncan@msn.com	100075	1995-08-12 05:38:46.0	2024-03-03 11:16:49.0
57	Diana Stevens	BradleyDuncan@msn.com	100005	1993-04-16 01:15:56.0	2025-10-27 14:14:33.0
57	Diana Stevens	BradleyDuncan@msn.com	100005	1986-08-17 19:45:12.0	2021-08-10 17:58:27.0
59	Mike Martinez	ValerieNichols@163.com	100010	1992-04-29 00:08:55.0	2023-02-05 19:44:05.0
60	Sherry Griffin	DerekAdams@msn.com	100087	1991-08-27 04:53:32.0	2019-11-15 18:01:17.0

首页 上一页 11 12 13 下一页 尾页

图 5-11 时态选择跨度查询结果图

图 5-12 主页功能区页面展示图

图 5-13 内置语句选项效果图

图 5-14 索引类型选项效果图

图 5-15 数据库信息展示效果图

图 5-16 团队信息效果图

查询结果显示区

sid	name	e_mail	course_id	vts_timeDB	vte_timeDB
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	2005-06-24 03:23:33.0	2024-09-04 15:45:43.0
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	2008-05-16 11:24:36.0	2017-03-29 05:48:35.0
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	2009-03-29 00:23:29.0	2025-10-31 10:05:23.0
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	1998-12-23 11:44:11.0	2025-01-17 04:44:51.0
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	1991-12-17 03:46:44.0	2026-02-28 18:44:29.0
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	1994-01-09 15:35:23.0	2016-05-31 14:54:01.0
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	2005-08-14 04:40:21.0	2023-09-07 20:53:55.0
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	2001-06-23 20:19:03.0	2026-07-31 10:46:15.0
0	Cheryl Wagner	AnnaJordan@21cn.com	100070	2004-08-14 11:59:15.0	2026-06-19 16:32:02.0
64	Ana Barnes	CarolMyers@sogou.com	100070	2012-03-28 09:56:12.0	2016-06-03 18:40:41.0

首页 1 2 下一页 尾页

图 5-17 时态选择期间包含查询结果图

5.4.2 多线程查询实例

本系统实现了多种方案的包含查询的选项，进行多线程查询的实例展示。

①输入 ATSQL 格式的时态语句，如图 5-18 所示，或者选取预定时态语句，如图 5-19 所示。

②在索引类型框中选择多线程方式，这里采用的是最优的划分策略方案-多线程的前部交叉策略，如图 5-20 所示。

图 5-18 输入语句

图 5-19 选择预定语句

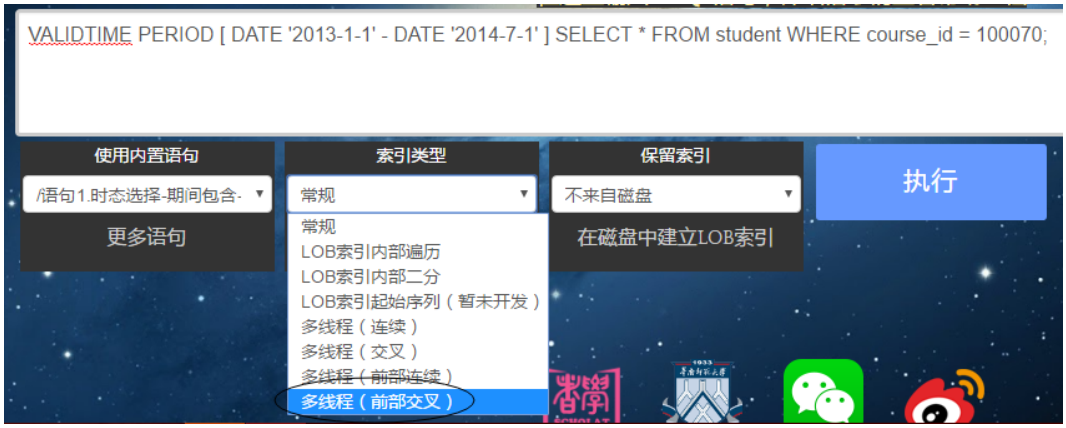


图 5-20 选择多线程策略

③点击执行按钮，如图 5-21 所示。

图 5-21 点击执行按钮

④执行查询逻辑，得到返回的提示信息，由于 student 表中的数据量为 1000 条，这里的开销小于 1ms，如图 5-22 所示。

⑤查看返回结果，以表格的形式显示，支持可翻页浏览，如图 5-23 所示。

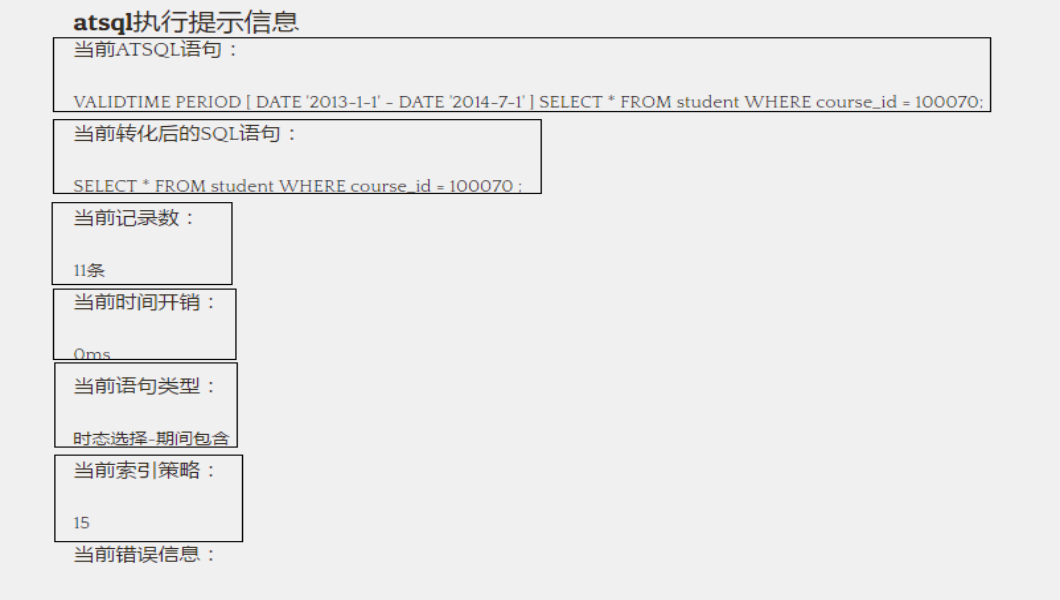


图 5-22 提示信息

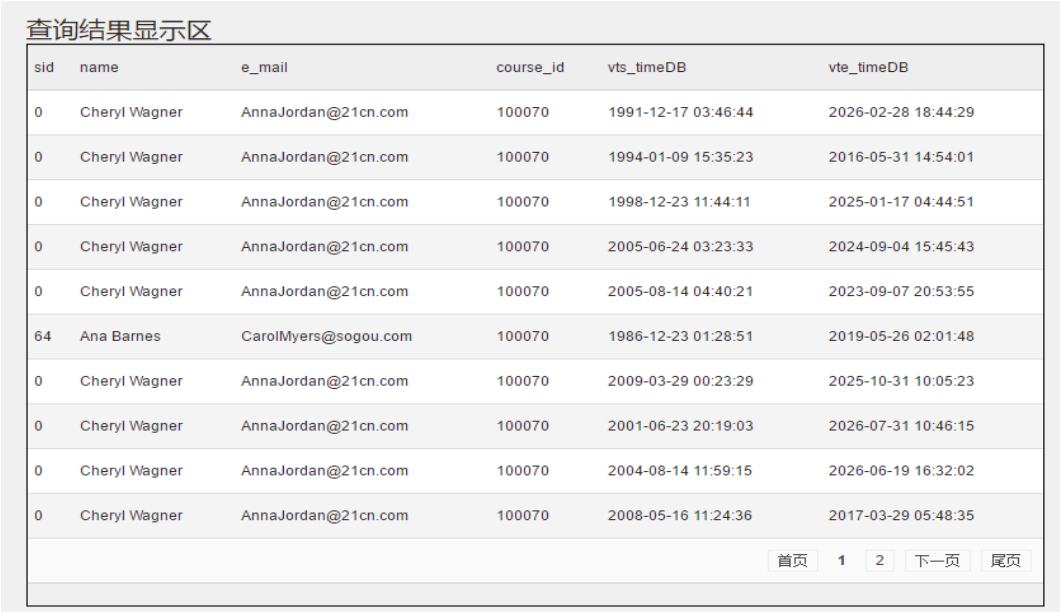


图 5-23 多线程查询结果

5.5 本章小结

本章主要搭建了 TempMT_Index 平台，通过模块的设计，系统实现，编译调试，运行案例效果来展示，既有底层业务的逻辑，在其上重点实现了多线程查询，可选择多种多线程划分策略，系统实用性强，页面简洁，适用于时态数据库的学习交流。

