中国科学院大学

University of Chinese Academy of Sciences

硕士学位论文

多路数据流 θ 连接的流过滤问题研究

作者姓名：胡紫🄵

指导教师：范小朋 副研究员 中国科学院深圳先进技术研究院

王洋 研究员 中国科学院深圳先进技术研究院

学位类别：工程硕士

学科专业：计算机技术

培养单位：中国科学院深圳先进技术研究院

2020 年 7 月

Research on the Filtering Problem of the θ Join between Multi-way Data Streams

A thesis submitted to the

University of Chinese Academy of Sciences

in partial fulfillment of the requirement

for the degree of

Master of Engineering

in Computing Technology

By

Hu Ziyue

Supervisor: Professor Fan Xiaopeng and Professor Wang Yang

Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences

July, 2020

中国科学院大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明或致谢。本人完全意识到本声明的法律结果由本人承担。

作者签名：

日　　期：

中国科学院大学
学位论文授权使用声明

本人完全了解并同意遵守中国科学院大学有关保存和使用学位论文的规定，即中国科学院大学有权保留送交学位论文的副本，允许该论文被查阅，可以按照学术研究公开原则和保护知识产权的原则公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存、汇编本学位论文。

涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：　　　　　　　　导师签名：

日　　期：　　　　　　　　日　　期：

摘 要

数据流是一组大量、快速、顺序、连续到达的数据集合。近年来，电子商务、网络监控、广告系统等用于数据流处理的应用越来越引起重视。作为基本操作之一，θ 连接在流的处理过程中起着非常重要的作用。θ 是连接条件，包括 $<$, $\leq$, $\geq$, $>$，如果 θ 为“ = ”，则称为等值连接。其目的在于找出不同数据集中满足 θ 连接条件的特定对象。针对海量流式数据分析处理中的多路数据流 θ 连接处理，本论文从等值连接和非等值连接两个模块分别进行了研究，并且分别提出了高效的解决办法。

在非等值连接方面，本文提出了 FastThetaJoin，这是一种多个数据流中用于 θ 连接操作的优化技术。θ 连接作为许多数据分析任务中经常使用的基本查询操作，在实际应用中，对多路数据流进行 θ 连接操作是非常困难的。由于多个操作组件之间的数据移动，因此涉及巨大的通讯和计算成本，使其难以在分布式环境中有效实施。与之前的研究方法一样，FastThetaJion 也会尝试最小化 θ 连接的数量，但是在制定分区策略，删除不必要的数据和执行笛卡尔积时，FastThetaJion 提出了与其他方法不同的分段过滤策略。通过这些策略，FastThetaJoin 不仅可以有效地减少 θ 连接的操作数量而且还可以有效提高其在分布式环境中的运行效率。本文在 Spark Streaming 框架中实现了 FastThetaJoin，实验结果表明，与现有解决方案相比，本文提出的方法可以减少参与 θ 连接的数据量，加快 θ 连接处理速度，进而提高 θ 连接的性能，和现有的算法相比可以提升 30% 以上的速度。此外，优化的效果与数据流的特性有关。数据特征差距越大，优化效果越明显。

在等值连接方面，本文提出了一个新的面向流的过滤器，名为多重布谷鸟过滤器（MCF）。这是一种基于经典布谷鸟过滤器的多重布谷鸟过滤器，可用于判断在特定时间段内所有数据流中是否存在某一相同元素。该方法将多个数据集的成员资格查询分解为多个操作，并将查询置于流环境中，每个数据流对应一个过滤器。实验结果表明，随着数据流数量的增加，插入和查询操作的时间也会增加。MCF 的查询时间也随着滑动窗口的减小和窗口数量的增加而逐渐增加。

关键词：θ 连接, 多路数据流, 数据流，布谷鸟过滤器，过滤器

Abstract

A data stream is a set of data collections which is large, fast, sequential, and continuous. In recent years, applications for data stream processing such as e-commerce, network monitoring, and advertising systems have attracted increasing attention. As one of the basic operations, the connection plays a very important role in the processing of the stream. θ is the connection condition, including$<$, $\leq$, $\geq$, $>$. If θ is equal to ”=”, it becomes an equivalent connection. The purpose of θ join is to find specific objects that meet the connection conditions in di erent data sets. Aiming at the multi-stream data θ connection processing in the analysis and processing of massive streaming data, this thesis has studied separately from the two modules of equal connection and non-equivalent connection, and proposed e cient solutions respectively.

In terms of non-equivalent connections, we propose FastThetaJoin which is an optimization technique for theta-join operation on multi-way data streams. As an essential query often used in many data analytical tasks, it is di cult to implement theta-join operation on multi-way data streams in practical application. It always involves tremendous communication and computing overhead due to the data movements between multiple operation components. Therefore it is tricky to implement theta-join in a distributed environment. As with previous methods, FastThetaJion also tries to minimize the number of theta-joins, but it is distinct from others when make partition strategy, delete unnecessary data and do Cartesian product. As such, FastThetaJoin can not only e ectively reduce the number of theta-joins but also improve the e ciency of its operations in a distributed environment. We implemented FastThetaJoin in the Spark Streaming framework, characterized by its e cient bucket implementation of the parameterzied windows. The experimental results show that compared with the existing solutions, our proposed method can not only reduce the theta-join overhead and speed up the theta-join processing improve the performance of theta-join. Compared with existing algorithms, FastThetaJoin can improve the speed by more than 30%. In addition, the specific e ect of optimization is related to the nature of data streams. The greater

the data di erence is, the more obvious the optimization e ect is.

In terms of equivalent connections, we present Multiple Cuckoo Filter (MCF), a new stream-oriented filter. MCF is based on the classic cuckoo filter, which can be used to determine whether there is a certain element in all data streams in a specific time period. This method decomposes the membership query of multiple data sets into single operations. It works in data streams and each data stream has an independent filter. Experimental results show that as the number of data streams increases, the time for insert and query operations also increases. The query time of MCF also increases gradually as the sliding window decreases and the number of windows increases.

Keywords: Theta-Join, Multi-Way data streams,Data streams,Cuckoo filter,Filter

目 录

第 1 章 引言 1

1.1 研究背景和意义 1

1.2 研究内容 2

1.3 国内外研究现状 3

1.3.1 等值连接 4

1.3.2 非等值连接 6

1.4 主要贡献 8

1.5 本章小结 9

第 2 章 等值连接之 Multiple Cuckoo Filter 算法 11

2.1 过滤器的选择 11

2.1.1 布隆过滤器 11

2.1.2 布谷鸟哈希和布谷鸟过滤器 12

2.2 Multiple Cuckoo Filter 算法 14

2.2.1 框架设计 14

2.2.2 插入操作 16

2.2.3 查询操作 16

2.2.4 删除操作 17

2.3 假阳性分析 19

2.4 本章小结 19

第 3 章 非等值连接之 FastThetaJoin 算法 21

3.1 框架的选择 21

3.1.1 Mapreduce 计算框架 21

3.1.2 Spark 计算框架 22

3.2 FastThetaJoin 算法 24

3.2.1 框架设计 24

3.2.2 过滤策略 26

3.2.3 笛卡儿积策略 29

3.2.4 数据倾斜处理 30

3.3 多个数据流上的拓展 31

3.4 本章小结 31

第 4 章 性能评估 …… 33
4.1 等值连接测试 …… 33
4.1.1 概述 …… 33
4.1.2 实验配置 …… 33
4.1.3 不同数据流个数各指标的对比 …… 34
4.1.4 不同参数下查询时间的对比 …… 36
4.2 非等值连接测试 …… 37
4.2.1 概述 …… 37
4.2.2 实验配置 …… 38
4.2.3 两个数据流下各指标的对比 …… 38
4.2.4 多个数据流下各指标的对比 …… 43
4.3 本章小结 …… 43
第 5 章 总结与展望 …… 45
5.1 总结 …… 45
5.2 展望 …… 46
参考文献 …… 47
作者简历及攻读学位期间发表的学术论文与研究成果 …… 51
致谢 …… 53

图形列表

1.1 算法对比 7

2.1 布隆过滤器的一个例子 12
2.2 布谷鸟哈希 13
2.3 布谷鸟过滤器的插入 13
2.4 MCF 结构 15

3.1 MapReduce 计算框架 22
3.2 Spark 计算框架 23
3.3 MapReduce 处理过程 24
3.4 Spark 处理过程 24
3.5 两个数据流处理过程 24
3.6 粗过滤阶段 26
3.7 笛卡儿积阶段 29
3.8 数据倾斜处理 30

4.1 不同数量的布谷鸟过滤器下插入时间和包含时间的差别 36
4.2 不同滑动窗口下的查询时间 37
4.3 不同数据量下与查询时间 37
4.4 两个数据流的性能测试 39
4.5 不同窗口大小下 FastThetaJoin 和 CFS 的对比 40
4.6 不同窗口大小下笛卡儿积次数的对比 41
4.7 分区数量和运行时间的关系 42

表格列表

4.1 **一个布谷鸟过滤器** · · · · · 34
4.2 **两个布谷鸟过滤器** · · · · · 35
4.3 **三个布谷鸟过滤器** · · · · · 35
4.4 **四个布谷鸟过滤器** · · · · · 35
4.5 **多路数据流 θ 连接** · · · · · 43

第 1 章　引言

本章分为五个小节，第一小节介绍了研究背景和意义，第二小节介绍了本文研究的主要内容具体是什么，第三小节介绍了国内外的相关研究以及现有研究的局限性与挑战性，第四小节介绍了本文的贡献与创新点，第五小节是对本章内容的小结。

1.1　研究背景和意义

随着移动互联网、物联网、云计算、人工智能的进一步发展，数据已呈指数的增长，大数据时代早已到来[1,2]。企业所处理的数据已经达到 PB 级，而全球每年所产生的数据量更是到了惊人的额 ZB 级。IDC 报告指出，预计到 2020 年全球的数据总量或许会超过 40ZB，这一数据量为 2011 年的 22 倍。而中国的数据总量平均每年以 50% 的速度在增长，预计到 2020 年这一比重将为全球 21%，这一现象表明中国正在成为世界上真正的数据资源大国。在这种数据爆炸的前景下，工业界和学术界在数据存储和访问方面都面临了新的挑战，这种挑战不仅是来自数据规模的扩大，同时也来自数据本身的复杂程度，以及相应的分析处理方法。

大数据技术的战略意义并非在于掌握庞大的数据信息，更为重要的是对这些代表一定意义的数据进行专业的处理。通俗来说，如果把大数据类比为一种产业，那么该产业实现盈利的关键在于提高对数据的“加工能力”，通过“数据加工”来实现数据的“增值”。大数据技术的广泛应用使其成为引领众多行业技术进步、促进效益增长的关键支撑技术。根据数据处理的时效性，可以把大数据处理系统分为批式大数据（历史大数据）和流式大数据（实时大数据）这两类。

数据流是一组顺序、大量、快速、连续到达的数据序列，一般情况下，数据流可被视为一个随时间延续而无限增长的动态数据集合。数据流包括多种数据，例如客户使用的移动或 Web 应用程序生成的日志文件、网购数据、游戏内玩家活动、社交网站信息、金融交易大厅或地理空间服务，以及来自数据中心内所连接设备或仪器的遥测数据。此类数据需要按照记录或根据滑动时间窗口按顺序进行递增式处理，可用于多种分析，包括关联、聚合、筛选和取样。对于持续生

成动态新数据的大多数场景，采用流数据处理是有利的。

近年来，网络监控、电子商务、广告系统等数据流处理的应用越来越引起人们的关注，其中 Join 多用来探索多流元组之间的相关性。很多应用程序以流的形式产生数据[3]。数据流是一段时间内可用的数据元素序列。通常情况下，每个单独的数据流只能提供部分信息，要在许多路流式数据应用程序中回答复杂的查询，将多路数据流相互组合非常重要，由此才能获得全面的答案。作为基本操作之一，Join 在流的处理过程中起着很重要的作用。然而，单个流或多个元组中的许多元组具有相同的关键属性，例如 GPS 轨迹中的车牌号码属性，股票报价中的股票代码属性等。一些常用的方法不能根据这些流的特点进行针对性的处理，在实时性方面有待提高。

在大数据时代背景下，工业界对大数据的处理需求日益迫切，以 Google, Amazon, Alibaba 为代表的互联网公司纷纷尝试开发适合自己企业需求的大数据存储和分析平台。学术界，各大高校和研究机构在大数据处理架构和算法上做出巨大贡献[3-5]。连接运算是对海量数据进行分析处理的核心内容之一。然而，由于连接操作的代价较高，如何提高连接操作的执行性能一直是研究的热点。

1.2 研究内容

数据流一个基本假设是数据元组属性变化快。比如 QQ 用户的上线和离线状态，股票价格的上涨和下跌，GPS 轨迹数据变化等。

θ 连接是一种特殊的联接操作在数据集 R 和数据集 S 之间。从广义笛卡尔坐标系形成新连接这两个数据集的乘积，并生成满足某些条件的元组，记为 $S \bowtie R(A\theta B)$，其中 A 是数据集 R 中的属性，B 是数据集 S 中的属性，并且 θ 是连接条件，包括 $<, \leq, \geq, >$。如果 θ 为“ = ”，则称为等值连接。θ 连接包括 $<,\leq,\geq,>$。与等值连接相比，θ 连接被更广泛地用于分析和处理数据。同时，这也意味着相较于等值连接，非等值连接的时间复杂度和计算复杂度增加，特别是在大规模操作中数据，改善 θ 连接效率显得尤其重要。

多路数据流 θ 连接的流过滤问题，是指给定内存空间和响应时间要求，对 θ 连接谓词中不同关系的关键属性值的筛选和过滤，以达到减少连接开销的目的。主要研究多路数据流 θ 连接中能把精确过滤转化为模糊过滤的概率数据结构，解决能够支持元素动态删除的多路数据流中元素过滤问题；研究如何在内存

空间和响应时间的约束下，设计一个针对多路数据流 θ 连接的多重并行过滤器以提高判断的准确性而降低假阳率，发现在准确性和空间需求之间建立平衡的新策略。

在多路数据流 θ 连接中，对谓词中涉及到的多路数据流中关键属性的状态判断，比如，判断某个元素是否还属于某个集合，如果该元素已经不属于原来的集合，则可以认为其所代表的元素已经失效，那么可通过过滤该元素以减少连接成本，尤其是在高选择率的全历史数据快照查询的情况下。再如，判断某个关键属性值的变化范围（最大最小值），对于 θ 连接来说可以大大减少比较操作的次数。但是多路数据流 θ 连接中数据个体状态易改变，带来了判断多路数据流中关键属性值变化的新挑战。数据流过滤问题是指过滤数据流中满足某个准则的元组集合，以减少连接操作的成本。多路数据流 θ 连接操作中，通过连接谓词 P=（$S_1.b_1\ \theta\ S_2.a_2$）∧（$S_2.b_2\ \theta\ S_3.a_2$）∧⋯∧（$S_{n-1}.b_{n-1}\ \theta\ S_n.b_n$）可以知道，如果能够在数据流中确认出每个谓词片段中元组的某个属性是否符合一定准则，那么就可以通过选择操作或者过滤操作来优化多路数据流 θ 连接的执行计划，不管是基于 BSP 模型的连接路径，还是基于连续算子模型的 Bushy 树。对于数据流中关键属性是否属于某个集合的问题，比如垃圾邮件问题，如果正常的邮件地址包含了亿级的数量，那么判断一个邮件地址是否属于这个集合就是一个难题，完全保存合法地址在内存中是不可能的，在多路数据流的情况下变得更加严重。

本文专注于在对多路数据流 θ 连接过程中，笛卡尔积阶段之前将流数据进行过滤与筛选，从而最小化网络传输开销，进而提高 θ 连接效率。本文提出了一种基于范围划分的优化方法 FastThetaJoin，这种算法可以解决所有 θ 连接问题，即 $<$，$\leq$，$\geq$，$>$，$=$。FastThetaJoin 算法在 Spark Streaming 计算平台上进行了验证。此外，本文还专门针对等值连接提出了一种算法 Window-based Multiple Cuckoo Filter（MCF），该算法结合布谷鸟过滤器解决了某一时间段内多个数据流中某一元素是否同时存在的问题。

1.3 国内外研究现状

近年来，数据库领域对于如何在云计算环境中支持关系型数据的大规模分析处理展开了大量的研究。大多都是基于 Mapreduce 进行优化，使 θ 连接效率提高。

目前，两表连接算法已得到了较充分的研究 Facebook 的 Hive[3] 以及 Yahoo 的 Pig[4] 提供了多种 2 表连接策略，如文献[5,6] 提出在 Mapreduce 阶段后增加 “Merge” 阶段，以实现连接运算等涉及异构数据表的关系代数运算。针对多表连接运算以及两表 θ 连接的问题，文献[7] 提出一种在单个 MapReduce 任务中同时连接多个关系表的优化算法。其主要思想是当 Map 端将元组发送到 Reduce 端时，是以一对多而非一对的方式发送这些元组（即一条元组从一个 Map 节点被发送到多个 Reduce 节点），然后在 Reduce 阶段完成连接运算。文献[8] 中，对 Mapreduce 上实现任意类型的连接情况进行了细致的研究。其主要思想是将 2 个表的元组对应成一个矩阵，然后根据矩阵大小和 Reducer 数量进行划分，同一区域的元组分到同一 Reducer 上，从而实现笛卡尔积连接。然而，此种方法并不支持多表进行连接的情况。为解决多表 θ 连接运算的问题，文献[9] 提出了支持星型连接的多 θ 连接算法，然而由于只适用于星型连接，因此具有定的局限性。文献[10] 提出通过 Hibernate 曲线对超方体进行划分的方法实现连接，但是由于算法要求进行 θ 连接的数据集需要具有相同的规模，因此并不适合于实际应用。文献[11] 通过将连接关系进行划分，利用 2 个 Mapreduce 任务分别处理非等值连接以及等值连接。文献[12] 考虑到 CMD 存储的特性，通过 CMD 的存储方式优化多表 θ 连接算法，然而这需要改变数据的存储方式。

1.3.1 等值连接

有的研究工作只关注解决等值连接问题[13-16]。在等值连接方面，有不少研究采用了过滤器。由于过滤器的多样化，其带来的效果不尽相同。以是应用过滤器来进行等值连接的相关研究。

布隆过滤器（Bloom Filter)[17] 是关于数据流过滤的好工具。但是现有的关于布隆过滤器的工作，主要是用于数据流中去重, 它将每个元素用 K 个哈希函数将元素映射到 K 个对应位置，当所有位置为 1 表明元素满足条件，只要一个不为 1 则不满足条件，其缺点是不支持动态删除。

布谷鸟过滤器（Cuckoo Filter)[18] 是一个基于布谷鸟哈希的压缩表，只存储每个元素的 fingerprint（指纹），即一个通过哈希函数的映射过来的位串，而不再采用键值对的方式，因此大大提高了过滤器的空间利用率。事实上指纹的大小和假阳性目标有关，假阳性率越小则需要的指纹位串越长。布谷鸟过滤器的作者证明了布谷鸟过滤器需要的指纹大小和集合中元素的多少成对数关系。对于数十

亿元素的集合来说，布谷鸟过滤器比不能支持删除的布隆过滤器，使用更少的空间既能支持删除还能保证假阳性率小于 3%。

商过滤器（Quotient filters)[19] 也是存储指纹以支持删除的紧凑散列表。它采用线性探测作为一种解决冲突的策略，过滤器中存储余数排序后的集合元素，相同商的元素将连续存储在一个桶中。

区块布隆过滤器（Blocked Bloom Filter)[20] 由一系列相对较小的布隆过滤器组成，每个元素由哈希分区确定存储位置，分别映射到对应的一个小布隆过滤器中，每个过滤器都存储在一个 CPU 高速缓存行。但区块布隆过滤器，且由于整个小布隆过滤器组的负载不均衡，导致误报率变得更高。

d-left 计数布隆过滤器（d-left Counting Bloom filters)[21] 中的 d-Left 指的是 d-Left 哈希，解决的就是负载均衡问题。是指在传统的 Counting Bloom Filter 基础上，将哈希表分为 d 个子表，同一个哈希值高位用作地址低位用作指纹，这就意味着同一个地址对应着多个指纹。一个地址对应一个 bucket，因此一个 bucket 需要存储多个指纹。这种方法相对于 Counting Bloom Filter 空间更少，假阳率更低。

动态布谷鸟过滤器（Dynamic Cuckoo Filter，DCF）提出集合的表示结构应该能够支持集合大小的动态变化，而且该结构还应该能够支持可靠的删除操作。DCF 提出独占式的指纹来描述一个元素来保证可靠删除，提出一个动态链式结构来支持动态集合。

计数布隆过滤器（Counting Bloom Filter)[22] 的出现解决了动态删除这一问题，它将标准布隆过滤器位数组的每一位扩展为一个小的计数器（Counter)，在插入元素时给对应的 k（k 为哈希函数个数）个计数器的值分别加 1，删除元素时给对应的 k 个计数器的值分别减 1，从而实现删除功能。但是，这种方法是基于已知将要剔除掉的元素，在多路数据流的实际情况下不太可能。对于跳跃窗口模型，采用的方法是把大型跳跃窗口分解为多个子窗口，然后用相同大小的计数布隆过滤器来代表这些子窗口。Deng 等人提出一种新的概率数据结构，即稳定布隆过滤器（Stable bloom filter)。这种结构考虑到无法保存整个数据流的历史，因此通过这种结构把旧的信息去除掉，并且保证一定的假阳性。

1.3.2 非等值连接

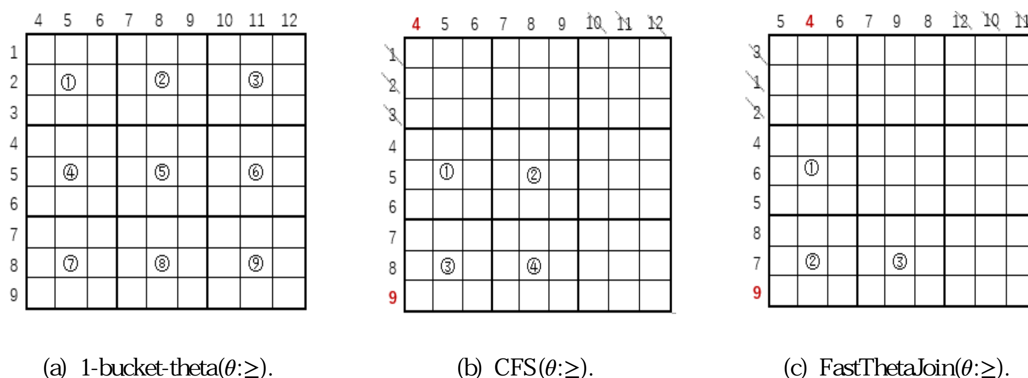
非等值连接中，目前的主流的研究中主要利用 MapReduce 框架来处理连接操作，该框架主要考虑网络中负载平衡的开销，当数据集变大时，大量的中间结果会导致很高的通信开销。在迭代式计算中，MapReduce 把每一个中间结果存于 HDFS 上，下次计算需要在从 HDFS 读取再计算，造成了很多不必要的 I/O 操作。为了解决这一问题，一些技术基于 Mapreduce 做了一些优化，尽可能使用一个 MapReduce 任务来计算。由于 Mapreduce 在迭代式计算中的劣势，一些方法在其他框架中处理 θ 连接，如 Storm[23] 或 Spark[14]，但是他们也需要改变框架以增加新的特性[6,24]。也有很多研究是针对数据流的[25-27]。

近几年也有一些着眼于流处理的研究。有的方法将静态数据集按照微批来进行处理，只要处理批次够小则这种方法可以近似地视为流处理，加快处理效率。这个方法需要使用到窗口这一概念。将小批量数据放入窗口进行处理，窗口内数据处理完成之后往窗口中灌入下一批数据[28,29]。只要窗口大小选择合适，就会达到处理数据流类似的效果。事实上，在某些情况下，这种用窗口机制实现微批处理的方式也是一个不错的选择。Continuous Query Language(CQL)[30,31]，给出了滑动窗口数据流的定义，将之视为连接关系图，每个视图包含当前窗口中的所有数据。

1-bucket-θ[8] 是评估单个 θ 连接的最常用方法。该算法基于 MapReduce 框架。它的思想是将两个输入数据集的笛卡儿积结果形成的矩形区域进行分割，然后根据分割后的结果将数据均匀分配到集群中的每个节点，实现节点之间的负载均衡。此方法首先根据指定的属性对数据集进行排序。然后根据 top-k 进行分区。即第（$0\times k+1$）到第（$1\times k$）被分到同一个分区，第（$1\times k+1$）至第（$2\times k$）分到同一个分区，依此类推。之后，将分区分发到相应的集群中的节点。接下来是笛卡尔积阶段。因为每个分区之间的大小关系未知，不同数据集中的每个分区都必须与另一个数据集中所有分区都做笛卡尔积。最后根据 条件一条条过滤得到最终结果。

图1.1(a)以两个静态数据集的 θ 连接为例，另 θ 为 $\geq$。即，选择第一个数据集大于第二个数据集的所有元素。行表示在第一个数据集中排序后指定的属性值为{1，2，3，4，5，6，7，8，9}，列表示在第二个数据集中排序后的指定属性值是{4，5，6，7，8，9，10，11，12}。根据上述规则，这里我们将 k 设置为

3。在第一个数据集中，第一条到第三条数据集，即{1，2，3}被划分为第一个分区，第四个到第六个数据划分到第二个分区，即{4，5，6}分为第二个分区，第七至第九个数据{7，8，9}划分到第三个分区。第二个数据集也以相同的方式分区。由于分区之间的大小关系未知，因此当前数据集中每个分区都必须与另一个数据集中的其他分区都进行笛卡尔积。这样我们就可以得到九部分笛卡尔积。最后，我们必须根据 θ 条件将九个部分的结果逐一选择出最终结果。



(a) 1-bucket-theta(θ:$\geq$).　(b) CFS(θ:$\geq$).　(c) FastThetaJoin(θ:$\geq$).

图 1.1 算法对比

Figure 1.1 Comparison of algorithms

Cross Filter Strategy (CFS)[32] 是一种关于 θ 连接的最新的研究方法。该方法基于 1-bucket-θ 方法，与之不同的是，CFS 用 Spark 取代了 MapReduce，并且在笛卡儿积运算开始前用数据集的最大值最小值进行了交叉过滤，预先删除不相关的数据，减少后续参与到笛卡尔积阶段的数据量，进而降低计算和存储成本。CFS 先获取每个数据集的最大值和最小值，交叉过滤之后再分区。分区方法和 1-bucket-θ 相同，每 k 个划分为一个分区，然后将数据分发到集群中的 reducer。图1.1(b)是 CFS 的一个具体例子。

图1.1(b)以和图1.1(a)同样的数据集和 θ 条件为例。不同于 1-bucket-θ，CFS 必须用最大值和最小值进行交叉过滤。如图所示，第一个数据集中的最大值为 9，第二个数据集中最小值为 4，用红色标出。由于 θ 是 $\geq$，因此显然数据集一中小于 4，和数据集二中大于 9 的所有元素肯定不会出现在最终结果中。所以我们可以在笛卡尔积之前将这些不必要的元素删除，即 \ 标记出的所有数据需要被删除。删除后进行分区。CFS 采用的分区策略与 1-bucket-θ 相同。为了方便比较，这里 k 也设置为 3。在第一个数据集中 4，5，6 划分为第一个分区，7，8，9 被划分到第二个分区。第二个数据集也以相同的方式进行分区。我们可以直观地看从

图1.1(b)和图1.1(a)中看出，相同条件下CFS的笛卡尔积数量明显小于1-bucket-θ算法。CFS只需要4个分区间的笛卡尔积，而1-bucket-θ需要9个。

Z. Khayya提出了一种基于排序、置换数组和位数组的θ连接方法IEJoin[24]。它将要连接的列放入排序数组中，并使用置换数组给元组重新排列，使其有序。该方法需要对并行框架进行根本性的修改，不便于实际应用。Mapreduce Jobs (MRJs)[11]将θ连接分解为非等连接和非等值连接，在每个θ连接处理中，也是采用1-bucket-θ方法。A multi-way theta-join[10]是一种基于1-bucket-θ的θ连接算法，其主要思想是利用Hilbert曲线实现链式连接。由于Hilbert曲线要求数据集的大小相同，因此在连接不同大小的表时不能使用。

Strict-Even-Join[33]是一种基于对称复制连接的算法。其主要思想是将多张关系表进行分片，在Map阶段扫描各表中的元组，并对应到所在的分片中。在Shu e阶段通过一对多的方式将各分片复制到相应的节点，并在Reduce端通过θ连接条件将符合条件的元组进行连接。然而随着关系表数量的增加，将会产生大量节点间不必要的数据传输。

现有大多数研究的局限性在于，虽然许多现有技术提供了join操作，但是使用的MapReduce框架，这种方式会使得大量的I/O浪费[34-36]，这是由MapReduce框架本身特性带来的。此外，大多数研究是根据在静态数据集上，很少用于流式计算。

1.4 主要贡献

本文对现有方法进行了改进，主要有以下贡献：

1. 针对等值连接，本文提出了多重布谷鸟过滤器(MCF)算法，该算法可以快速判断出某一时间段内多个数据流中是否同时存在某一元素。该算法基于传统布谷鸟过滤器，每个数据流对应于一个布谷鸟过滤器，并且增加了窗口机制，将每个数据流中待处理数据按照窗口进行切割，再将每个数据流当前窗口中的数据添加到过滤器中。在过滤器里判断元素是否存在。由于过滤器是基于哈希的，并且该算法采用一旦找到立即返回窗口不再继续向下滑动的方法，因此该算法可以快速判断元素是否同时存在于多个数据流中。

2. 针对非等值连接，本文提出了FastThetaJoin，该算法可以实现多个数据流之间的快速θ连接操作。该算法对θ连接进行了优化。主要创新点在于采用了

一种特殊的过滤策略和笛卡尔积策略。本算法的过滤策略中不需要排序，最新的 CFS 算法需要全局排序。CFS 算法快则 O(n(logn)) 慢则 O(n×n) 的时间复杂度才能实现过滤。本文的过滤策略只需要 O(1) 的时间复杂度就可以过滤掉绝大多数不必要元素，然后再局部范围内遍历过滤掉所有不必要元素。同时该算法提出的笛卡尔积策略可以使 θ 连接中数据传输和笛卡尔积次数尽可能少。由于分区间已经有序，只需要部分分区之间进行笛卡尔积。经实验验证，该算法比最新最近的相关研究时间复杂度更低，笛卡尔积次数更少，效率更高。

3. 本文基于 Spark 框架实现了非等值连接算法，并且在集群环境中与现有相关算法进行了实验对比，同时采用了 Spark SQL 作为对比实验之一，验证了实验结果的正确性与可靠性。通过实验结果表明了本文提出的非等值连接算法的高效性。