第6章 实验评估

本章实验机器环境如下：8 核心机器硬件环境为：CPU（AMD FX8350 (Eight Core) 4.00GHz），内存（8G），硬盘（1T）；软件环境为：操作系统（Win10），JAVA（JDK1.8），Mysql（5.6），IDE（Eclipse）。4 核心机器硬件环境为：CPU（INTEL COREI5 4300U 1.9GHz），内存（4G），硬盘（1T+128 固态）；软件环境为：操作系统（Win7），JAVA（JDK1.8），Mysql（5.6），IDE（Eclipse）。2 核心机器硬件环境为：CPU（INTEL CORE2T6400 2GHz），内存（4G），硬盘（500G+128 固态）；软件环境为：操作系统（Win10），JAVA（JDK1.8），Mysql（5.6），IDE（Eclipse）。1 核心机器硬件环境为：（在 8 核心机器采用 VMware12 的虚拟机）CPU（虚拟的 1 核心 CPU），内存（虚拟的 2G），硬盘（虚拟的 60G）；软件环境为：操作系统（Win10），JAVA（JDK1.8），Mysql（5.6），IDE（Eclipse）。默认情况下采用 8 核心机器环境。实验数据都是通过程序随机生成。以下实验用到的变量：将数据集的最大时间期间跨度记为 T，T=最大结束有效时间-最大开始有效时间，查询窗口设为 Q，按一定的跨度取查询窗口随机生成 500 组，对每一组窗口查询，取 500 次的平均开销时间。以下实验涉及到的未提及到的类型的查询都为包含查询，也就是算子 Contains（a，b）的查询。

6.1 时态查询仿真评估

时态投影运算，对现有数据做进行投影，数据量从 2w-100w。随机生成 500 个查询期间，如图 6-1 所示，横坐标为数据量，纵坐标为时间开销，很明显随着数据量的增加，时间开销曲线近似成直线，说明时态投影运算的查询对于数据量增加时表现稳定。

快照查询，对现有数据做快照查询，数据量从 2w-100w。随机生成 500 个查询期间。如图 6-2 所示，横坐标为数据量，纵坐标为时间开销，很明显随着数据量的增加，时间开销曲线近似成直线，说明快照查询对于数据量增加时表现稳定。

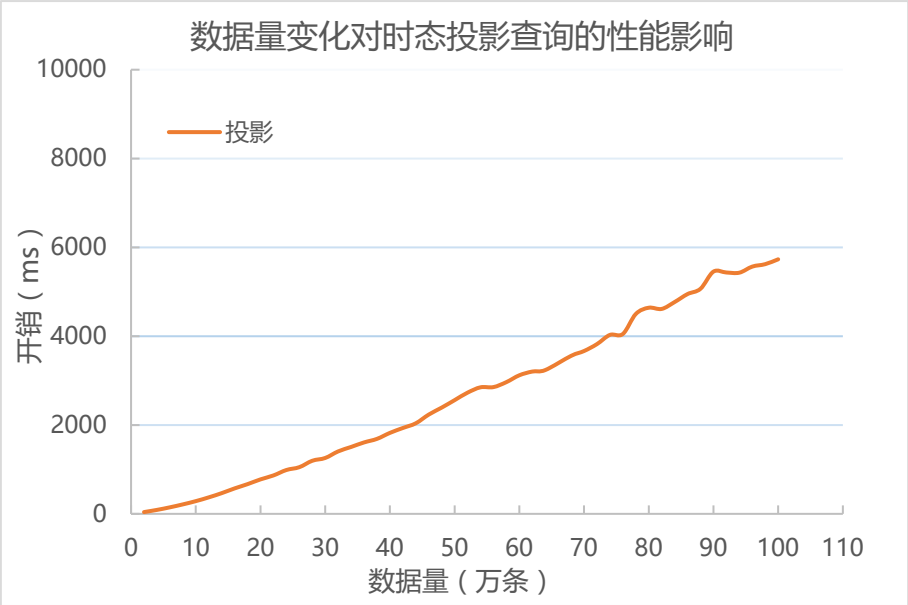


图 6-1 数据量变化对时态投影查询性能影响

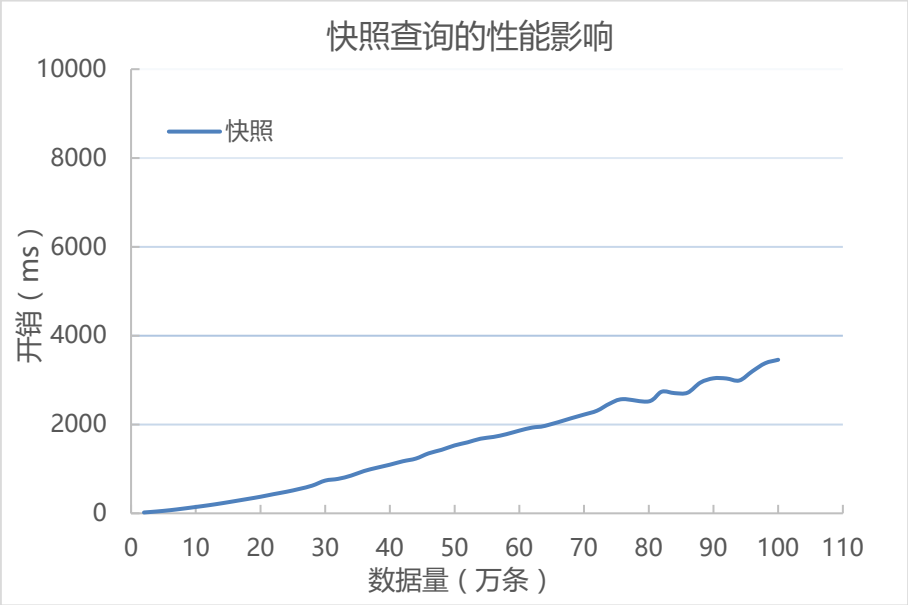


图 6-2 数据量变化对快照查询的性能影响

时态选择包含查询方法采用 TDIndex 索引，对 3.4.1 的两种查询方法与逐个遍历查询方法进行实验，该查询类型也就是算子 Contains（a，b）的查询。主要对比的是始末点序列二分的查询，与简单二分查询和逐个遍历查询的对比实验。

查询窗口跨度设为 10%T，数据量分别取 100w、200w、300w、400w、500w。计算开销的平均值进行研究。对查询的开销性能实验如图 6-3 所示，横坐标为数据量，纵坐标为时间开销。时间开销随着数据量的增长而缓慢增长，

整体近似于直线趋势。表明建立索引的开销不会因为数据量的增长，而急剧上升。在进行查询时采取两种方法，一种是基于二分的查询，很明显二分的查询开销小于逐个遍历，所以说二分查询性能高于逐个遍历。