1.5　本章小结

本章介绍了本文的研究背景与研究意义，从这两方面体现了本文的研究价值。还介绍了国内外与之相关的研究工作，从等值连接和非等值连接两个方面介绍了大数据中连接操作的相关研究。等值连接是指在 θ 连接中连接符号为“ = ”号时的特殊情况下的连接，非等值连接就是指在 θ 连接中连接符号为 <，≤，≥，> 号时的情况下的连接。等值连接中，着重介绍了布隆过滤器和布谷鸟过滤器各自的原理以及优缺点；非等值连接中着重介绍了 1-bucket-theta 算法和 Cross Filter Strategy 算法的数据结构、算法原理、以及应用场景的区别。除此之外还介绍了一些等值连接和非等值连接中的其他常见算法。最后指出了现有研究的局限性以及本文的贡献与创新点。

第 2 章　等值连接之 Multiple Cuckoo Filter 算法

由于设备和应用特性的限制，单个数据流只能提供部分数据，并且用户需要集成多个数据源才能获得完整而全面的信息。对于多集合的表示和成员资格查询问题，当前提出的几种解决方案均基于布隆过滤器。受到此启发，本文设计了 Multiple Cuckoo Filter（MCF），这种算法基于经典的布谷鸟过滤器来处理多个数据流中，试图找到满足指定关系的对象，从而为研究多路数据流关系奠定了基础。与标准的布谷鸟过滤器相比，多重布谷鸟过滤器具有更多的潜力，因为它支持多个数据流的查询。

本章分为四个小节，第一小节详细对比了布隆过滤器和布谷鸟过滤器的各自优劣性，综合考虑选择基于哪一个过滤器进行下一步研究。第二小节详细介绍了本文在等值连接方面提出的 MCF 算法，第三小节分析了 MCF 算法中存在的假阳性问题，第四小节是对本章内容的小结。

2.1　过滤器的选择

2.1.1　布隆过滤器

布隆过滤器[17]，是一种适用于数据量很多时，可以快速表示数据集并且可以判断一个元素是否存在的数据结构。该过滤器基于多个不同的哈希函数，将元素经过哈希后压缩映射到多个 bit 位的数据结构。所有元素都用一个位串来进行表示。插入一个元素时，先将该元素用多个哈希函数计算并将对应的 bit 位设为 1，用这样的方式来表示一个元素。但过滤器中表示的元素达到一定程度时，可能会带来假阳性的问题。也就是本不存在的元素，经过哈希后，发现对应的 bit 位都为 1，则会被过滤器误认为存在。这是因为当元素很多时，多个不同的元素经过哈希后将对应的 bit 位设为了 1，组合祈祷可能会恰好是一个不存在的元素的哈希值的 bit 位都恰好为 1。也正是由于同样的原因，使布鲁姆过滤器不可以支持删除某个元素的操作。因此，布隆过滤器可以得到一个“肯定不存在，可能存在”的判断。因为对任何一个元素来说，但凡存在哈希之后对应的 bit 位不为 1，则该元素肯定不存在。所有 bit 位均为 1，那么该元素极大可能存在。

图2.1所示是布隆过滤器的一个简单例子，来说明布隆过滤器的插入和查询

操作以及假阳性问题。该布隆过滤器是一个由 12 个 bit 位组成的数组，所有 bit 位初始值均为 0。X_1, X_2 均为当前需要插入的元素，哈希函数有 3 个。X_1, X_2 分别经过三个哈希函数哈希后，将对应位置设为了 1，位置如图中灰色小方框所示。这就是布隆过滤器的插入过程。y_1, y_2 均为待查询的元素，查询这两个元素是否存在于布隆过滤器中。y_1 经过三个哈希函数哈希后，存在对应的 bit 位不为 1，则说明布隆过滤器中 y_1 一定不存在。y_2 经过三个哈希函数哈希后，三个对应的 bit 位均为 1，则说明布隆过滤器中 y_2 极大可能存在。然而由于之前我们往初始为空的布隆过滤器中只插入了 X_1, X_2，并没有插入 y_2，而此时却判断为查找成功，因此出现了假阳，将本不存在的 y_2 误判为存在。

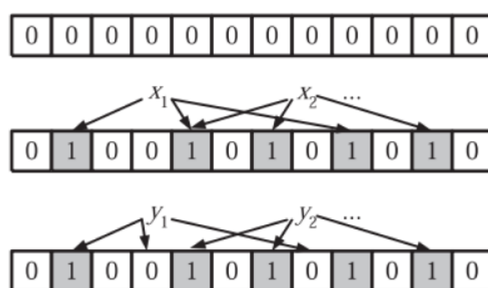


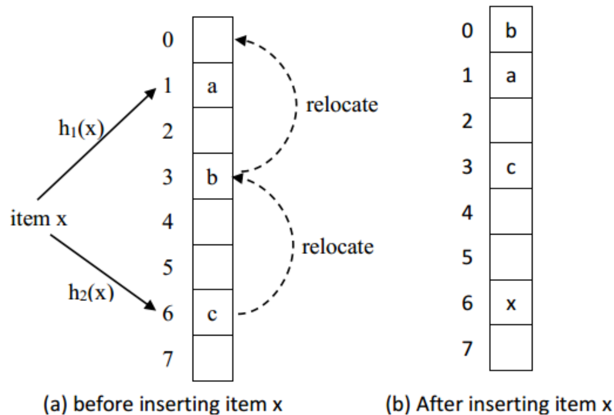
图 2.1 布隆过滤器的一个例子

Figure 2.1 An example of bloom filter

综上所述，传统的布隆过滤器存在一定的缺陷，无法删除已经存在的元素，除非重建整个过滤器，或者引入更高的假阳性。虽然布隆过滤器的一些变种可以支持删除操作，比如计数布隆过滤器（Counting bloom filter）[18]，但在某些场合仍不是一个好的选择。因此，本文选择结合布谷鸟过滤器来进行研究，设计了一种能够在多个数据流中快速判断某一元素是否在同一时间段内在每个数据流中都出现的场景。

2.1.2 布谷鸟哈希和布谷鸟过滤器

布谷鸟哈希 Cuckoo hashing[37] 是对静态数据集的一种动态化处理方式，其目的在于当发生哈希冲突时，通过将元素从其原始位置踢出到备用位置来解决哈希冲突。每个元素通过两个哈希函数来映射到两个可选位置。插入的方式为：当两个候选位置都为空时，随机选择一个位置进行插入；如果只有一个为空，则将元素放在空的位置上；如果两个都不空，则随机选择一个位置，将原本存在于该位置的数据踢出，被踢出次数加一，将当前元素放入该位置，接下来就用同样



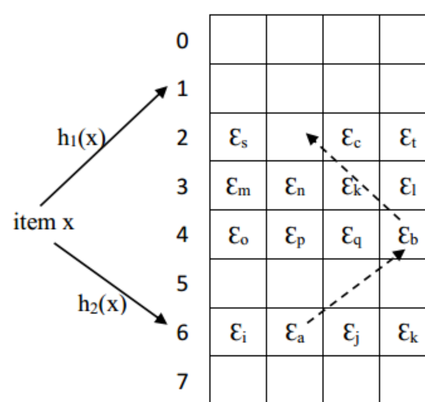


图 2.2 布谷鸟哈希　　　　**图 2.3 布谷鸟过滤器的插入**

Figure 2.2 Cuckoo hash　　　　**Figure 2.3 insert in Cuckoo filter**

的方式将被提出的元素重定位也就是重新插入，依此类推。这一插入过程会在两种情况下停止：一是元素成功插入；二是被踢出次数高于阈值，则放弃插入。

图2.2展示的就是布谷鸟哈希的插入过程。原本哈希列表里已经存在元素 *a*, *b*, *c*, 分别位于位置 1,3,6. 当前需要插入元素 *x*, 先将元素 x 用两个哈希函数进行哈希，得到对应的两个肯选位置 1 和 6。由于两个位置都不为空，因此随机选择一个位置进行剔除、插入、重定位。这里随机选到了位置 6，于是将位置 6 原本的元素 *c* 给踢出，将元素 *x* 放入位置 6，将元素 *c* 重定位。需要注意的是，重定位的时候有最大被踢出次数限制，也就是说重定位次数是有上限的。如果被踢出次数不超过给定阈值，则将 *c* 重定位到另一个备选位置，且可踢出次数减 1，否则丢弃被踢出元素。由于 *c* 的另一个备选位置是 3，则将元素 *c* 放入位置 3，将位置 3 原本元素 *b* 剔除。同理，如果被踢出次数不超过给定阈值，则将 *b* 重定位到另一个备选位置，且可踢出次数减 1，否则丢弃被踢出元素。由于 *b* 的另一个备选位置是 0，为空，则直接将 *b* 放入位置 0。图2.2(b) 是插入元素 *x* 后的结果。

如图2.3所示，布谷鸟过滤器与常规哈希表有很大不同。布谷鸟过滤器数据结构是“数组 + 链表”形式，即一个布谷鸟过滤器有一系列 bucket 组成，每个 bucket 对应于一个哈希值，且每个 bucket 由多个 entry 组成，用于存放多个相同哈希值的元素。且布谷鸟过滤器仅存储数据的指纹而不是元素本身。这里的指纹是通过将指定属性的值进行哈希得到的一串比特位。图2.3中的 ϵ_x, ϵ_a 以及 ϵ_j 就表示的是指纹。由于指纹占用的空间少于其元素本身，因此布谷鸟过滤器的空间利用率要高于普通布谷鸟哈希表。布谷鸟过滤器的增删改查大体和布谷鸟哈希

类似，一个重要区别在于布谷鸟过滤器的每个哈希值可以存放多个指纹，而布谷鸟哈希只能存放 1 个。对于每个元素，都有两个备选存储位置。由两个哈希函数来对其进行映射，映射到两个不同的 bucket。这两个哈希函数并不是完全独立，是包含异或关系的，这样可以保证每个元素只会映射到两个 bucket 而不会在下次被踢出时映射到其他 bucket。具体而言，是通过公式2.1来计算元素 x 的指纹存储位置，其中 ϵ_x 就是指指纹：

$$\begin{aligned} h_1(x) &= hash(x) \\ h_2(x) &= h_1(x) \oplus hash(\epsilon_x) \end{aligned} \tag{2.1}$$

因此，在插入元素时，使用哈希函数来计算其指纹可能的两个存储位置。插入过程类似于布谷鸟哈希，区别在于存储的是元素的指纹而不是元素本身，另一方面是是，布谷鸟过滤器每个 bucket 可以存储多个指纹，而布谷鸟哈希每个哈希值只能对应于一个元素。直到插入成功或者被踢出次数达到阈值上限，插入过程结束。要查找某个元素是否在布谷鸟过滤器中，我们需要先用两个哈希函数对该元素的指纹进行哈希，找到其可能存在的两个 bucket，再在每个 bucket 里面遍历，如果找到匹配的指纹则返回 TRUE 表示该元素存在，否则返回 FALSE 表示不存在。当然由于哈希的方式，布谷鸟过滤器存在假阳性，其特性是一定不存在可能存在。由于元素在进行哈希得到指纹时，可能不同元素得到了相同的指纹，因此说“可能存在”，即存在一定的假阳率，将原本不存在的元素误判为存在。删除时也要先对该元素的指纹进行哈希，找到其可能存在的两个 bucket，再在每个 bucket 里面遍历，如果找到匹配的指纹则从过滤器中删除该指纹。

$$\epsilon_{CF} = 1 - (1 - 1/2^f)^{2b} \approx 2b/2^f \tag{2.2}$$

布谷鸟过滤器中假阳率的上限可以通过公式2.2来计算。与布隆过滤器相比，布谷鸟过滤器具有明显的优势。支持删除操作是主要优点，并且在许多情况下，当假阳率小于 3% 时，布谷鸟过滤器还具有更好的查找性能和更高的空间效率。

2.2 Multiple Cuckoo Filter 算法

2.2.1 框架设计

多重布谷鸟过滤器 Multiple Cuckoo Filter(MCF) 由多个标准布谷鸟过滤器组成，并且个数应与数据流总数相同。即，如果要判断 N 个数据流在某个时间段

内是否同时存在某一元素，则需要给每个数据流都对应一个标准的布谷鸟过滤器，每个过滤器大小结构都可以不同。对于每个布谷鸟过滤器，窗口这一概念是解决无界数据的常用方法。滑动窗口将沿按照时间为切割单位将数据流切割为多个片段，这些片段即是在每个时间段内应该的分段数据的数据数量。比较每个滑动窗口中的指纹（这里指纹指的是指定属性经过哈希后的压缩存储方式）以检查某个时间段内同时在这多个数据流中是否存在某个元素。值得注意的是，这里的窗口大小比时间段跨度要小很多。示意图如图2.4所示。

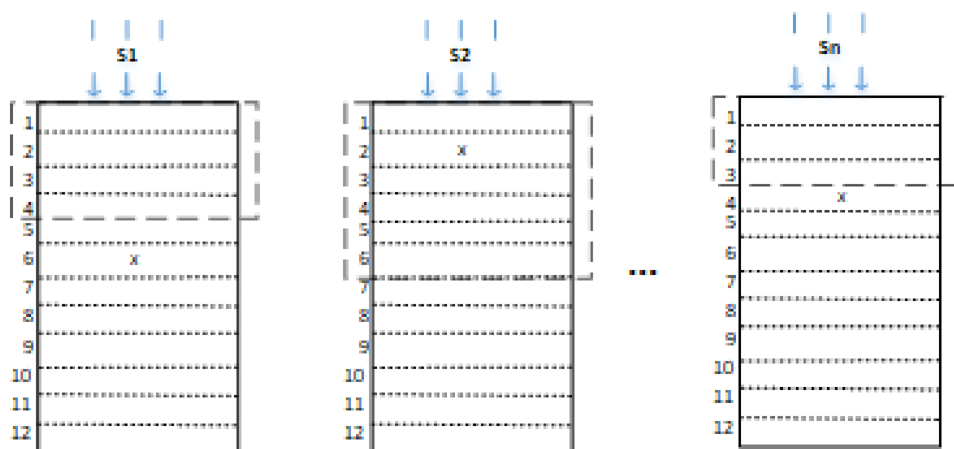


图 2.4 MCF 结构

Figure 2.4 structure of MCF

图2.4中，$S1$，$S2...Sn$ 分别表示不同的数据流，每个数据流需要找寻的是第 1 个到第 12 个元素中，是否同时存在元素 x。每个数据流先用窗口切分，窗口大小可以不同，窗口从数据流的顶部开始往下滑动，假定滑动间隔和窗口大小相同，即滑动窗口成为了滚动窗口。数据流 $S1$ 中窗口大小为 4，即一次性对数据流 $S1$ 中的 4 条数据中进行检索；数据流 $S2$ 中窗口大小为 6，即一次性对数据流 $S2$ 中的 6 条数据中进行检索；数据流 Sn 中窗口大小为 3，即一次性对数据流 Sn 中的 3 条数据中进行检索。由图可知，数据流 S1 中元素 x 在第 6 条数据中，数据流 $S2$ 中元素 x 在第 2 条数据中，数据流 Sn 中元素 x 在第 4 条数据中。先将数据流 $S1$ 的当前窗口中的所有数据插入数据流 $S1$ 的过滤器，将数据流 $S2$ 的当前窗口中的所有数据插入数据流 $S2$ 的过滤器... 将数据流 Sn 的当前窗口中的所有数据插入数据流 Sn 的过滤器。接下来同时在每个过滤器中查找当前窗口下元素 x 是否存在。数据流 $S1$ 当前窗口下没找到元素 x，且当前窗口没有到达最低端，则清空数据流 $S1$ 的过滤器，窗口下移，将新窗口中的数据全部重新填

入数据流 $S1$ 的过滤器。接下来用同样的方法查找。由图2.4可知，数据流 $S1$ 在第二种窗口中会找到元素 x，一旦找到则返回 TRUE。同样的检索方式来检索数据流 $S1$，$S2$ 一直到 Sn。只有当所有数据流都返回 TRUE 的时候，才能最终判断元素 x 在指定时间范围内在每个数据流中都存在。但凡有一个数据流返回的为 FALSE，则可直接判断元素 x 在指定时间范围内并不是在每个数据流中都存在。

2.2.2 插入操作

当对过滤器进行插入之前，需要先将滑动窗口从待处理的数据的起始位置开始，按照窗口大小和滑动间隔将数据流进行切分；若布谷鸟过滤器为空，则将窗口中的数据插入到布谷鸟过滤器中，若不为空，则清空布谷鸟过滤器，再将窗口中的数据插入到布谷鸟过滤器中。滑动窗口中的所有数据都存储在布谷鸟过滤器中。滑动窗口使用队列数据结构，因为在动态更改滑动窗口时，需要分别删除和添加窗口的头部和尾部。过滤器中存储的是每个记录的指纹而不是存储记录本身，这是为了减少存储空间的开销。

窗口中每条数据在插入过程中，需要进行如下步骤：首先要计算待插入记录的指纹，该值是由指定属性经过哈希后得到。再根据布谷鸟过滤器的两个哈希函数计算该指纹的两个候选 bucket 位置。如果两个 bucket 均负载不满，则任选一个 bucket 进行插入；如果两个 bucket 只有一个负载不满，则将带插入元素插入到负载不满的 bucket 中的任意一个空的位置；如果两个 bucket 均无剩余位置，则任选一个 bucket 中的任一位置，将原本的元素提出，将带插入元素插入，被提出的元素重新插入到另一个 bucket。

2.2.3 查询操作

算法1展示了本文提出的 Multiple Cuckoo Filter(MCF) 的查询算法，以确定是否可以同时在多个布谷鸟过滤器中找到特定元素 X。简而言之，对每个数据流而言，对查询操作的处理步骤如下:

步骤 1：计算待查询元素的指纹；

步骤 2：计算该指纹的两个候选 bucket 位置；

步骤 3：在当前窗口中查询两个候选 bucket 中是否存在指定元素，若存在则窗口停止滑动并返回 TRUE，若均不存在进行步骤 4；

步骤 4：判断窗口是否已经滑动至当前数据流的末尾，如果是则返回 FALSE，不是则窗口继续往下滑动，滑动的距离为输入的滑动间隔大小；

步骤 5：判断是否每个数据流都返回 TRUE，都是则表示该元素的确在每个数据流中都存在，但凡一个数据流返回 FALSE 则表示该元素并不是在每个数据流中都存在。

由于在步骤 2 中使用布谷鸟过滤器来存储每个记录的指纹而非记录本身，因此这里只需要更快的搜索时间与更少的存储空间就可以判断出步骤 3 中当前窗口中是否存在元素 x。并且本文采取一旦发现立即返回的策略，这种策略可以使每个数据流中一旦检测到当前窗口包含指定元素，则该数据流返回查找成功，窗口不用再继续往下滑动，这种策略可以减少查询时间和查找对象的数量。并且这里仅存储每个记录的指纹，而不是记录本身，从空间利用率而言，可以使用更少的空间。

在算法1中，第 1 行中的 $window[num]$ 表示每个数据流的窗口。滑动窗口采用队列数据结构，因为在动态更改滑动窗口时，需要对窗口的头部进行移除对窗口尾部进行添加操作。第 2 行中的 $bottomIndex[num]$ 存储的是每个数据流在指定范围内处理的数据位置的下限。第 2 行中的 $windowSize[num]$ 用于存储每个数据流的窗口大小。第 3 行中的 $item_index[num]$ 用于标定每个数据流中元素 x 的位置。第 3 行中的 $window_back[num]$ 用于存储当前窗口的下一个起点位置。第 17 行中 $move_num[i]$ 表示当前窗口的滑动间隔。

2.2.4　删除操作

删除一个过滤器中本来存在的元素时，同样需要先计算待删除记录的指纹。之后根据布谷鸟过滤器的两个哈希函数计算该指纹的两个候选 bucket 位置。接下来遍历两个候选 bucket，如果存在该指纹，则将其中一个删除。由于不同元素在哈希之后得到的指纹有可能相同，因此只删除一个指纹即可，切不可删除所有对应的指纹，否则会造成记录的缺失。

在这里的每个布谷鸟过滤器中没有单独存在的“更新操作”这一说，或则换言之，所谓“更新”即是先将该元素删除，然后重新插入新的值。

算法 1 Checkup(num, x)

Require: num for datastreams, x for checkup.

Ensure: true or false.

```
1:  deque window[num];
2:  size_t bottomIndex[num], windowSize[num];
3:  size_t item_index[num], window_back[num];
4:  for (each i ∈ 0.num) do
5:      window[i].initialize();
6:      window_back[i] ← GetNextWindowIndex();
7:  top:
8:      filter[i].clear();
9:      filter[i].Add(windowSize[i]);
10:     item_index[i] ← GetItemIndex(x);
11:     if item_index[i] != NULL then
12:         return true;
13:     end if
14:     if (window_back[i]+1 ≥ bottomIndex[i]) then
15:         return false;
16:     end if
17:     for (j ← 0...move_num[i] && window_back[i]+1 ≤ bottomIndex[i]) do
18:         window[i].pop_front();
19:         window[i].push_back(window_back[i]++);
20:     end for
21:     goto top.
22: end for
```

2.3 假阳性分析

布谷鸟过滤器本身存在假阳性，这是因为布谷鸟过滤器存的是每个元素的指纹, 而不同属性值有可能在经过哈希之后产生相同的指纹，从而造成一定的误判，将原本不存在的元素视为存在，也就发生了假阳，且在流式计算中这种假阳的情况发生的概率在可接受范围内。但布谷鸟过滤器可以保证的是，原本存在的元素一定找得到，基于这个原因本文基于布谷鸟过滤器完成多路数据流中同时存在问题的解决。在不考虑动态窗口的情况下，由于涉及到对多个布谷鸟过滤器的查询操作。因此，多重布谷鸟过滤器的误报率是指将原本不存在的元素判为了存在。假设布谷鸟过滤器的误报率为 ϵ_{CF}，那么单个布谷鸟过滤器没有发生误判的概率为 $(1-\epsilon_{CF})^s$，其中 s 是指布谷鸟过滤器的个数。因此，在本文提出的 MCF 算法中，假阳率的上限为 $1-(1-\epsilon_{CF})^s$。

再考虑到滑动窗口在动态变化之中，假设当前数据流待处理数据总共包含 m 个元素，滑动窗口的大小是 w，并且每次移动 k 个元素，则总共 $\lfloor(m-w+1)/k\rfloor+1$ 个滑动窗口生成。因此，在 S 个布谷鸟过滤器窗口中，实际上比较了 $s\cdot(\lfloor(m-w+1)/k\rfloor+1)$ 次，则 MCF 的误报率也就是假阳率应为:

$$\begin{aligned}\epsilon_{MCF} &= 1-(1-\epsilon_{CF})^{s\cdot(\lfloor(m-w+1)/k\rfloor+1)} \\ &\approx \frac{2bs(m-w+1)}{k\cdot 2^f}.\end{aligned} \tag{2.3}$$

根据公式2.3可以看出本文提出的 MCF 算法的假阳率与布谷鸟过滤器的数量，待处理元素的总数，滑动窗口的大小，窗口滑动间隔以及指纹的长短有关。具体而言，指纹越长，可以显著降低布谷鸟过滤器和 MCF 的假阳率，但这样也会大大增加空间的开销。通常，指纹的长度对布谷鸟过滤器和 MCF 有重要影响。

2.4 本章小结

本章介绍了本文提出了 MCF 算法，该方法基于经典布谷鸟过滤器。该方法将多个数据集的成员资格查询分解为多个操作，每个数据流对应一个过滤器。与传统的布谷鸟过滤器相比，MCF 可以判断在特定时间段内所有数据流中是否存在某一相同元素。

第 3 章　非等值连接之 FastThetaJoin 算法

现有大多数研究的局限性体现在一是大多数只关注等值连接，相比等值连接，在对数据的分析处理上，θ 连接适用范围更加广泛，但同时也意味着时间复杂度以及计算复杂度增加，尤其是海量数据的操作时，提高 θ 连接的效率显得尤为重要。二是虽然许多现有技术提供了 join 操作，但是使用的 MapReduce 框架，这种方式会使得大量的 I/O 浪费[34]，这是由 MapReduce 框架本身特性带来的。此外，传统 θ 连接优化方法不适用于大规模数据处理[38-40]。

本章分为四个小节，第一小节详细对比了 Mapreduce 框架和 Spark 框架的各自特性，综合考虑选择基于哪一个框架进行下一步研究。第二小节详细介绍了本文在非等值连接方面提出的 FastThetaJoin 算法，第三小节是 FastThetaJoin 算法在多个数据流中的拓展，第四小节是对本章内容的小结。

3.1　框架的选择

3.1.1　Mapreduce 计算框架

当前多数研究基于 Mapreduce 计算框架，这种方法需要大幅修改 Mapreduce 框架。如图3.1所示的是 MapReduce 的计算框架图。在集群上，输入的数据可能分布在分布式文件系统（DFS）上的多个不同物理机上。初始化 MapReduce 作业后，这些输入文件将以块的形式传输到对应的 Map 节点上。每个切片后的输入文件，都将作为一个 Map 文件来执行后续操作。在 Map 阶段中，每个 Map 文件中的每条数据都将被映射为为（$k1$，$v1$）这样的键/值对，其中 $k1$ 表示主键，$v1$ 表示值。并输出新的（$k2$，$v2$）键/值对，其中 $k2$ 表示主键，$v2$ 表示值。这些 Map 操作直接输出在 Mapper 节点上。主键可以是默认的也可以用户自定义，默认情况每条记录的第一个属性作为主键，其他作为值。自定义的时候，用户可以指定任意属性为主键，其他为值。Reduce 节点从 Map 节点上拉取对应的记录。这一过程成为 shu e（混洗），混洗可以将 Map 节点上的数据按照主键值进行分区以及排序，主键相同的会被映射到一个分区里，发送到对应的 Reduce 节点上，混洗阶段完成后每个 Reduce 节点上就会生成键/值对列表。这些列表被经过按主键值排序之后被传输到对应的 Reduce 节点。这是通过调用 Reduce 方法来处理每

个生成的键/值对列表（k2，list（v2））。Reduce 节点的输出为 list（v2）类型，然后将其传输到 DFS 节点。这里需要注意的是，Reduce 还可以产生不同类型的 list（v3）的输出。

在迭代式计算中，Mapreduce 框架劣势更为明显。Mapreduce 作业默认调度方式是 FIFO，一次只运行一个作业。Mapreduce 需要将中间数据输出到 HDFS，在迭代式操作中，需要将中间数据从 HDFS 中先取出，再进行下一步操作，当迭代次数多的时候，Mapreduce 框架会造成很多不必要的 I/O 浪费，从而在这一方面上也会导致连接时间长，效率低。例如图3.3：四个数据集的连接任务 R⋈S⋈T⋈N，大多数基于 Mapreduce 系统将该作业分解为三个顺序链接的 Mapreduce 子任务，第一个子任务负责数据集 R 和 S 的连接工作，得到中间结果集 U1 输出到分布式文件系统 HDFS 中，第二个子任务再从 HDFS 中读出 U1 与 T 进行关联连接得到中间结果 U2 写入 HDFS 文件中。第三个任务的工作就是将第二个子任务得到的结果 U2 与 N 进行关联连接得到最终结果 U3 写入 HDFS 文件中。

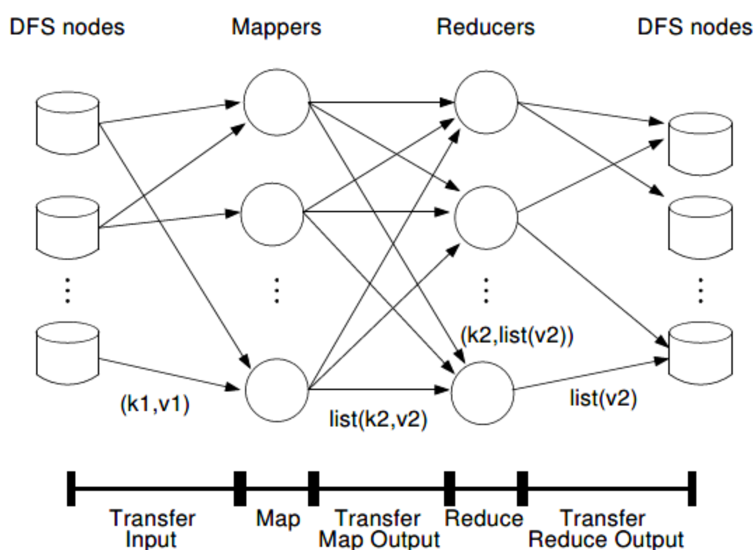


图 3.1 MapReduce 计算框架

Figure 3.1 MapReduce computing framework

3.1.2　Spark 计算框架

与 Mapreduce 框架相比，Spark 更适合用于迭代式计算。Spark 基于弹性分布式数据集 RDD，讲所有操作划分为 stage，根据整个业务逻辑画一个了逻辑上的有向图。Sprk 包含两种算子，转移算子和执行算子，转移算子只会根据任务逻辑划分 stage 而不执行当前操作。执行算子则会出发操作。图3.2所示为 Spark 在

计算过程中划分任务 stage 的一个示例。如图所示，该作业被划分为 3 个不同的 stage。*groupBy* 是一个转移算子，用于将 map 后的记录根据 key 值分组，让 key 相同的分到同一个组里。*union* 也是一个转移算子，用于合并不同的 RDD。*join* 也是转移算子，表示不同的 RDD 之间做内积操作。

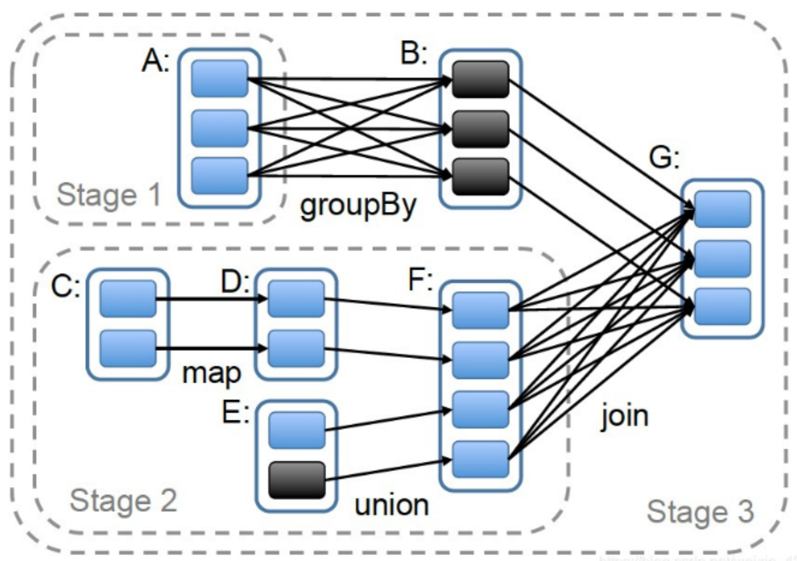


图 3.2 Spark 计算框架

Figure 3.2 Spark computing framework

但如果使用 spark 框架，则计算顺序将会发生如下改变。如图3.4spark 系统根据任务创建 DAG 图，stage1 负责数据集 R 和 S 的连接工作，得到中间结果 RDD1 并存到内存中；stage2 负责数据集 T 和 N 的连接工作，得到中间结果 RDD2 并存到内存中。stage1 和 stage2 并发执行。Stage3 负责 RDD1 和 RDD2 的连接，其结果 RDD3 为最终结果，存放于内存中。可以发现如果连接的数据集足够多或者中间结果集数据量很大则会带来巨大的磁盘 I/O 浪费。