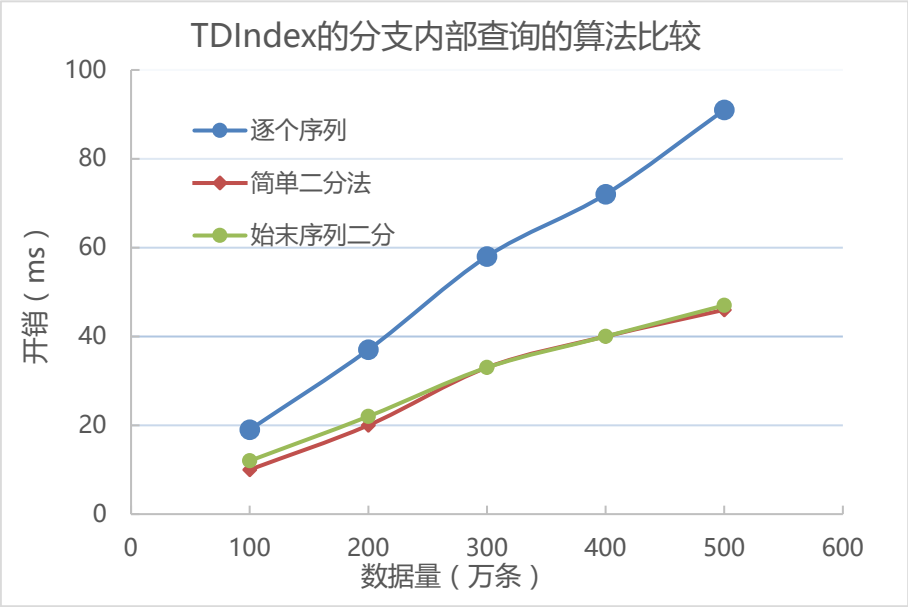


图 6-3 TDIndex 的分支内部查询的算法比较

6.2 磁盘中的 TDIndex 索引

对于 TDIndex，都是在内存中做性能的比较，本节做了将索引以文件形式分树目录存放在磁盘中，分别对时态数据索引的构建与时态数据查询两方面的实验做评估。

6.2.1 磁盘中的 TDIndex 索引构建

对于磁盘中的索引，本小节通过在磁盘中建立索引的开销，数据占磁盘空间与索引占磁盘空间，数据量与线序分支个数三个方面来做了实验评估。

将数据集的最大时间期间跨度记为 T，查询窗口设为 Q，数据量从 100w-240w 分别取 100w、120w、140w、160w、180w、200w、220w、240w。随机生成 500 组相同数据量大小的数据，建立索引并取 500 次开销的平均值进行研究。对建立索引开销的性能实验如图 6-4 所示，横坐标为数据量，纵坐标为时间开销。时间开销随着数据量的增长而缓慢增长，整体近似于直线趋势。表明建立索引的开销不会因为数据量的增长，而急剧上升。如图 6-5 所示，横坐标

为数据量，纵坐标为数据量与时间开销比。随着数据量的增长，比值呈现平缓趋势，表明建立索引的开销不会因为数据量的增长，而急剧上升。

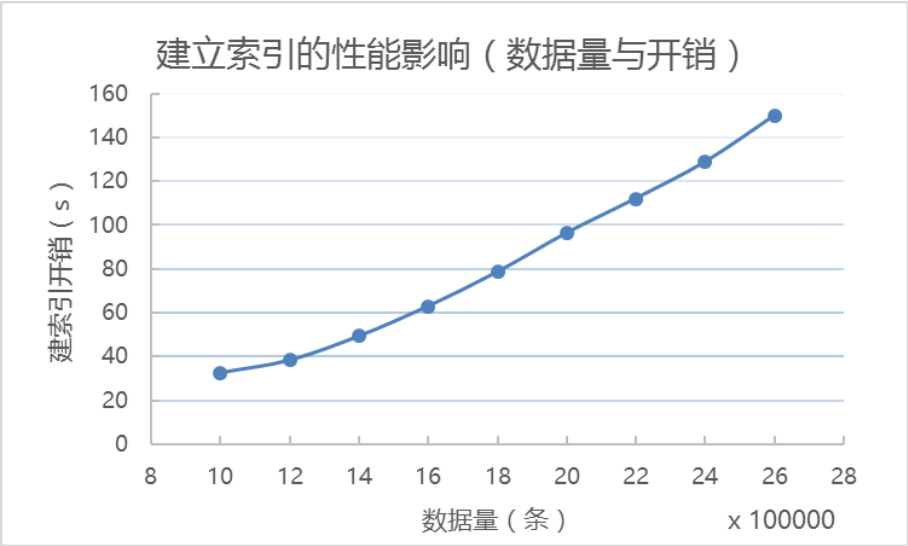


图 6-4 建立索引的性能影响（数据量与开销）

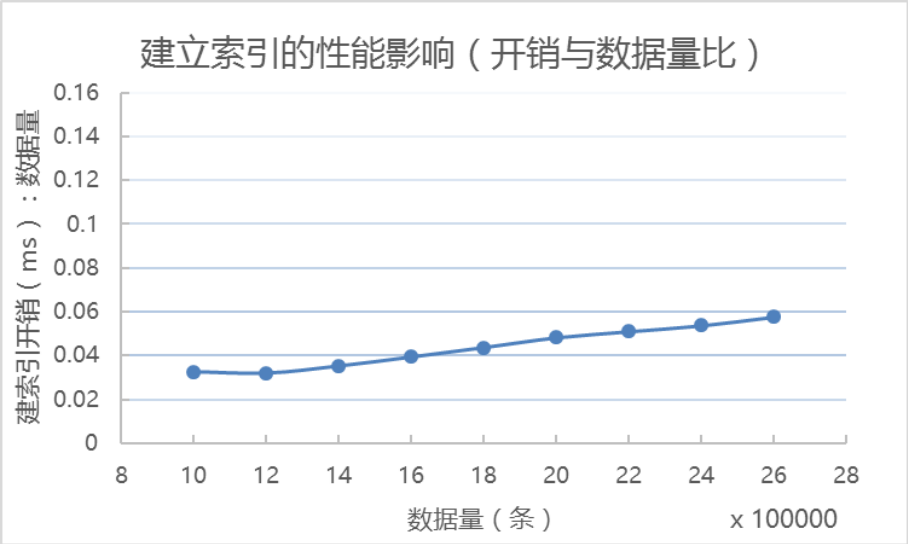


图 6-5 建立索引的性能影响（开销与数据量比）

如图 6-6 表示，横坐标为数据量，纵坐标为索引与数据量空间比，随着数据量从 100w、200w、300w、400w、500w 的递增，索引与数据量空间比逐渐减小，表明索引占用空间不会随数据量的增大而急剧增加。

如图 6-7 表示，横坐标为数据量，纵坐标为线序个数与索引比，随着数据量从 100w、200w、300w、400w、500w 的递增，TDIndex 的线序分支个数与数据量比逐渐减小，说明当数据量越大时，线序分支个数缓慢的增长，表明线序