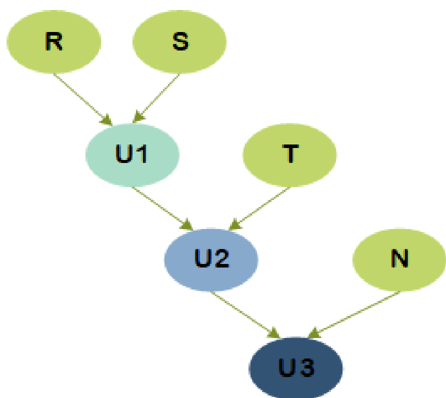


图 3.3 MapReduce 处理过程

Figure 3.3 Processing in MapReduce

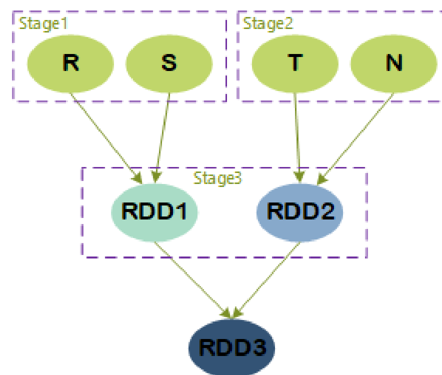


图 3.4 Spark 处理过程

Figure 3.4 Processing in Spark

3.2 FastThetaJoin 算法

在本节中，本文将详细描述 FastThetaJoin 的设计。分四小节将描述 FastThetaJoin 应用于两个数据流中 θ 连接的情况。下一个部分，将阐述 FastThetaJoin 在多路数据流中 θ 连接的应用。

3.2.1 框架设计

两个数据流之间的 θ 连接，处理流程图如图所3.5示。

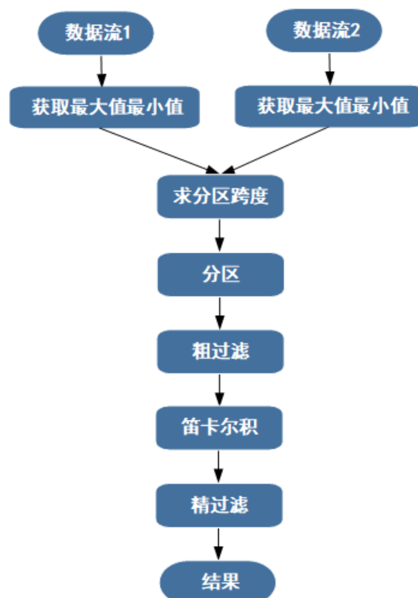


图 3.5 两个数据流处理过程

Figure 3.5 Processing of two data streams

步骤 1：分别获得两个数据流的最大值和最小值（算法2中的第 2行）。

步骤 2: 获得分区跨度。本文为了比较不同分区数与处理速度的关系，将分区数也作为其中一个传入参数。每个数据流最大值中最大的哪一个和最小值中最小的那一个作为范围边界，除以分区数之后得到分区跨度。并且在这一阶段还可以根据数据产生速率得到平均负载的预估值。

步骤 3: 每个数据流按跨度分区（算法2中的第 3 行到第 6 行）。

步骤 4: 粗过滤。用最大值最小值按照 θ 条件进行交叉过滤（算法2中的第 7 行到第 42 行）。粗过滤过程分为以下四种情况：1) 当连接条件为“数据流 1 指定属性大于数据流 2 指定属性”时。用数据流 2 的最小值作为初始过滤条件，过滤掉数据流 1 中不大于它的数据；用数据流 1 的最大值作为初始过滤条件，过滤掉数据流 2 中不小于它的数据。

2) 当连接条件为“数据流 1 指定属性小于数据流 2 指定属性”时。用数据流 2 的最大值作为初始过滤条件，过滤掉数据流 1 中不小于它的数据；用数据流 1 的最小值作为初始过滤条件，过滤掉数据流 2 中不大于它的数据。

3) 当连接条件为“数据流 1 指定属性大于等于数据流 2 指定属性”时。用数据流 2 的最小值作为初始过滤条件，过滤掉数据流 1 中小于它的数据；用数据流 1 的最大值作为初始过滤条件，过滤掉数据流 2 中大于它的数据。

4) 当连接条件为“数据流 1 指定属性小于等于数据流 2 指定属性”时。用数据流 2 的最小值作为初始过滤条件，过滤掉数据流 1 中大于它的数据；用数据流 1 的最大值作为初始过滤条件，过滤掉数据流 2 中小于它的数据。

步骤 5: 笛卡尔积。由于分区范围已知，分区间已经有序，因此这里只需要部分分区间进行笛卡尔积。

步骤 6: 最后用 θ 条件进行筛选，剔除步骤 5 笛卡儿积之后的数据中不满足条件的数据，得到最终结果。

每个流都有自己的窗口大小，可以通过输入参数来设置大小。每个数据流窗口的滑动间隔由输入参数决定。按照窗口大小对每个数据流进行拆分。这里将第一个数据流的当前窗口中的数据定义为数据集 R，将第二个数据流的当前窗口中的数据定义为数据集 S。B 是两个数据流中的公共属性之一。当然，这两个流还可以包含 B 以外的其他属性。许多现有方法非常适合于等价连接，本文提出的方法 FastThetaJoin 在非等价连接中具有更明显的优势。因此，这里仅讨论 θ 条件属于 $\{<, \leq, \geq, >\}$ 的情况。

3.2.2 过滤策略

在预处理阶段，本文将 join 属性作为键，并将所有属性合并为值，即每个元素都按照〈键, 值〉这样的 pair 对格式存储。首先，本算法需要先获得每个数据集中的最大值和最小值。然后在笛卡儿积阶段之前使用每个数据流中的最大值或最小值进行交叉过滤，这样操作的目的是尽可能减少混洗阶段的数据传输，提高混洗传输效率。由于数据流的范围和分区数已经确定（本文为研究不同分区数对性能的影响，将分区数作为传入参数之一），因此可以将数据进行范围划分，在 map 阶段将所有数据按照对应的范围进行分区。以图3.6为例来详细说明这一过滤过程。

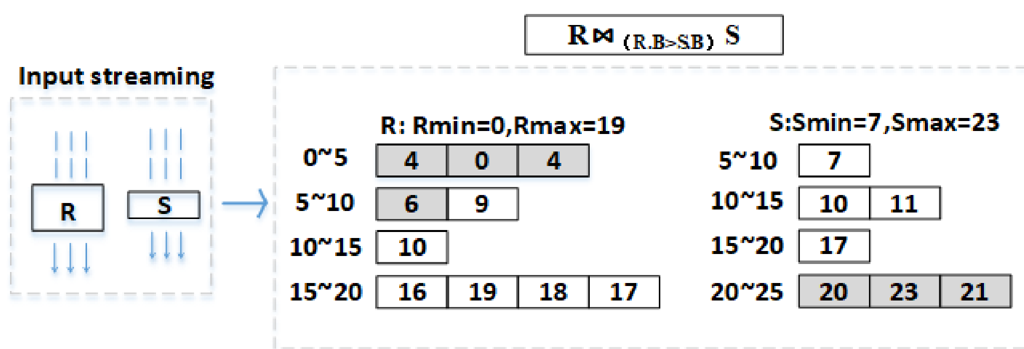


图 3.6 粗过滤阶段

Figure 3.6 Coarse filtration stage

这里假设数据流 1 包含属性 A 和属性 B，数据流 2 中包含属性 B 和属性 C。数据集 R 是数据流 1 当前窗口中亟待处理的数据，表示为 R（A，B）；数据集 S 是数据流 2 中需要处理的数据，表示为 S（B，C），连接条件为：找出数据集 R 中属性 B 的值大于 S 中 B 属性的所有数据，表示为 R$\bowtie_{(R.B>S.B)}$S。数据集 R 中属性 B 的值包含{4，0，4，6，9，10，16，19，18 17}（重复的数字是由于时间戳或其他属性不同）。数据集 R 中属性 B 的值包含{7，10，11，17，20，23，21}。Rmin 表示数据集 R 中的最小值，Rmax 表示数据集 R 中的最大值，Smin 表示数据集 S 中的最小值，Smax 表示数据集 S 中的最大值。在此示例中，Rmin = 0，Rmax = 19，Smin = 7，Smax= 23。假设此处的分区数为 4。我们可以将数据进行如下划分。对于数据集 R，[0,5) 这一分区中包含{0,1,4}；[5,10) 这一分区中包含{6,9}；[10,15) 这一分区中包含{10}；[15，20) 这一分区中包含{16，19，18，17}。同样地，对数据集 S 采用相同的分区策略来划分分区。因为 θ 是

≥，所以对于数据集 R，仅需要保留 B 属性的值大于 Smin 的数据。对于数据集 S，仅需要保留 B 属性的值小于 Rmax 的数据。由于 Smin＝ 7 属于第二个分区 [5, 10)，我们只需要 O（1）的时间复杂度来找到相应的分区，在第二个分区。然后，将数据集 R 中小于第二个分区的所有分区直接删除。对于在分区 [5, 10）的数据，分区内进行遍历即可删除数据集 R 中所有小于 Smin 的数据。由于 Rmax＝19 属于第三分区 [15,20]，因此直接删除数据集 S 中大于第三分区的分区中的所有分区，时间复杂度也只需要 O（1）。对于在分区 [15, 20）中的数据，分区内进行遍历即可删除数据集 S 中删除大于 Rmax 的数据。综上，在笛卡尔积阶段前删除图3.6中阴影部分表示出来的数据。本算法提出的过滤策略可以实现快速删除，而无需全局排序也无需全局遍历比较数据集中的所有数据。我们只需要 O（1）进行索引，O（m）的时间复杂度在一个很小的区域内一次遍历就可实现删除所有不必要的元素这一功能，并且 m 远远小于数据集中数据的总数。

本算法提出的过滤策略在算法2中显示。这一策略比目前最新的研究方法 CFS 更好，因为本算法所提出的过滤策略没有全局排序时间开销 O（n（logn））。我们仅使用 O（1）的时间复杂度即可删除大多数无用的数据。O（m）的时间复杂度在小范围内一次遍历，其中 m 远远小于数据集中数据的总数。

算法 2 FilterStrategy

Require: $firstStream$ including attr. B, secondStream including attr. B, function θ, $windowSize$, $windowSliding$.

Ensure: filtered pairs collection of $firstStream$ P_R, filtered pairs collection of $secondStream$ P_S.

1: get data sets R and S according to $windowSize$;
2: $P_R \leftarrow firstStream$.map{ hash(_.B),_.B} ;
3: $P_R \leftarrow P_R$.groupbyKey;
4: $P_S \leftarrow secondStream$.map{ hash(_.B),_.B} ;
5: $P_S \leftarrow P_S$.groupbyKey;
6: get $Rmin, Rmax, Smin, Smax$;
7: $P_R \leftarrow P_R$.filter{ record ⇒
8: **if** θ is > or ≥ **then**
9: 　　record.key ≥ hash($Smin$)
10: **else if** θ is < or ≤ **then**
11: 　　record.key ≤ hash($Smax$)
12: **end if**}

13: $P_R \leftarrow P_R$.filter{record $\Rightarrow$

14: **if** record.key $=$ hash ($Smin$) **then**

15:     **if** θ is $>$ **then**

16:         record.value $> Smin$

17:     **else if** θ is $\geq$ **then**

18:         record.value $\geq Smin$

19:     **else if** θ is $<$ **then**

20:         record.value $< Smax$

21:     **else if** θ is $\leq$ **then**

22:         record.value $\leq Smax$

23:     **end if**

24: **end if**}

25: $P_S \leftarrow P_S$.filter{ record $\Rightarrow$

26: **if** θ is $>$ or $\geq$ **then**

27:     record.key $\geq$ hash($Rmin$)

28: **else if** θ is $<$ or $\leq$ **then**

29:     record.key $\leq$ hash($Rmax$)

30: **end if**}

31: $P_S \leftarrow P_S$.filter{ record $\Rightarrow$

32: **if** record.key $=$ hash($Rmin$) **then**

33:     **if** θ is $>$ **then**

34:         record.value $> Rmin$

35:     **else if** θ is $\geq$ **then**

36:         record.value $\geq Rmin$

37:     **else if** θ is $<$ **then**

38:         record.value $< Smax$

39:     **else if** θ is $\leq$ **then**

40:         record.value $\leq Rmax$

41:     **end if**

42: **end if**}

43: window slides according to $windowSliding$;

44: return P_R, P_S;

3.2.3　笛卡儿积策略

在分布式计算中，由于通信成本很高，因此，通过控制数据分布以实现最小的网络传输，可以大大提高整体性能。在 θ 连接中，合理的分区显得尤为重要。好的分区方法可以将尽可能少的数据传输到对应的节点。本文提出的 FastThetaJoin 可以通过控制分区的方式来降低传输成本，从而使集群中的每个节点都能处理尽可能少的数据。如上一小节所述，本文提出的 FastThetaJoin 算法已经使得分区之间是有序的，分区内部是无序的，因此我们可以基于这一特点对混洗中的数据传输进行优化。优化的整体宗旨是：仅在部分分区之间进行笛卡尔积，而不是在过滤所有分区间进行笛卡尔积。如图3.7所示，这里假定使用两个节点进行 reduce。为了减少数据传输，将数据集 R 中范围为 [5、10)，[10、15) 的数据发送到 Node1，并将范围为 [15、20) 的数据发送到 Node2。数据集 S 中的所有数据都发送到 Node2，范围 [5、10)，[10、15) 中的数据发送到 Node1。这样，只需要 6 次分区间的笛卡尔积。而 CFS 和 1-bucket-theta 需要进行 3×3＝ 9 次的分区间笛卡尔积，这是由于这两种算法无法确定分区之间的大小关系。本算法提出的这种优化策略进一步减少了笛卡尔积的次数，并且在分布式计算中，数据传输也减少了，在海量数据 θ 连接中优势更加明显。

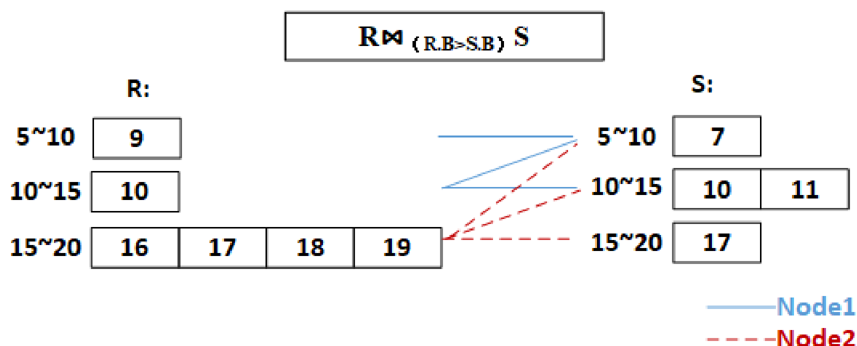


图 3.7 笛卡儿积阶段

Figure 3.7 Cartesian product phase

与目前的最新研究 CFS 方法相比，由于本算法的过滤策略和笛卡尔积策略更好，FastThetaJoin 使得笛卡尔积的数据更少。FastThetaJoin 采取了进好的策略来减少过滤开销，并在进行笛卡尔积运算之前加快了过滤速度。这里将使用图1.1(c) 进行更加详细的说明。