分支个数不会随着数据量的增大而急剧增加。可以预测，数据量十分大的时候，线序分支个数会保持稳定。

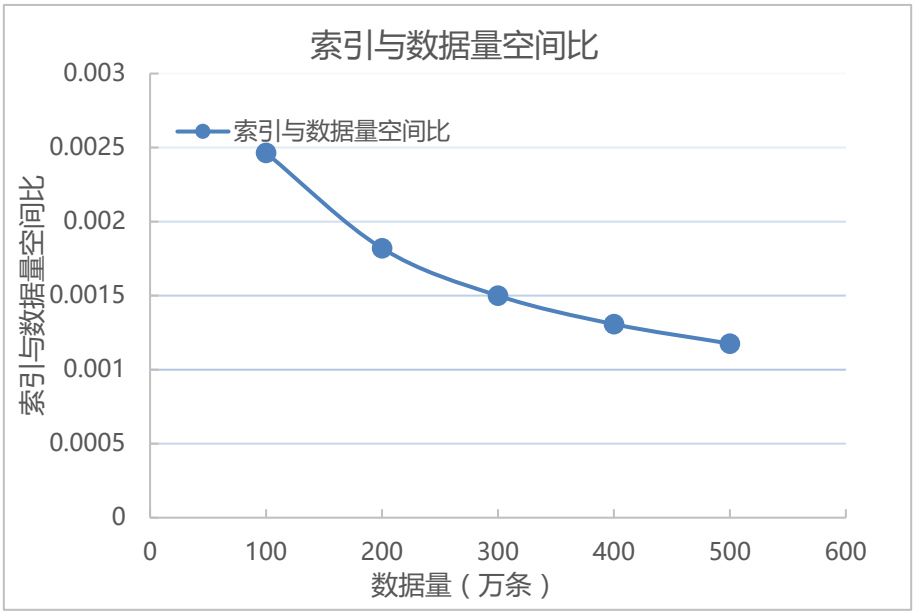


图 6-6 TDIndex 索引与数据量空间比

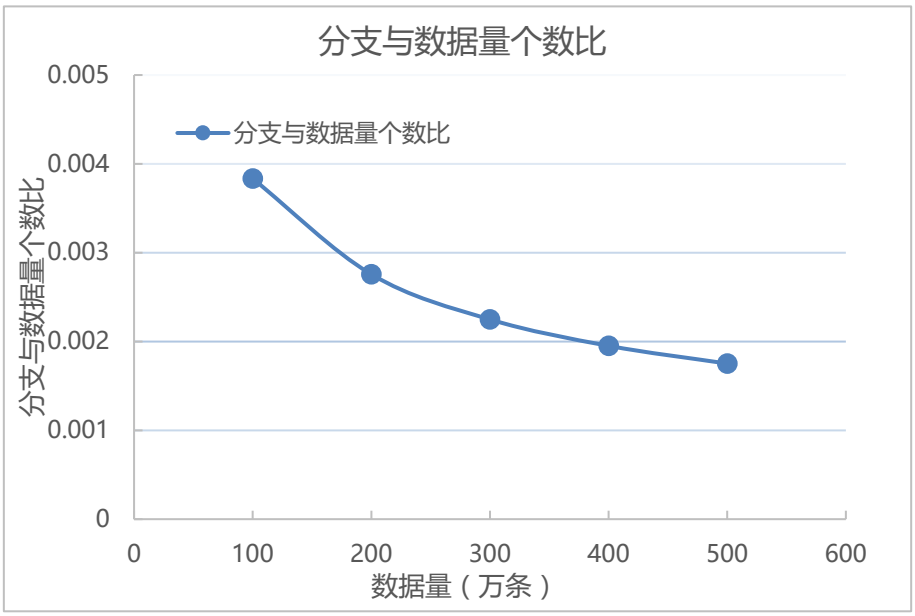


图 6-7 TDIndex 分支个数与数据量比

6.2.2 磁盘中的 TDIndex 索引查询

将 TDIndex 的索引和数据放入磁盘中，因为 Mysql 的索引和数据也放入磁盘，与之设计实验对比。数据量分别取 100w、200w、300w、400w、500w。随

机生成跨度为 10%T 的 500 组查询窗口数据，进行 500 次查询，计算开销的平均值进行比较。对查询的开销性能实验如图 6-8 所示，横坐标为数据量，纵坐标为时间开销。时间开销随着数据量的增长而缓慢增长，整体呈曲线增长趋势。表明建立索引的开销不会因为数据量的增长，而急剧上升。在进行查询时采取两种方法，一种是基于 mysql 自带索引的查询，一种是基于磁盘分文件 TDIndex 索引的查询，内部分支采用性能最差的逐个遍历算法。很明显发基于磁盘分文件 TDIndex 索引的查询开销在 100 万条时与基于 mysql 自带索引的查询开销不相上下，而随着数据量增加，基于磁盘分文件 TDIndex 索引的查询开销缓慢增长，而基于 mysql 自带索引的查询开销则快速增长，很明显得出结论基于磁盘份文件 TDIndex 索引的查询效率远快于基于 mysql 自带索引的查询效率。

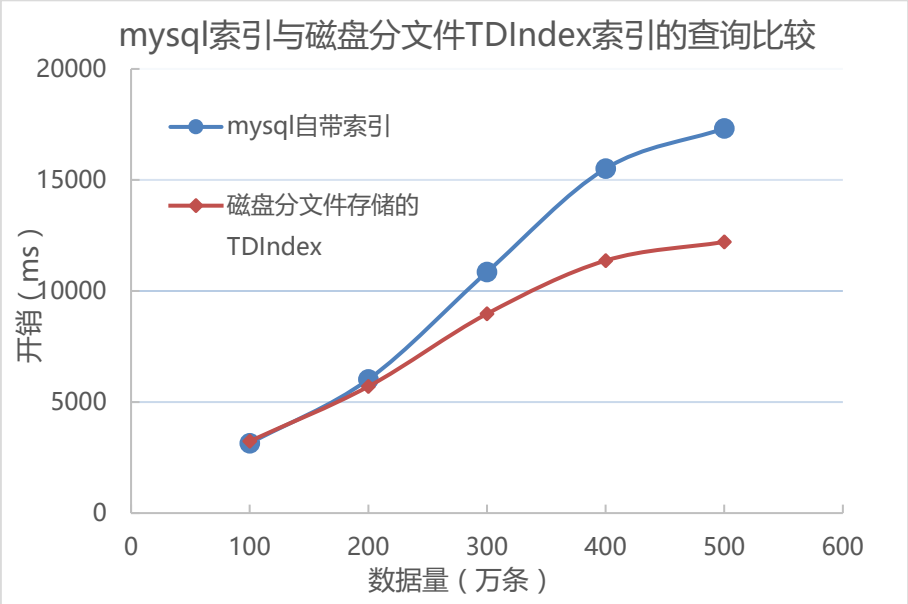


图 6-8 mysql 索引与磁盘分文件 TDIndex 索引的查询比较

将 TDIndex 的索引和数据放入磁盘中，因为 Mysql 的索引和数据也放入磁盘，与之设计实验对比。数据量取 500w。随机生成跨度分别为 T 的 10%、20%、30%、40%、50%、60%的各 500 组查询窗口数据，进行 500 次查询，计算开销的平均值进行比较。对查询的开销性能实验如图 6-9 所示，横坐标为查询窗口跨度，纵坐标为时间开销。时间开销随着查询窗口跨度的减少而缓慢增长，整体近似于直线趋势。表明建立索引的开销不会因为数据量的增长，而急剧上升。在进行查询时采取两种方法，一种是基于 mysql 自带索引的查询，一种是基于磁盘分文件 TDIndex 索引的查询，内部分支采用性能最差的逐个遍历

算法。很明显发基于磁盘分文件 TDIndex 索引的查询开销在 500 万条时与基于 mysql 自带索引的查询开销不相上下，而随着查询窗口跨度减少，基于磁盘分文件 TDIndex 索引的查询开销增长，而基于 mysql 自带索引的查询开销也增长，得出结论基于磁盘份文件 TDIndex 索引的查询效率快于基于 mysql 自带索引的查询效率。

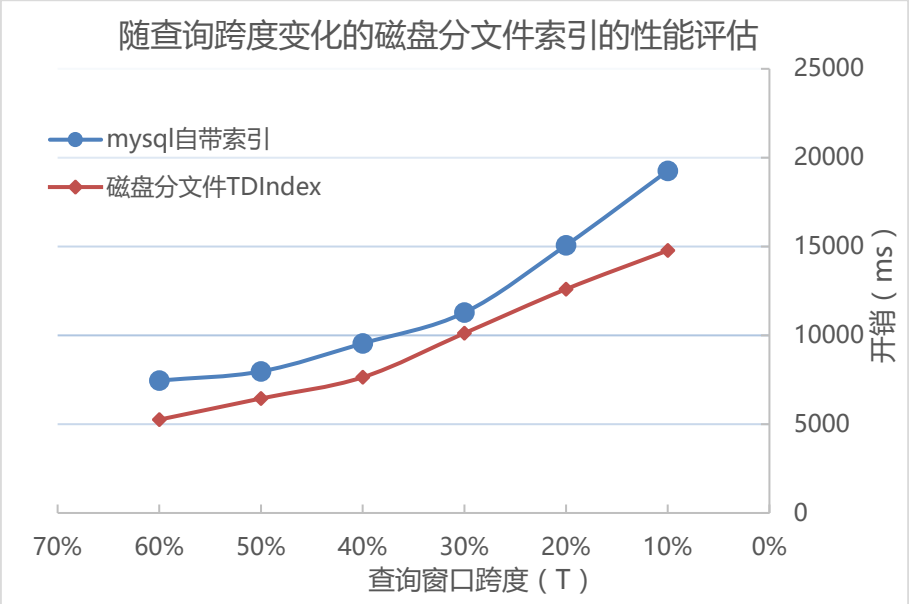


图 6-9 查询跨度变化的磁盘分文件索引的性能评估

6.3 基于多线程的 TDIndex 查询

本节提及到的实验的查询类型都为包含查询，也就是算子 Contains（a，b）的查询。

随机生成 500 个查询窗口，时间开销取 500 个查询窗口的平均值，取线程数 8 个，采用连续划分策略。随着数据量的增大，多线程优化后的 TDIndex 较单线程下 TDIndex 的查询效率更优，实验数据见图 6-10。