为了直观地反映笛卡尔乘积策略中 FastThetaJoin 与 CFS 和 1-bucket-θ 的不

同，图1.1(c)使用与图1.1(b)相同的数据集，并且 θ 仍是 $\geq$。经过最大值和最小值的交叉过滤后，由于分区之间的大小关系未知，CFS 必须计算四部分笛卡尔积。在 FastThetaJoin 中，经过交叉过滤后，显然可以知道第一个分区的范围 [4, 6] 小于第二个流的第二个分区。因此，无需计算它们之间的笛卡尔积。因此，FastThetaJoin 有更少的数据传输，以减少笛卡尔积的数量，降低网络传输成本，降低生产成本，加快 θ 连接处理时间，提高 θ 连接的性能。

3.2.4 数据倾斜处理

FastThetaJoin 还考虑数据倾斜。根据数据流产生速率，窗口大小和分区数量，很容易估计出每个分区中的预期最大负载。一旦检测到某个分区中的数据数量超过了预估阈值，就判断它会发生数据倾斜。然后，将对倾斜的数据进行重新分区以平衡负载。在完成倾斜处理过程之前，需要执行上述过滤器策略，笛卡尔积策略等。这里以图3.8为例来详细描述本算法中如何处理发生倾斜的数据。

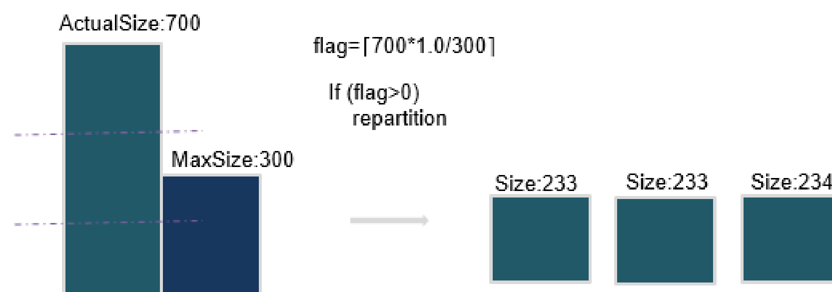


图 3.8 数据倾斜处理

Figure 3.8 Processing of data skew

MaxSize 表示每个节点在最佳性能下可以处理的最大数据量，即最大阈值。MaxSize = $\lceil windowSize/partitionNumber \rceil$。ActualSize 表示当前节点处理的实际数据量。flag 用于判断是否发生歪斜，并且可以表示如果发生倾斜要将数据拆成几份。flag = $\lceil ActualSize \times 1.0/MaxSize \rceil$。如果 flag 大于 0，则表示发生倾斜，我们需要将数据重新分配到 flag 的值所表示的分区个数中。在图3.8中，这里假设 ActualSize = 700，MaxSize = 300。因此，flag = $\lceil 700 \times 1.0/300 \rceil = 3$。flag > 0 表示数据发生了。根据本算法提出的数据倾斜处理策略，这里需要将这些数据重新划分为 3 个分区，每个分区大小分别为 233，233 和 234。

3.3 多个数据流上的拓展

本文提出的方法不仅适用于两个数据流，而且适用于多路数据流。对于多路数据流之间的 θ 连接，多路数据流可以先两两组成 pair 对。然后执行上述步骤 1 至步骤 6，对个 pair 对可以并发执行。输出结果作为下一轮的输入，再进行两两组成 pair 对，直到只有一个输出为止。对于不成对的数据，将其直接输出到下一轮，作为下一轮的输入。例如，当需要计算 n 个数据流的 θ 连接时，我们可以先计算每两个相邻的数据量的 θ 连接。例如 (R1.a1 θ R2.a2) $\bowtie$ (R2.a1 θ R3.a2) $\bowtie$...$\bowtie$ (Rn-1.ai θ Rn.aj)，其中 R1，R2... Rn 分别代表不同的流，a1，a2... ai，aj 分别表示对应流包含的属性。

算法3中展示了多路数据流 θ 连接过程。

算法 3 FastThetaJoin based on Multi-way Data Streams

Require: firstDataStream, $secondDataStream$, thirdDataStream and forthDataStream all includes attribution B, Join condition θ_1, θ_2, θ_3, $windowSize$, $windowSliding$.

Ensure: theta-join results collection (P_R.B θ_1 P_S.B).B θ_2 (P_L.B θ_3 P_T.B).B;

1: get data sets R, S, T, L according to $windowSize$;

2: $P_{R,S}$ = filtered R and S by FastThetaJoin;

3: $P_{L,T}$ = filtered L and T by FastThetaJoin;

4: $P_{(R,S),(L,T)}$ = filtered $P_{R,S}$ and $P_{L,T}$ by FastThetaJoin;

5: return $P_{(R,S),(L,T)}$;

3.4 本章小结

本章介绍了本文提出的适用于非等值连接的 FastThetaJoin 优化算法。该方法根据指定属性值进行分区，使得分区间有序分区内无序。每个数据集都先获得指定属性的最小值 min 和最大值 max。在笛卡尔积阶段之前，先预先删除两个数据流中所有不满足当前 θ 连接条件的数据，从而减少混洗阶段的数据传输和参与笛卡尔积的数据量。由于本文提出的 FastThetaJoin 算法使得分区之间是有序的，所以当执行删除操作时，可以快速锁定需要删除的分区。不符合条件的分区可以直接删除所有数据，而不必遍历数据流中的所有数据。这种分区方式不仅可以加快过滤时的删除速度，而且可以在笛卡儿积阶段部分分区间进行笛卡儿积，从而进一步减少了笛卡尔积连接的数量。

第 4 章　性能评估

本章分为三个小节，第一小节是针对等值连接方面提出的 MCF 算法的实验结果以及分析。第二小节是针对非等值连接方面提出的 FastThetaJoin 算法的实验结果以及分析。第三小节是对本章内容的小结。

4.1　等值连接测试

4.1.1　概述

本文用 C++ 来实现课题中提出的 MCF 算法。对每个查询语句，该算法先用 *CityHash*[41] 为每个元素随机生成一个 64 位的哈希值。高 32 位用于表示存储区的位置索引，低 32 位用于生成指纹。并且该算法采用 *MurmurHash*[42] 来计算每个元素的备用位置。

这里假设有 n 个数据流集合 $S=(x_1, x_2, \ldots, x_n)$，每个待处理数据集中的记录总数为 m。为了便于理论分析，同一对照组中数据流的所有记录不发生改变。滑动窗口的大小和间隔可以预先设置，也可以随机生成。在 MCF 中，如果每个数据找到了待查找元素的指纹则窗口无需向下滑动；否则窗口下滑继续查找，如果窗口滑动到下确界仍未找到待查询元素，则整个查询过程立即终止并返回 *false*，每个数据流中都找到了指定元素时才返回 *true*。这部分实验的数据是随机产生的正整数。

实验中不应将滑动窗口的大小设置得太大，因为滑动窗口越大，表示在给定时间段内包含的指纹越多，可能会导致查找效率与数据流产生速率不匹配问题，即：查找时间太长，导致数据流中未处理的数据滞留过长。因此，本文将每个滑动窗口的大小初始值设置为 2000。一旦某个窗口滑动到该数据流的下确界仍未找到待查询元素，则立即终止查询并立即返回 *false*。因为一旦在某个滑动窗口中未找到要查询的元素，则该元素一定不可能同时存在于所有数据流中。

4.1.2　实验配置

MCF 算法实验中，服务器型号是曙光 I620-G20，处理器为双核英特尔 Xeon E5-2640v3，主频为 2.60GHz，内存容量为 8 条 16G DDR4 内存条，服务器操作系统是 Centos6.5，编译工具为 gcc7.3.0。

4.1.3 不同数据流个数各指标的对比

将数据流的数量从只有 1 个数据流 1 增加到有 4 个数据流，记录不同数据流下的查询时间情况。由于每个数据流对应于一个布谷鸟过滤器，因此随着数据流的数量的增加，布谷鸟过滤器的总数也随之增加。本节用四个表来表示了在不同数据流个数前提下，每个数据流中的数据切割到为不同窗口个数时查询时间的变化。每个表格统计了三组实验数据，每组实验查询了 100 次。查询过程中的待查询元素由程序随机生成。由于每个元素都有两个候选存储索引位置，因此即使是相同的元素，在 MCF 算法中的真实存储位置也可能不同。

表 4.1 一个布谷鸟过滤器

Table 4.1 one cuckoo filter

matrics	1	2	3
min time(ms)	0	0	0
max time(ms)	335	345	298
average time(ms)	98.5	97.78	85.28
success rate	100%	100%	100%

表4.1展示的是，当只有一个数据流时，即只有一个布谷鸟过滤器时，窗口数量由 1 到 3 时查询时间的变化。并且这里将待查询元素设置为了其中随机一个已生成的数据，因此随机生成的待查询元素必然在存在于窗口中，因此滑动窗口将依次遍历每个窗口，并且每个查询肯定会成功找到该元素，所以 100 次查询中，查询成功次数为 100。由表4.1可见，只有 1 个数据流时，随着每个数据流窗口数量越多，查找时间越快。这组实验主要目的是为了证明只要待查询元素存在，MCF 算法是可以查得到的。

如表4.2所示，当布谷鸟过滤器的数量为 2 时，虽然仍然存在瞬时查询，但是成功查询的数量仅为总数的 1/3，平均查询时间几乎是一个布谷鸟过滤器的平均查询时间的 4 倍。

数据流个数再次增加，因此布谷鸟过滤器的个数再次增加。从表4.3中可以看出，很难立即找到元素。成功的检查次数越来越少，平均查询时间也越来越长，几乎是一个布谷鸟过滤器的 8 倍。

表 4.2 两个布谷鸟过滤器

Table 4.2 two cuckoo filters

matrics	1	2	3
min time(ms)	0	0	0
max time(ms)	1023	991	874
average time(ms)	400.88	368.33	336.58
success rate	34%	46%	36%

表 4.3 三个布谷鸟过滤器

Table 4.3 three cuckoo filters

matrics	1	2	3
min time(ms)	521	472	274
max time(ms)	1525	1589	1271
average time(ms)	898	994.57	855
success rate	6%	7%	10%

如表4.4所示, 当布谷鸟过滤器的数量为 4 时，可以发现每 100 个查询只有 3 次成功，成功率下降到 3%。

表 4.4 四个布谷鸟过滤器

Table 4.4 four cuckoo filters

matrics	1	2	3
min time(ms)	520	1	38
max time(ms)	1545	1457	1632
average time(ms)	1088.25	821.33	1056.33
success rate	4%	3%	3%

从上图的数据可以看出，当布谷鸟过滤器的数量从 1 增加到 4 时，成功查询的数量将大大减少，并且检查时间会线性增加。当布谷鸟过滤器的数量更多时，很难在滑动窗口中的任何时间成功地检查指定元素的指纹。散列函数越多，散列

冲突发生的可能性就越小，MCF 的假阳性率越小。

插入操作就是指将元素添加到布谷鸟过滤器中，包含操作是指检索过滤器中是否包含某些特定元素。如图4.1所示，将布谷鸟过滤器总数从 1 增加到 10，可以看出，随着布谷鸟过滤器数目的增加，插入和查询时间呈线性增加。插入操作的增长率高于查询操作的 40%，因为插入期间可能会发生踢出和重定位。

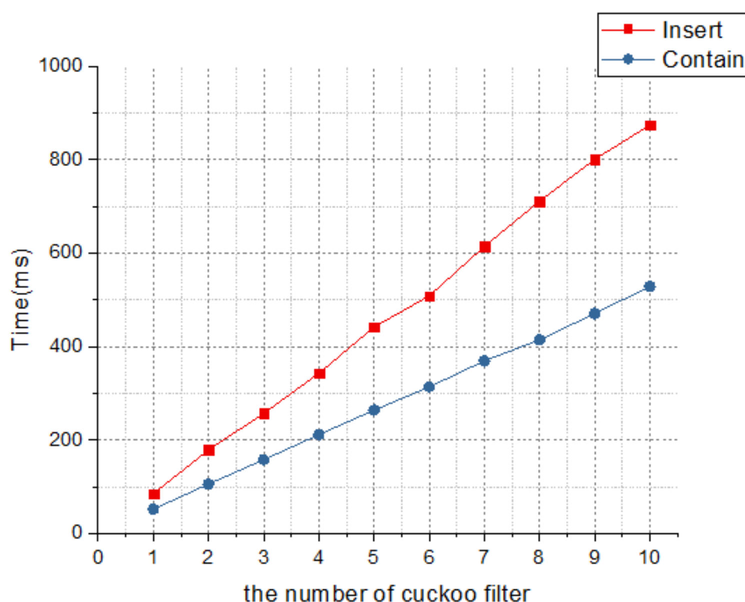


图 4.1 不同数量的布谷鸟过滤器下插入时间和包含时间的差别

Figure 4.1 Insert and Contain time and the number of cuckoo filters.

4.1.4 不同参数下查询时间的对比

当研究查询时间与滑动窗口大小之间的关系时，不再随机生成滑动窗口的大小。滑动窗口的大小上限等于元素总数。对每个数据流而言，窗口越小，该数据中窗口数量则会越多。从实验结果可以看出，如图4.2所示，当布谷鸟过滤器为 2 个和布谷鸟过滤器为 3 个时，查询时间随着滑动窗口数量的增加而减少。当布谷鸟过滤器的数量较大时，某些元素的查询时间也会显着增加。简而言之，在相同数量的布谷鸟过滤器之间，滑动窗口越小，滑动窗口数越多，检查时间越短。

由于 $size_t$ 类型可以表示从 0 到 4,294,967,295 的范围，因此设置元素的数量从 10^6 增加到 7×10^6，每次增加 10^6。随着元素总数的增加，随时间变化的滑动窗口的数量也增加。查询比较的次数也会相应增加，因此导致查询时间增加。当元素总数变大时，如果滑动窗口不增加，则每个滑动窗口中成功查询的机会将

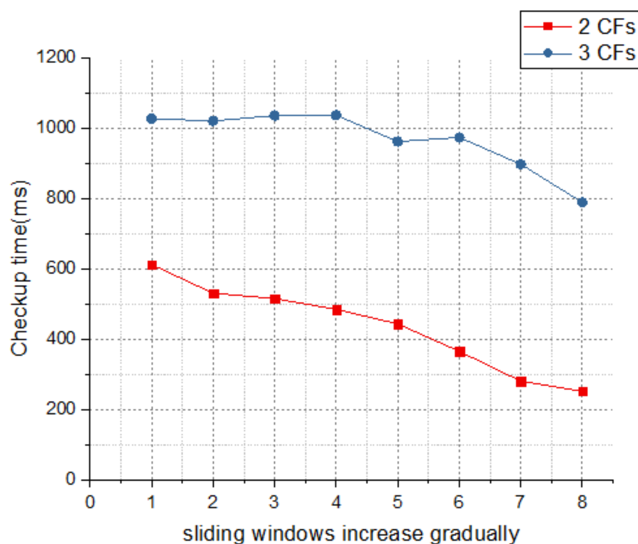


图 4.2 不同滑动窗口下的查询时间

Figure 4.2 Checkup time under different sliding windows.