将数据集的最大时间期间跨度记为 T，查询窗口设为 Q，Q 的时间跨度分别取 1%T、5%T、10%T、20%T、35%T、50%T，采用连续划分策略。论文实验研究查询窗口的时间参数变化对多线程优化后的 TDIndex 索引查询性能的影响。实验数据集为 100W 数据元组，线程数仍取 8 个，对时间参数的查询性能实验如图 6-11 所示，由于多线程能对现有的线序分支高效计算，所以多线程优化后的 TDIndex 较 TDIndex 更优。

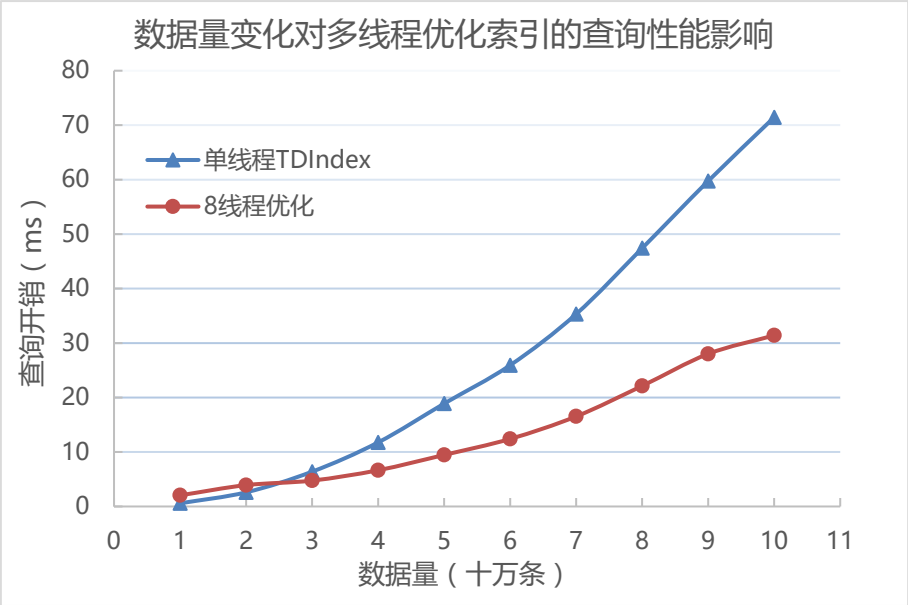


图 6-10 数据量变化对多线程优化索引的查询性能影响

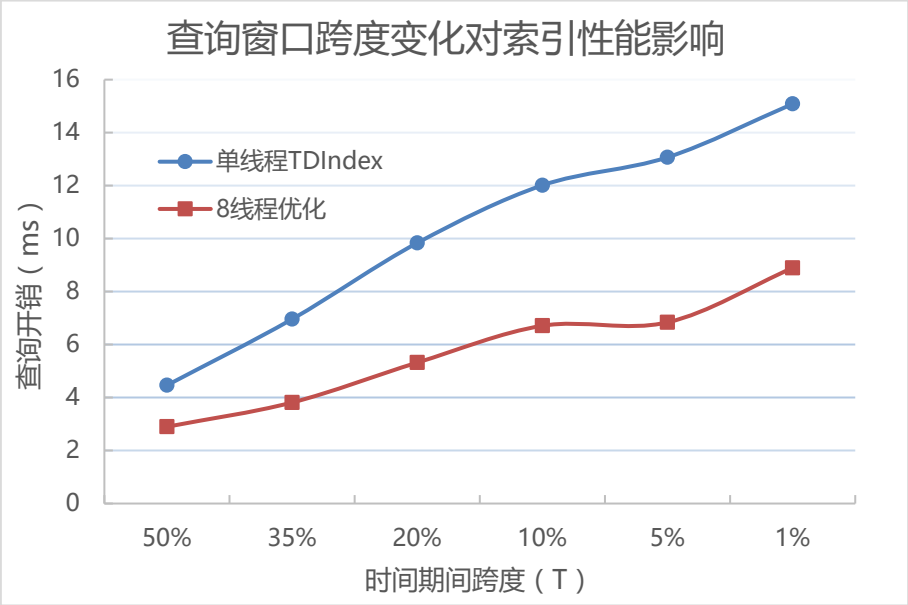


图 6-11 查询窗口变化对索引性能影响

将数据集的最大时间期间跨度记为 T，实验数据集为 100W 数据元组，查询窗口 Q 的时间跨度取 10%T，Q 随机生成 500 个。论文实验研究连续策略划分、交叉策略、前部连续策略、前部交叉策略划分对多线程优化后的 TDIndex 索引查询性能的影响。实验数据集取 100W、200W、300W、400W、500W 数据元组，线程数仍取 8 个，对划分策略的查询性能实验如图 6-12 所示，得出结论，随着数据量增大，连续策略划分与前部策略划分查询开销增长得较快，查询开销都大于交叉策略划分与前部交叉策略划分，而交叉策略划分与前部交叉

策略划分增长得较为缓慢，查询性能更优，前部交叉策略划分在数据量更大时略优于交叉策略划分。

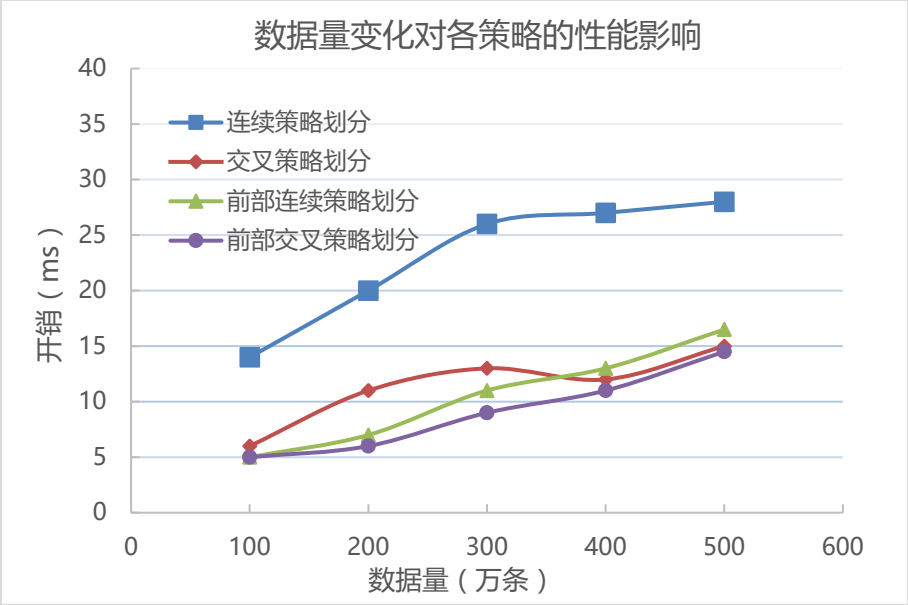


图 6-12 数据量变化对各策略的性能影响

将数据集的最大时间期间跨度记为 T，实验数据集为 500W 数据元组，查询窗口 Q 的时间跨度取 10%T、20%T、30%T、40%T、50%T、60%T，Q 随机生成 500 个。论文实验研究连续策略划分、交叉策略划分、前部连续策略划分、前部交叉策略划分对多线程优化后的 TDIndex 索引查询性能的影响。实验数据集取 500W 数据元组，线程数仍取 8 个，对划分策略的查询性能实验如图 6-13 所示，得出结论：随着查询窗口时间跨度的减小，连续策略划分与前部策略划分查询开销增长得较快，查询开销都大于交叉策略划分与前部交叉策略划分，而交叉策略划分与前部交叉策略划分增长得较为缓慢，查询性能更优，前部交叉策略划分在数据量更大时略优于交叉策略划分。

将数据集的最大时间期间跨度记为 T，实验数据集为 500W 数据元组，查询窗口 Q 的时间跨度取 10%T，Q 随机生成 500 个，线程个数分别取 1、2、3、4、5、6、7、8。论文实验研究针对不同线程个数变化的多线程加速系数比较。连续策略划分、交叉策略划分、前部连续策略划分、前部交叉策略划分对多线程优化后的 TDIndex 索引查询性能的影响。如图 6-14 所示，得出结论：随着线程个数的增加，连续策略划分与前部策略划分多线程加速比增长得较慢，查询开销都大于交叉策略划分与前部交叉策略划分，而交叉策略划分与前部交叉

策略划分增长多线程加速比增长得较快，负载均衡做得更好，查询性能更优，交叉策略划分在线程数量更多时性能更接近于前部交叉策略划分。

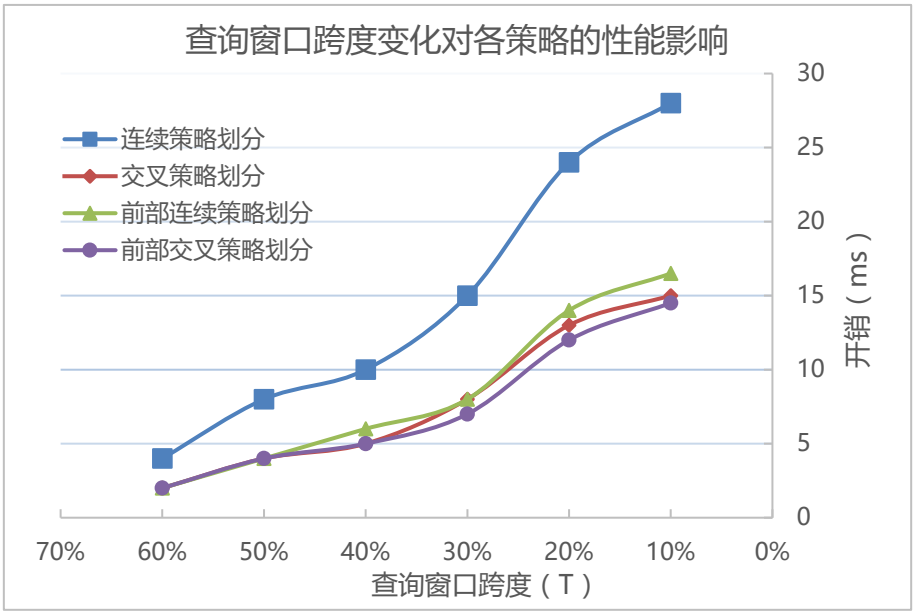


图 6-13 查询窗口跨度变化对各策略的性能影响

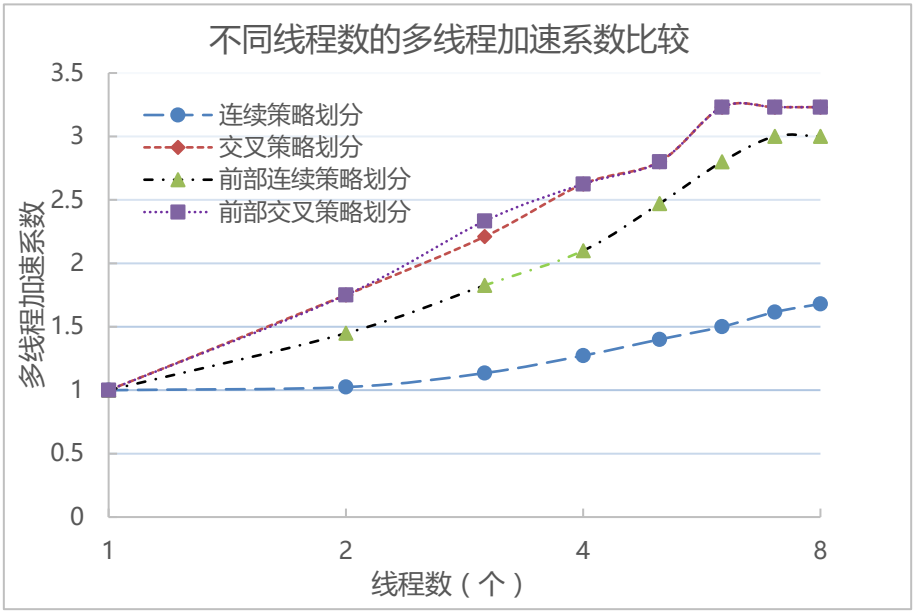


图 6-14 不同线程数的多线程加速系数比较

前面实验都使用的实验机器为 8 核心的 CPU，这里将线程数采设置为 8，对应的实验机器则分别采用 CPU 核心数为（1，2，4，8），将核心记为 CORE，使用的线程数取 8。论文实验研究线程个数变化对多线程优化后的 TDIndex 查询性能的影响。实验数据集为 100W 数据元组，查询窗口 Q 的时间跨度取 10%T，Q 随机生成 500 个，对线程个数变化的查询性能实验如图 6-15

所示，在同等线程数下 CPU 核心数越接近实际开辟线程数值时，查询性能最优。这说明当硬件条件满足时，使用上述策略开辟更多的线程，可以加快索引查询的效率。

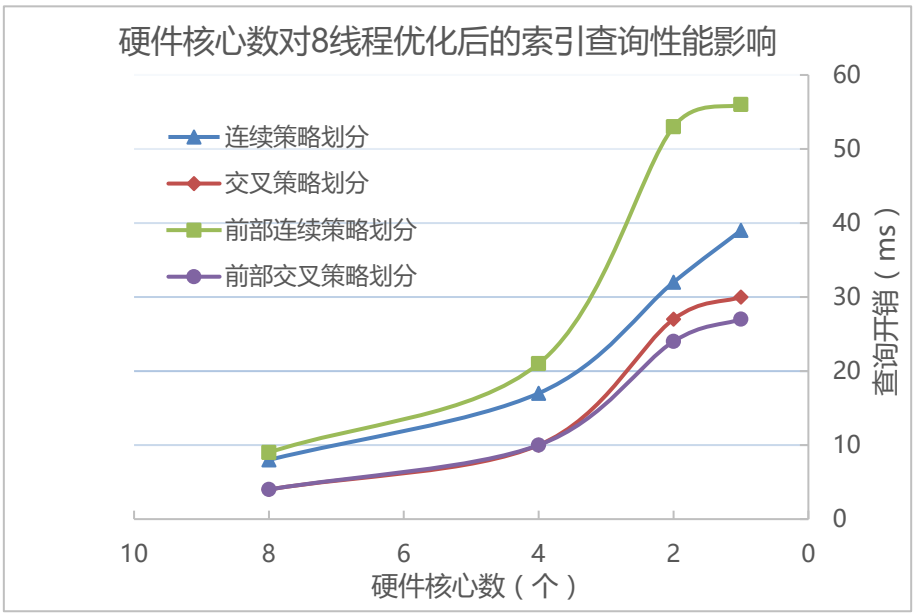


图 6-15 硬件核心数对 8 线程优化后的索引查询性能影响

前面实验都使用 8 线程数（实验机器的使用 8 核心的 CPU）来进行优化，这里将线程数采用（1，2，4，8）一一对应的实验机器采用 CPU 核心数为（1，2，4，8），将线程记为 TH，使用的线程数分别取 1TH、2TH、4TH、8TH。论文实验研究线程个数变化对多线程优化后的 TDIndex 查询性能的影响。实验数据集为 100W 数据元组，查询窗口 Q 的时间跨度取 10%T，Q 随机生成 500 个，对线程个数变化的查询性能实验如图 6-16 所示，CPU 核心数和线程个数越多，查询性能越优。这说明当硬件条件提升时，使用上述策略开辟更多的线程，可以加快索引查询的效率。