减少。因此，随着总数的增加，查询时间将明显增加。如图4.3所示。

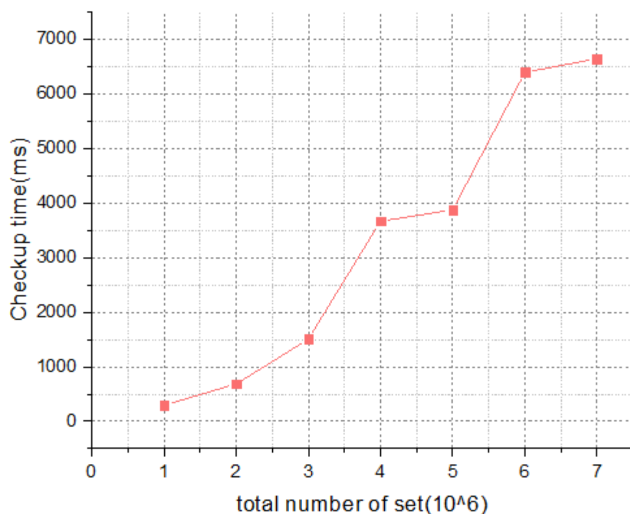


图 4.3 不同数据量下与查询时间

Figure 4.3 Checkup time under different total number.

4.2　非等值连接测试

4.2.1　概述

在实验中，我们使用其他三种方法进行比较。对于两个数据流之间的 θ 连接而言，作为对比的算法包括 CrossFilterStrategy（CFS），SparkSQL，DirectCrossProd-

uct 和本文提出的 FastThetaJoin 的算法。对于多个数据流之间的 θ 连接而言，使用 CFS 和 FastThetaJoin 作为对比进行实验。CFS 是现有最新的算法。SparkSQL 是指使用 Spark 的功能编程 API 编写用于 θ 连接的 SQL[43]。DirectCrossProduct 指直接做笛卡尔积，最后进行滤以实现 θ 连接。在实验中，我们使用了窗口机制。从数据源获取数据后，将数据按照产生时间为切割条件切成有限的块进行处理。这些基于时间的滑动窗口已得到广泛使用[44,45]。

4.2.2 实验配置

实验环境。所有实验均在 Spark Streaming 上进行评估。该集群由 14 个刀片服务器，1 个 driver 和 13 个 executor。所需的组件版本是 Spark2.3.0，Scala2.11，Kafka 2.11，JDK 1.8.0，Hadoop 2.7.4。core 总数为 24。每个 core 有 2 个线程。每个节点都是具有 32GB 内存的 AMD Opteron（TM）处理器 6238。每个节点中的操作系统是 CentOS 7.4.17。

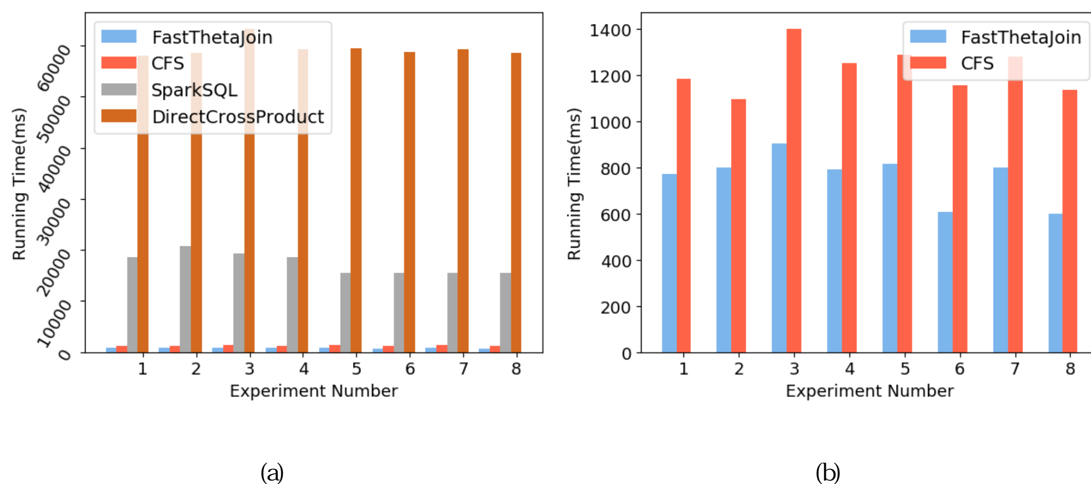
数据源。在 GitHub 上下载安装了一个数据流产生器[46]。将该数据流产生器产生的数据流作为数据源。数据流生成器的输入参数包括数据流个数，窗口时间跨度，窗口滑动间隔，每个流的数据分布类型，每个流的数据范围。实际上，窗口时间跨度和窗口滑动间隔用于计算每个流中的数据产生速率。由于这些参数由输入参数确定，因此数据源鲁邦性良好。数据流生成器生成的数据至少包含以下属性：主题，时间戳，值。

4.2.3 两个数据流下各指标的对比

每个数据流都设置一个窗口，表示当前需要处理的数据。影响 θ 连接的主要因素包括：1）窗口大小，表示一次需要拦截数据的时间范围；2）窗口滑动间隔，表示窗口滑动的时间间隔；3）分区数。实际上，窗口大小和窗口滑动度决定一起处理多少数据。因此，在本节中，我们将上述因素作为输入参数来测试每种算法的性能。另一个输入参数是 θ 的值，θ 属于 $\{<, \leq, \geq, >\}$。这里我们用 θ 为 $>$ 来说明。即，需要找出第一条数据流中属性 B 的值比第二条数据流中属性 B 的值大的所有数据。实验有两个评价指标：运行时间和笛卡尔积次数。

首先，这里假设每个窗口只处理 1000 条数据，分区总数设为 10。实验结果如图4.4所示。每次相同参数做 8 个实验。横坐标表示实验次数。纵坐标表示运行时间，以毫秒为单位。如图4.4(a)所示，DirectCrossProduct 方法执行时间最

长，几乎是 SparkSQL 的 3 倍，并且远大于其他两种方法。SparkSQL 的运行时间第二长，并且比本文提出的 CFS 和 FastThetaJoin 执行时间多得多。由于 CFS 和 FastThetaJoin 之间的差异在图4.4(a)中不明显。所以这里将 CFS 和 FastThetaJoin 的结果抽取出来作为单独的对比，其结果如图 4.4(b)所示。在图 4.4(b)中可以明显看出本文提出的 FastThetaJoin 比当前最新技术 CFS 更快。



(a)　　(b)

图 4.4 两个数据流的性能测试

Figure 4.4 Performance test of two data streams

从本文提出的 FastThetaJoin 算法设计思想上看会。很容易理解为什么 DirectCrossProduct 和 SparkSQL 的运行时间如此之长。从字面上看，DirectCrossProduct 不会对原始数据进行任何优化，在 DirectCrossProduct 中每一个数据流中的每条数据都必须与其他数据流中的每条数据进行笛卡尔积，时间复杂度为 O（n×m），因此这种方式会带来很大的数据传输，开销很大。尽管 SparkSQL 仅在满足 θ 条件时才执行笛卡尔乘积，但实际上中每一个数据流中的每条数据都会与另一个流中的其他数字进行比较。但是，由于 SparkSQL 框架将带来的一些优化，因此，SparkSQL 总是比 DirectCrossProduct 快。CFS 和 FastThetaJoin 两种方法都采取了在笛卡尔乘积之前进行过滤这一措施，使得很大一部分不必要的元素不会参与到混洗这一过程中，数据传输也会相应的减少，也减少了笛卡尔乘积的计算量，因此效率会比么 DirectCrossProduct 和 SparkSQ 要快很多。

与 CFS 相比，本文提出的 FastThetaJoin 更快的原因主要有两点：一是在过滤阶段 FastThetaJoin 没有全局排序时间，CFS 需要全局排序，这一过程所需时

间平均是 O（n（logn）)，而本文提出的 FastThetaJoin 算法仅使用 O（1）的时间即可删除大多数无用的数据，再在极小部分进行遍历；二是在笛卡尔积阶段，CFS 需要每个数据流中的每个分区都与另一个数据流中的每个分区进行笛卡尔积，而在本文提出的 FastThetaJoin 算法中，只需要部分分区之间进行笛卡尔积。从以上两个方面来看，本文提出的 FastThetaJoin 算法可以进一步减少过滤开销并减少执行时间。

为了探索不同窗口大小和运行时间之间的关系，这里将窗口大小作为变量，将数据流产生速率和窗口时间作为传入参数，窗口大小等于二者乘积。这里逐渐增加窗口大小，记录下不同窗口大小下的执行时间。首先，这里将窗口大小设置为 1000，然后测量该窗口大小下程序的运行时间，重复十次，然后取平均值作为实验结果。测量一组数据后，逐渐增加窗口大小，每组总是执行 10 次并取平均值。这里不仅计算 CFS 算法的结果，还计算 FastThetaJoin 算法的结果。图4.5展示出了窗口大小与运行时间之间的关系。由于当窗口大小仅为 1000 时，CFS 和 FastThetaJoin 处理时间已经明显比 DirectCrossProduct 和 SparkSQL 快得多，如图4.4(a)所示，因此，这里在窗口大于 1000 时仅展示 FastThetaJoin 和 CFS 的执行结果作对比。在图4.5中，横坐标表示不同的窗口大小，纵坐标表示运行时间，运行时间以毫秒为单位。实验结果表明，与 CFS 相比，FastThetaJoin 具有更短的运行时间和更快的计算速度。特别是当窗口更大时，FastThetaJoin 的优势更加明显。

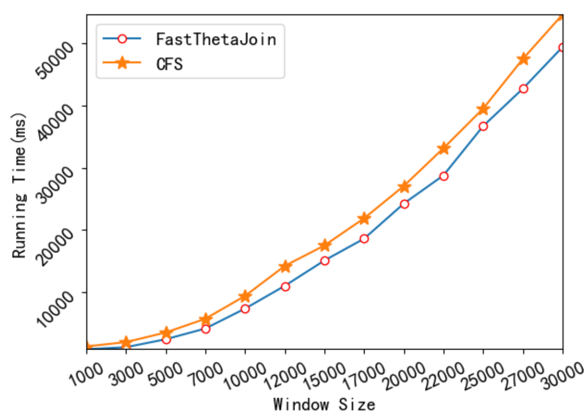


图 4.5 不同窗口大小下 FastThetaJoin 和 CFS 的对比

Figure 4.5 Comparison of FastThetaJoin and CFS under different window sizes

在实际实验过程中，我们发现，窗口大小并不是越大越好，应该应合理设置。

有两个主要原因：一方面，如果窗口大小过大，则中间结果将太多而导致 OOM 错误；另一方面，实时流计算时，窗口越大，需要处理的数据越多，处理时间越长，这可能导致数据处理时间比数据生成时间长得多，导致数据产生速率赶不上数据处理速率，两个速率不匹配会最终影响性能。

图4.6展示了不同算法中的笛卡儿积执行次数。这里首先将窗口大小设置为等于 1000，即每次处理 1000 条数据，执行 10 次，然后取平均值。之后调整窗口大小，每次增加窗口大小，同样的参数也是执行 10 次，然后取平均值。纵坐标表示笛卡儿积次数。如图4.6所示，当窗口大小固定时，就笛卡儿积次数这一评价指标而言，DirectCrossProduct 这种直接进行笛卡儿积的方法笛卡儿积次数最多，其次是 CFS，FastThetaJoin 的笛卡儿积次数最少。这表明本文提出的算法 FastThetaJoin 的笛卡儿积连接成本最低。不难发现，随着窗口大小的增加，笛卡儿积的次数将呈现爆炸性的增长趋势，这也从另一个方面表明窗口的大小不应太大。

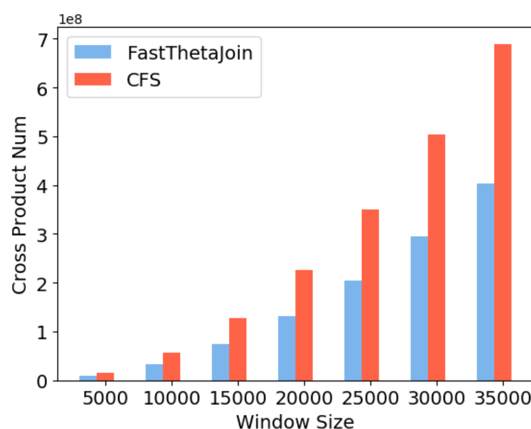


图 4.6 不同窗口大小下笛卡儿积次数的对比

Figure 4.6 Comparison of cartesian product times under different window size

实验结果表明，与预期的一样，本文提出的 FastThetaJoin 比最新方法 CFS 更快。有两个主要原因。一是没有全局排序时间，因此我们可以节省 O（nlogn）时间。另一个是在混洗阶段，因为分区是有序的，所以并非所有分区都需要彼此计算笛卡尔积。从这个角度来看，我们可以节省数据传输时间和笛卡尔积次数。本文认为 DirectCrossProduct 结果的 θ 连接是正确的，因为这种方式是每个数据流的每条数据与另一个数据流的每条数据直接进行笛卡儿积之后然后根据 θ 一条一条选择出来的结果。因此，在验证结果正确性时，我们将其他算法的 θ 连接

结果与 DirectCrossProduct 进行比较。比较结果证明，每种算法的实验结果都与 DirectCrossProduct 结果是一样的。因此以上所有实验结果都是正确、真实、有效的。

图4.7展示了分区数量与运行时间之间的关系。这里根据将输入参数中数据产生速率，窗口时间大小，窗口间隔这三个参数固定，仅改变分区数这一传入参数的大小，观察在不同分区数下处理速度的变化趋势。同样地，由于处理数据量为 1000 条数据的情况下，CFS 和 FastThetaJoin 处理时间已经明显比 DirectCrossProduct 和 SparkSQL 快得多，因此这里也仅将 CFS 与本文中提出的方法 FastThetaJoin 进行比较。可以看到，将分区数设置为 1 时，速度非常慢。随着分区数量的增加，运行时间在急剧下降后趋于稳定。这种现象是合理的。因为当分区数较少时，每个节点负载较大，因此处理速度很慢。随着分区数量的增加多，个节点并发执行，因此处理速度变快，运行时间变短。因为我们有一个估计的最小阈值。因此，在最小阈值以下，随着节点数量的增加，其他冗余节点将不参与执行，因此总执行时间保持稳定。

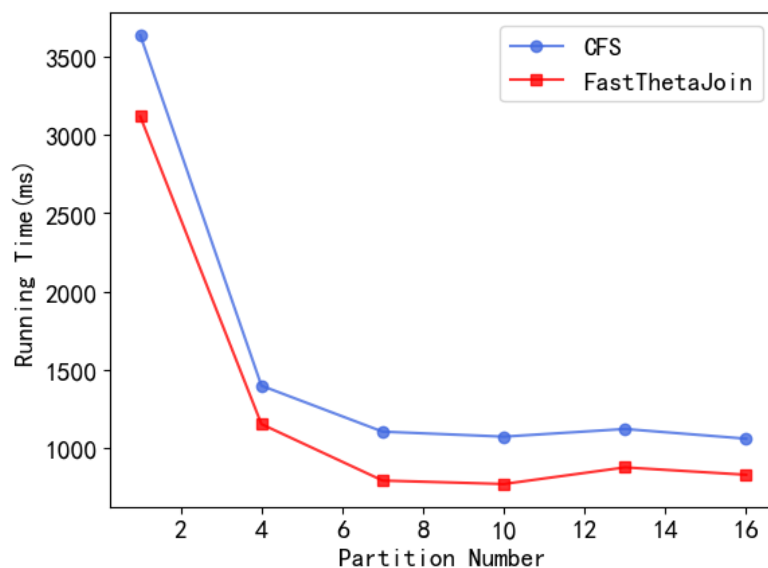


图 4.7 分区数量和运行时间的关系

Figure 4.7 The relationship between the number of partitions and the running time