综合以上几个实验可知，前部交叉策略划分在多线程的程序中在各种环境下都是最优划分方法，适合各种环境。而对于交叉策略划分而言，在对于使用线程数越多的环境下，性能更接近前部交叉策略划分。

下面实验采用最优的前部策略。对单核 CPU 的机器分别划分 1TH，2TH，4TH，8TH。论文实验研究线程个数变化对多线程优化后的 TDIndex 查询性能的影响。实验数据集为 100W 数据元组，查询窗口 Q 的时间跨度取 10%T，Q 随机生成 500 个，对线程个数变化的查询性能实验结果如图 6-17 所示。得出结

论单核 CPU 运行多线程策略时，性能未有提升，相反，还增加额外开销。

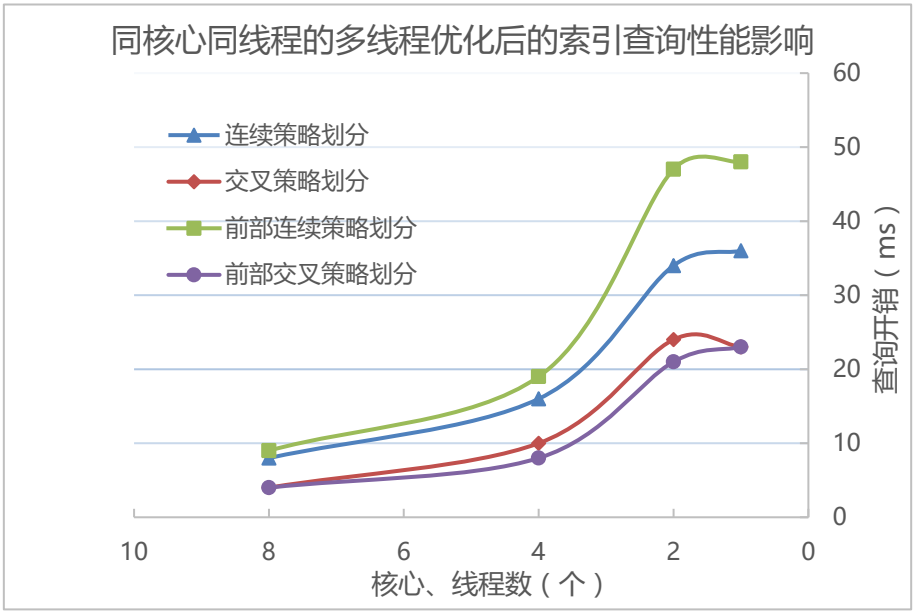


图 6-16 同核心同线程的多线程优化后的索引查询性能影响

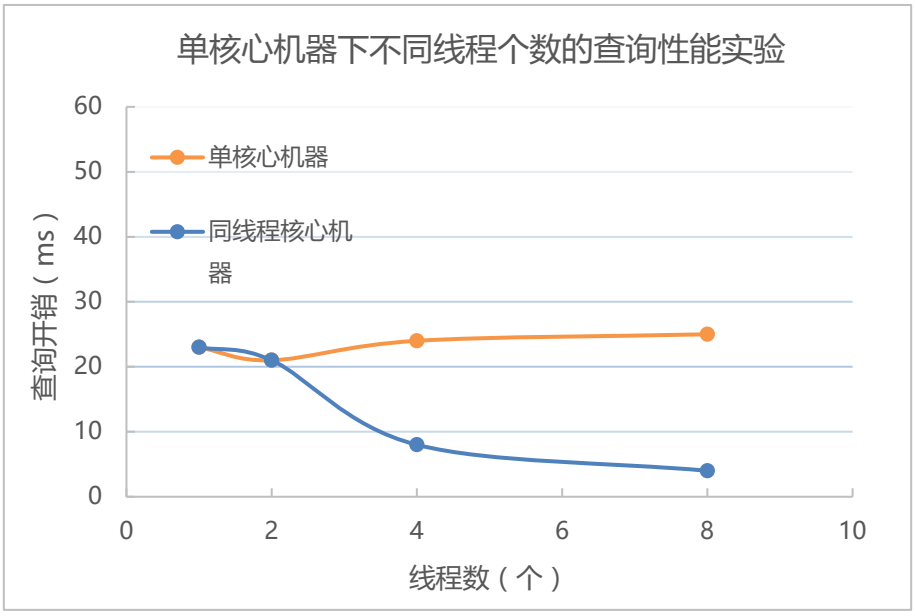


图 6-17 单核心机器下不同线程个数的查询性能实验

6.4 本章小结

本章对时态操作、磁盘中的 TDIndex 与 mysql 索引查询做了对比。时态数据索引的多线程策略划分进行实验仿真，评估论文中索引的可行性和有效性，实验通过多线程与单线程做对比，表明多线程实现算法的优势，从线序划分的

特点出发，用四种不同的多线程划分策略进行对比，表明 TDIndex 时态索引采用多线程不同划分策略的优势。在数据集大小、查询期间跨度、实现线程的核心数等多个维度进行了实验分析，实验结果可见对于该索引对于经典的工作有很大的优势，表明论文工作的重要意义。

第7章 总结与展望

在新兴互联网时代，数据库中的数据与时间有着越来越密切的关系。对于之前的很多工作在理论上已经知晓了时态索引的性能优于传统的数据库中时间的处理，但始终还未有真正的系统。

在时态索引的查询中，采用上层索引存放于一个文件，针对每个线序分支的数据结点都以文件存储的形式，模拟 Mysql 数据 B+索引的底层磁盘存储方案，设计磁盘中的 TDIndex，与 Mysql 做了对比，用实验证明了磁盘中的 TDIndex 对于查询时态数据的性能大于 mysql 索引查询时态数据的性能。对于单条 LOB 分支内部的索引查询，进行了优化，优化方法是根据开始有效时间序列的单增特性和结束有效时间序列的单减特性，提出始末序列二分算法，减少了总体的比较次数，性能得到提升。

在时态数据量增大的前提下，实际应用中服务器 CPU 的核数与个数不断增大，有必要对单个查询做并行化的处理。本论文在时态索引数据结构的基础上，完善了时态索引更新的内容，在时态索引中对于期间包含查询采用并行计算的思路，在保证硬件软件系统支持的情况上利用并行的思路实现了多线程查询，对时态数据集合分别进行线序连续分组、线序交叉分组、线序前部策略分组等，搭建了 TempMT_Index 平台，其实现了很多基本的时态数据库功能，包括数据索引 TDIndex，TDIndex 的数据查询和更新等，将多线程技术加入进来模拟真实的时态数据库系统，其对多线程时态数据查询运用到时态数据库系统中具有重要的意义。最后，设计了相应实验仿真，采用通用数据，并通过理论和实验证明，多线程查询的可行性以及有效性，在性能方面通过对多线程的多组实验，证明了多线程查询对于性能的提升远高于不采用多线程的情况，在对四种不同的划分策略比较中，得出采用前部交叉划分策略的多线程查询，能够最大的提升性能。与现有工作进行比较评估。将本文研究的大部分的内容都放入该平台中运行，体现了本文理论的实用性和可行性。方便用户使用其他各项的数据。

后续的研究可以从几个角度进行。未来对于索引的应用，可考虑不止运用到包含查询和被包含查询两个算子中，也可在其他的查询算子中实现。对多线

程查询进行进一步深入研究，使考虑多线程并行最大化，由于本文研究出的最优线程优化策略是前部交叉策略划分。其划分策略不止是可用于多线程的工作中，还可以扩展应用到分布式计算等领域，具有扩展意义。接下来的工作，可以分析单个线序的长度，将单个线程需要检索的线序的长度也让其能够平衡，更好的做到负载均衡的效果。也可以将划分策略运用于大数据量的分布式计算中去。

参考文献

[1] 叶小平,汤庸,林衍崇,陈钊滢,张智博,陈瑞鑫. 时态数据索引 TDindex 研究与应用[J]. 中国科学:信息科学,2015,(08):1025-1045.

[2] 汤庸. 时态数据库导论. 第一版. 北京: 北京大学出版社, 2004

[3] 王珊. 数据库技术回顾和展望. 李昭原 主编, 数据库技术新进展(第 2 版).北京: 清华大学出版社, 2007, 1~22.

[4] J F Allen．Maintaining knowledge about temporal intervals．Communication of the ACM，1983，26(11)：832—843

[5] 眭俊华,刘慧娜,王建鑫,秦庆旺. 多核多线程技术综述[J]. 计算机应用,2013,(S1):239-242+261.

[6] 叶小平,周畅,廖青云,朱峰华. DTindex:分布式时态索引技术[J]. 华南师范大学学报(自然科学版),2013,(03):40-44.

[7] 汤庸等. 高级数据库技术与应用. 第 2 版. 高等教育出版社

[8] Mário A. Nascimento. Efficient Indexing of Temporal Databases via B+-Trees. Ph D dissertation. USA: School of Engineering and Applied Science Southern Methodist University, 1996

[9] XU X, HAN J, LU W. RT-tree: An improved R-tree index structure for spatio-temporal databases[C]. Proceedings of the 4th International Symposium on Spatial Data H and ling. Zurich: Springer,1990:1040~1049.

[10] Bliujute R, Jensen C S, Saltenis S, et al. Light-weight indexing of general bitemporal data[C]//Scientific and Statistical Database Management, 2000. Proceedings. 12th International Conference on. IEEE, 2000: 125-138.

[11] Bliujute R, Jensen C S, Saltenis S, et al. R-tree based indexing of now-relative bitemporal data[C]//VLDB. 1998: 345-356.

[12] 康向锋,汤庸,叶小平,汤娜. 一种基于时态中间件的高效双时态索引模型[J]. 计算机科学,2005,(09):91-95.

[13] 谭柱成.时态数据库中间件系统 TDBEngine 的研究与开发.中山大学硕士学位论文,2005.

[14] 康向锋. 一种基于时态中间件的高效双时态索引模型[A]. 中国计算机学会

数据库专业委员会.第二十二届中国数据库学术会议论文集（研究报告篇）[C].中国计算机学会数据库专业委员会:,2005:1.

[15] TimeDB—A Temporal Relational DBMS．http://www.timeconsult.com/Software/Software.html

[16] 汤庸. TempDB:时态数据管理系统[A]. 中国计算机学会数据库专业委员会（CCF DBTC）.NDBC2010 第 27 届中国数据库学术会议论文集(B 辑)[C].中国计算机学会数据库专业委员会（CCF DBTC）:,2010:4.

[17] Tao Y, Papadias D. The mv3r-tree: A spatio-temporal access method for timestamp and interval queries[C]//Proceedings of Very Large Data Bases Conference (VLDB), 11-14 September, Rome. 2001.

[18] Tao Y, Papadias D, Sun J. The TPR*-tree: an optimized spatio-temporal access method for predictive queries[C]//Proceedings of the 29th international conference on Very large data bases-Volume 29. VLDB Endowment, 2003: 790-801.

[19] Abdelguerfi M, Givaudan J, Shaw K, et al. The 2-3TR-tree, a trajectory-oriented index structure for fully evolving valid-time spatio-temporal datasets[C]//Proceedings of the 10th ACM international symposium on Advances in geographic information systems. ACM, 2002: 29-34.

[20] Pfoser D, Jensen C S, Theodoridis Y. Novel approaches to the indexing of moving object trajectories[C]//VLDB. 2000: 395-406.

[21] Song Z, Roussopoulos N. SEB-tree: An approach to index continuously moving objects[C]//International Conference on Mobile Data Management. Springer Berlin Heidelberg, 2003: 340-344.

[22] Xu X, Han J, Lu W. RT-tree: an improved R-tree index structure for spatiotemporal databases[C]//Proceedings of the 4th international symposium on spatial data handling. IGU Commission on GIS, 1990, 2: 1040-1049.

[23] Nascimento M A, Silva J R O. Towards historical R-trees[C]//Proceedings of the 1998 ACM symposium on Applied Computing. ACM, 1998: 235-240.

[24] Tao Y, Papadias D. Efficient historical R-trees[C]//Scientific and Statistical Database Management, 2001. SSDBM 2001. Proceedings. Thirteenth

International Conference on. IEEE, 2001: 223-232.

[25] Rizzolo F, Vaisman A A. Temporal XML: modeling, indexing, and query processing[J]. The VLDB Journal—The International Journal on Very Large Data Bases, 2008, 17(5): 1179-1212.

[26] 郭欢, 叶小平, 汤庸, 等. 基于时态编码和线序划分的时态 XML 索引[J]. 软件学报, 2012, 23(8): 2042-2057.

[27] 叶小平,朱峰华,汤庸,周畅,廖青云. 一种基于线序划分的时态数据索引技术[J]. 计算机科学,2013,(01):187-190.

[28] 叶小平,汤庸,林衍崇,陈钊滢,张智博. 时态拟序数据结构研究及应用[J]. 软件学报,2014,(11):2587-2601.

[29] Ye X P, Tang Y, Guo H. Study and application of temporal index technology[J]. Science in China Press (Science in China Series F: Information Sciences), Volume 55 No.7 2009:899~913.

[30] 陈瑛,叶小平. 时态拟序数据索引 TQD-tree[J]. 计算机应用研究,2015,(03):666-668.

[31] Jan Chomicki,David Toman,Michael H. Böhlen. Querying ATSQL databases with temporal logic[J]. ACM Transactions on Database Systems (TODS),2001,26(2):.

[32] 黄永钊. 时态数据处理构件的性能优化研究与实现[D]. 广州: 中山大学, 2008.

[33] 李秀芳. 基于多核的多线程算法并行优化[D].郑州大学,2010.

[34] 郭广军,胡玉平,戴经国. 基于 Java 多线程的并行计算技术研究及应用[J]. 华中师范大学学报(自然科学版),2005,(02):169-173.

附录

另附各项工作手册。

致谢

时光飞逝，岁月如梭，，研究生的学习生涯转眼间就到了要结束的时候。在即将毕业之际，借此机会我要对所有关心和帮助过我的老师们、同学们以及一直支撑我的家人、朋友们表达最诚挚的谢意！

首先，我要感谢我的导师叶小平教授(博士生导师)。在过去三年的学习生活中，叶老师的悉心指导和不断督促使我不断获得进步。叶老师严谨治学，勤于研究，在学术领域，老师从带我读论文到一起讨论论文，使我循序渐进地掌握了做学问的功夫。同时，叶老师还教会我脚踏实地的做事态度和严谨的工作作风，让我在过去的学习生活中不断进取，也为我能更从容地面对未来的学习工作生活。叶老师在研究和生活上对我的教导，鼓舞着我一路前行，不仅传授我做学问的功夫，同时教会我要做聪明的人，谦虚做人，这将使我终身受益，在此再一次向叶老师表达我的敬意！

接下来，我要感谢华南师范大学计算机学院的院长汤庸教授在学习上教给我知识，感谢汤娜副教授在学术研究领域不吝赐教，感谢刘海老师对我悉心教导，让我不断获得进步。同时，感谢学院的辅导员谢子娟老师，张丽君老师，研究生生涯是我的辅导员，在学习和生活中给予无尽的我关怀和帮助。

在此，我还要感谢我们研究团队的林衍崇师兄、陈钊滢师姐、彭鹏师兄、郑凡清师兄、张启师姐等师兄师姐，以及徐植君同学、杜梦圆同学、王津凌同学、潘明明师妹，等等，非常感谢你们在我的研究生生活中给予我帮助和鼓励，让我度过快乐而充实的三年。实验室一周一次的论文讨论，团队的外出活动，以及好友们不断的帮助和鼓励，将成为我美好的回忆。

最后，我还要感谢家人，因为有你们无私的关爱和支持，我才能不断地克服困难，朝着自己梦想前进，静下心来做学问，心无旁骛学习技术，顺利完成自己的学业！感谢一直有你们陪伴我，未来的日子里，我将继续努力，成为你们的骄傲。