表 4.5 多路数据流 θ 连接

Table 4.5 θ join of Multiple data streams

θ	FastThetaJoin	CFS	Improve Ratio(%)
$>, \geq, <$	2480	3746	33.8
$\geq, >, <$	2224	3592	38.1
$<, >, <$	2277	3536	35.6

4.2.4　多个数据流下各指标的对比

在本节中，我们将以 4 个数据流进行实验，说明多个数据流下本文提出的 FastThetaJoin 算法在 θ 连接中的执行效率。将每个窗口的大小设置为 1000，分区数仍设置为 10。对于多路数据流 θ 连接，参与对比的算法包含 CFS 和 FastThetaJoin。处理四个数据流的总体思路是将数据流两两配对分成两对，多对并发执行。第一轮的结果将进入下一轮，作为下一轮的输入，多路数据流中无法配对的数据流将自动作为下一阶段的输入参与下一阶段。如表4.5所示，第一列依次表示四个流之间的 θ 条件。第二列表示 FastThetaJoin 算法的运行时间（以毫秒为单位）。第三列表示 CFS 算法的运行时间（以毫秒为单位）。最后一列表示 FastThetaJoin 比 CFS 快多少。θ_1 为 $>$, θ_2 为 $\geq$，θ_3 为 $<$。为了提高并发效率，加快处理速度，我们可以将程序划分为以下三个块 $(P_R.\mathrm{B} > P_S.\mathrm{B}).\mathrm{B} \geq (P_L.\mathrm{B} < P_T.\mathrm{B}).\mathrm{B}$。正如我们在测试结果中看到的那样，在多路数据流 θ 连接查询中，FastThetaJoin 也比 CFS 更快。

4.3　本章小结

本章介绍了本文中提出的 MCF 和 FastThetaJoin 这两个算法的各自实验设置以及实验结果。FastThetaJoin 适用于多个数据流之间的非等值连接，其实验结果与直接先笛卡尔积再用连接条件过滤对比，发现实验结果是可靠正确的。实验结果表明该算法与最新的研究 CFS 相比更快更高效，且随着数据流数量的增加，插入和查询操作的时间也会增加。MCF 适用于多个数据流之间的等值连接，实验结果表明 MCF 的查询时间也随着滑动窗口的减小和窗口数量的增加而逐渐增加。

第 5 章　总结与展望

5.1　总结

针对海量流式数据分析处理中的多路数据流 θ 连接处理，本文从等值连接和非等值连接两个模块分别进行了研究，并且提出了高效地解决办法。

等值连接方面，本文提出了 MCF 算法，这是一种基于经典布谷鸟过滤器的多重布谷鸟过滤器。该方法将多个数据集的成员资格查询分解为多个操作，并将查询置于流环境中，每个数据流对应一个过滤器。实验结果表明，随着数据流数量的增加，插入和查询操作的时间也会增加。MCF 的查询时间也随着滑动窗口的减小和窗口数量的增加而逐渐增加。与传统的布谷鸟过滤器相比，MCF 可以判断在特定时间段内所有数据流中是否存在某一相同元素。

非等值连接方面，本文提出了一种优化技术 FastThetaJoin。此方法根据指定属性值进行分区，本算法的特有分区方式使分区间有序分区内无序。每个数据集都获得指定属性的最小值 min 和最大值 max。在笛卡尔积阶段之前，先预先删除两个数据流中所有不满足当前 θ 连接条件的数据，从而减少混洗阶段的数据传输和参与笛卡尔积的数据量。由于本文提出的 FastThetaJoin 算法使得分区之间是有序的，所以当执行删除操作时，可以快速锁定需要删除的分区。不符合条件的分区可以直接删除所有数据，而不必遍历数据流中的所有数据。这种分区方式不仅可以加快过滤时的删除速度，而且可以在笛卡儿积阶段部分分区间进行笛卡儿积，从而进一步减少了笛卡尔积连接的数量。

相比于现有研究方法，本文提出的 FastThetaJoin 有两点优化。一是过滤阶段，过滤前采取的分区策略使得分区间有序，因此只需要 O(1) 的时间复杂度即可删除大多数不必要元素，再在局部范围内遍历即可在混洗阶段笛卡尔积之前删除所有不必要元素。二是混洗阶段之后只需要部分分区之间进行笛卡尔积，这是因为本算法分区方式使得分区之间是有序的，因此只需要满足 θ 条件的分区之间进行笛卡尔积，而不需要一个数据流中的所有分区都与另一个数据流中的所有分区进行笛卡尔积。从以上两个方面进行优化，本文提出的 FastThetaJoin 算法可以实现最少的笛卡尔积运算，大大减少了　连接的数量，计算速度大大提高。性能优化效果与数据本身的特征有关。在实验中，我们发现不同数据流之间

的范围差距越大，本文提出的方法的性能越好。并且当数据流增加时，性能提升也很明显。

5.2 展望

本文基于 Spark 对多个数据流中 θ 连接问题进行了研究。Spark Streaming 是按微批处理的方式进行流式计算，流式计算中还存在一些较为前沿技术的 Flink、Blink。二者一条一条逐条处理数据，因此可以进一步调研怎样将多路数据流 θ 连接问题应用于 Flink、Blink 上。

参考文献

[1] LYNCH C. How do your data grow?[J]. Nature, 2008, 455(7209):28-29.

[2] LI G, CHENG X. Research status and scientific thinking of big data[J]. Bulletin of Chinese Academy of Sciences, 2012, 27(6):647-657.

[3] THUSOO A, SARMA J S, JAIN N, et al. Hive: a warehousing solution over a map-reduce framework[J]. Proceedings of the VLDB Endowment, 2009, 2(2):1626-1629.

[4] OLSTON C, REED B, SRIVASTAVA U, et al. Pig latin: a not-so-foreign language for data processing[C]//Proceedings of the 2008 ACM SIGMOD international conference on Management of data. ACM, 2008: 1099-1110.

[5] OLSTON C, REED B, SILBERSTEIN A, et al. Automatic optimization of parallel dataflow programs.[C]//USENIX Annual Technical Conference. 2008: 267-273.

[6] YANG H C, DASDAN A, HSIAO R L, et al. Map-reduce-merge: simplified relational data processing on large clusters[C]//Proceedings of the 2007 ACM SIGMOD international conference on Management of data. ACM, 2007: 1029-1040.

[7] JIANG D, TUNG A K, CHEN G. Map-join-reduce: Toward scalable and e cient data analysis on large clusters[J]. IEEE transactions on knowledge and data engineering, 2010, 23(9):1299-1311.

[8] OKCAN A, RIEDEWALD M. Processing theta-joins using mapreduce[C]//Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, 2011: 949-960.

[9] CHEN S Y, CHANG T P, CHANG Z H. An e cient theta-join query processing algorithm on mapreduce framework[C]//2012 International Symposium on Computer, Consumer and Control. IEEE, 2012: 686-689.

[10] ZHANG X, CHEN L, WANG M. E cient multi-way theta-join processing using mapreduce [J]. Proceedings of the Vldb Endowment, 2012, 5(11):1184-1195.

[11] YAN K, ZHU H. Two mrjs for multi-way theta-join in mapreduce[C]//International Conference on Internet and Distributed Computing Systems. Springer, 2013: 321-332.

[12] LI L, GAO H, ZHU M, et al. Multi-way theta-join based on cmd storage method[C]// International Conference on Database Systems for Advanced Applications. Springer, 2014: 62-78.

[13] BLANAS S, PATEL J M, ERCEGOVAC V, et al. A comparison of join algorithms for log processing in mapreduce[C]//Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010: 975-986.

[14] XIE D, LI F, YAO B, et al. Simba: E cient in-memory spatial analytics[C]//Proceedings of the 2016 International Conference on Management of Data. ACM, 2016: 1071-1085.

[15] AVNUR R, HELLERSTEIN J M. Eddies: Continuously adaptive query processing[C]// Proceedings of the 2000 ACM SIGMOD international conference on Management of data. 2000: 261-272.

[16] BABU S, WIDOM J. Streamon: an adaptive engine for stream query processing[C]// Proceedings of the 2004 ACM SIGMOD international conference on Management of data. 2004: 931-932.

[17] BLOOM B H. Space/time trade-o s in hash coding with allowable errors[J]. Communications of the ACM, 1970, 13(7):422-426.

[18] FAN B, ANDERSEN D G, KAMINSKY M, et al. Cuckoo filter: Practically better than bloom [C]//Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies. 2014: 75-88.

[19] BENDER M A, FARACH-COLTON M, JOHNSON R, et al. Don't thrash: How to cache your hash on flash.[J]. PVLDB, 2012, 5(11):1627-1637.

[20] PUTZE F, SANDERS P, SINGLER J. Cache-, hash-and space-e cient bloom filters[C]// International Workshop on Experimental and E cient Algorithms. Springer, 2007: 108-121.

[21] BONOMI F, MITZENMACHER M, PANIGRAHY R, et al. An improved construction for counting bloom filters[C]//European Symposium on Algorithms. Springer, 2006: 684-695.

[22] FAN L, CAO P, ALMEIDA J, et al. Summary cache: a scalable wide-area web cache sharing protocol[J]. IEEE/ACM transactions on networking, 2000, 8(3):281-293.

[23] LIN Q, OOI B C, WANG Z, et al. Scalable distributed stream join processing[C]//Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, 2015: 811-825.

[24] KHAYYAT Z, LUCIA W, SINGH M, et al. Lightning fast and space e cient inequality joins [J]. Proceedings of the VLDB Endowment, 2015, 8(13):2074-2085.

[25] ARASU A, BABCOCK B, BABU S, et al. Stream: the stanford stream data manager (demonstration description)[C]//Proceedings of the 2003 ACM SIGMOD international conference on Management of data. 2003: 665-665.

[26] ARASU A, BABCOCK B, BABU S, et al. Characterizing memory requirements for queries over continuous data streams[J]. ACM Transactions on Database Systems (TODS), 2004, 29 (1):162-194.

[27] ARASU A, WIDOM J. A denotational semantics for continuous queries over streams and relations[J]. ACM Sigmod Record, 2004, 33(3):6-11.

[28] [EB/OL]. https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101.

[29] [EB/OL]. https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102.

[30] ARASU A, BABU S, WIDOM J. The cql continuous query language: semantic foundations and query execution[J]. The VLDB Journal, 2006, 15(2):121-142.

[31] GOLAB L, ÖZSU M T. Update-pattern-aware modeling and processing of continuous queries [C]//Proceedings of the 2005 ACM SIGMOD international conference on Management of data. ACM, 2005: 658-669.

[32] LIU W, LI Z, ZHOU Y. An e cient filter strategy for theta-join query in distributed environment[C]//2017 46th International Conference on Parallel Processing Workshops (ICPPW). IEEE, 2017: 77-84.

[33] ZHANG C, LI J, WU L, et al. Sej: an even approach to multiway theta-joins using mapreduce [C]//2012 Second International Conference on Cloud and Green Computing. IEEE, 2012: 73-80.

[34] AFRATI F N, ULLMAN J D. Optimizing joins in a map-reduce environment[C]//Proceedings of the 13th International Conference on Extending Database Technology. ACM, 2010: 99-110.

[35] CHANG F, DEAN J, GHEMAWAT S, et al. Bigtable: A distributed storage system for structured data[J]. ACM Transactions on Computer Systems (TOCS), 2008, 26(2):1-26.

[36] DEAN J, GHEMAWAT S. Mapreduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1):107-113.

[37] PAGH R, RODLER F F. Cuckoo hashing[C]//European Symposium on Algorithms. Springer, 2001: 121-133.

[38] CHAUDHURI S, VARDI M Y. Optimization of real conjunctive queries[C]//Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. 1993: 59-70.

[39] LEE C, SHIH C S, CHEN Y H. Optimizing large join queries using a graph-based approach [J]. IEEE Transactions on Knowledge and Data Engineering, 2001, 13(2):298-315.

[40] TAN K L, LU H. A note on the strategy space of multiway join query optimization problem in parallel systems[J]. ACM SIGMOD Record, 1991, 20(4):81-82.

[41] [EB/OL]. https://www.cityhash.org.uk.

[42] APPLEBY A. Murmurhash[J]. URL https://sites. google. com/site/murmurhash, 2008.

[43] ARMBRUST M, XIN R S, LIAN C, et al. Spark sql: Relational data processing in spark[C]// Proceedings of the 2015 ACM SIGMOD international conference on management of data. ACM, 2015: 1383-1394.

[44] CARNEY D, ÇETINTEMEL U, CHERNIACK M, et al. Monitoring streams: a new class of data management applications[C]//Proceedings of the 28th international conference on Very Large Data Bases. VLDB Endowment, 2002: 215-226.

[45] HAMMAD M A, AREF W G, ELMAGARMID A K. Stream window join: Tracking moving objects in sensor-network databases[C]//15th International Conference on Scientific and Statistical Database Management, 2003. IEEE, 2003: 75-84.

[46] [EB/OL]. https://github.com/YongyiZhou/Multiway-Stream-Generator/blob/master/src/main/java/DSMain.java.

作者简历及攻读学位期间发表的学术论文与研究成果

作者简历

胡紫　，四川省达州市人，中国科学院大学深圳先进技术研究硕士研究生。

已发表 (或正式接受) 的学术论文:

1. MCF: Towards Window-based Multiple Cuckoo Filter in Stream Computing. Ziyue Hu, Menglu Wu, Xiaopeng Fan, Yang Wang, Chengzhong Xu. 2020 International Conference on Big Data, 2020
2. FastThetaJoin: An Optimization on Multi-way Data Stream Theta-Join with Range Constraints. Ziyue Hu, Xiaopeng Fan, Yang Wang, Chengzhong Xu. ICA3PP 2020, 2020

申请或已获得的专利:

1. **胡紫　，范小朋，须成忠，多路数据流　连接优化方法及系统**, **中国发明专利。 申请号：**CN201910774187.3
2. **胡紫　，范小朋，须成忠，多路数据流　连接优化方法及系统**, **世界知识产权组织（国际局）专利。 申请号：**PCT/CN2019/101811

参加的研究项目及获奖情况:

“流式-批处理混合计算模型下的多数据流连接优化”项目

致 谢

本论文在范小朋老师和王洋老师的悉心指导下完成。在我的研究过程中，老师们渊博的知识和开阔的视野给了我深深的启迪。在我课题研究中遇到瓶颈时，老师们为我指出问题，引导我想出解决办法；在我想要发表论文中帮助我找到切入点和创新点；在我撰写论文的时候指导我论文的写法和注意事项。老师们犹如灯塔般为我指点方向走出迷航，在我整个研究生期间给予我极大的帮助。老师们治学严谨的态度也让我深受启发。在以后的学习以及生活中，我将谨记老师的教诲，并将所知所学应用于工作生活。

感谢老师们对我毕业课题的指导，对我学习习惯的培养！