密级：_____

浙江大学

硕 士 学 位 论 文

论文题目 基于模糊查询的大数据分析处理系统的研究与实现

作者姓名 金明健

指导教师 伍赛

学科(专业) 计算机科学与技术

所在学院 计算机科学与技术

提交日期 2017 年 1 月 15 日

A Dissertation Submitted to Zhejiang University for the Degree of Master of Engineering

TITLE: An Online Big Data Analytic System Leveraging Uncertain Query Processing

Author: Jin Mingjian

Supervisor: Wu Sai

Subject: Computer Science and Technology

College: Computer Science and Technology

Submitted Date: 2017.01.15

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得 浙江大学 或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名：　　签字日期：　　年　　月　　日

学位论文版权使用授权书

本学位论文作者完全了解 浙江大学 有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权 浙江大学 可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后适用本授权书）

学位论文作者签名：　　导师签名：

签字日期：　　年　　月　　日　　签字日期：　　年　　月　　日

学位论文作者毕业后去向：

工作单位：　　电话：

通讯地址：　　邮编

摘要

随着大数据分析技术的日渐成熟，大数据所蕴含的巨大价值已经越来越被重视。由于数据量巨大，对大数据进行分析一般是很耗费时间的。然而，在很多情况下，用户并不需要精确的查询结果，数据大概的轮廓就可以满足大部分的分析需求。

本文研究并实现了一种基于模糊查询的大数据分析处理系统。该系统为用户定义了一套查询接口，这些接口支持用户进行各种聚集查询（Group By）。系统将会为用户查询返回一个模糊结果。本系统可以在秒级内返回上百 G 数据的模糊查询结果。

利用在线聚集技术可以快速生成数据轮廓的特点，本文将在线聚集技术应用到了系统中。同时，系统中相邻查询得到的结果集是有交叠的，如果能够将系统已经处理的查询所采集到的样本和计算出的中间结果保存起来，就可以加速系统处理后面查询的速度。基于此，本文对在线聚集技术做了优化。

首先，本文对数据集进行随机化处理，生成一个随机数据集，这样，就可以通过顺序扫描随机数据集来达到在数据集中随机采样的效果。然后，本文通过在线聚集技术处理用户的查询请求。在线聚集技术在生成查询结果的同时，会把已经获取的样本和产生的中间结果存储在一棵样本管理树中。相应的，用户的查询也会首先在这棵树中进行处理。当在树中查询到的结果不能满足用户的需求时，系统再从数据源读取数据。通过这种方式，在线聚集技术中采取的样本和中间结果可以有效地被多个查询使用。同时，本文还提供了一种整合多个中间结果的方法，以生成最终查询结果。最后，通过在 TPC-H 基准上的实验结果，验证了本文所设计并实现的系统的有效性。

关键词：在线聚集 样本 置信区间 树

Abstract

With the mature of big data analysis technology, large data has more and more attention for its huge value. Because of the huge amount of data, analysis of big data is usually very time consuming. However, in many cases, the user does not need accurate query results. The outline of data can meet the demand of most of the analysis.

This paper studies and completes a big data analysis and processing system based on fuzzy query. In this system, we defines a set of query interface for user. The interface allows users to do all kinds of aggregate query (Group By). System will return a fuzzy results to user. This system can process hundreds of G data in second level.

Online aggregation technology has the characteristic that it can generate data outline quickly. So we introduce this technology in our system. At the same time, adjacent query results usually overlapping in our system. If we are able to stores the acquired samples and the intermediate results that produced in previous queries, we can speed up the system of dealing with the queries.

First, we randomize the dataset to generate a random dataset, so that we can scan the random datasets sequentially to achieve the effect of randomly retrieving the data set. Then, we use the online aggregation technology to handle the user's query operation. The online aggregation technique generates the query results and stores the acquired samples and the intermediate results in a sample management tree. Accordingly, the user's query will also be first processed in the sample management tree. When the results generated from the tree can’t meet the accuracy that set by user, we continue to read data from the data source. In this way, the sample and intermediate results can be effectively used by multiple queries. We uses a number of statistical methods to integrate multiple intermediate results to generate the final query results. Finally, the experimental results on the TPC-H benchmark prove the effectiveness of the technology.

Keywords： Online Aggregation, Sample, Confidence Interval, Tree

目录

图目录

表目录

第1章 绪论

1.1 课题背景与研究意义

随着社会的发展和技术的进步，数据在当前社会中的作用越来越大。通过对数据进行分析，我们可以从中窥测出诸多有益的信息。数据分析让商家能实时读懂消费者看似毫无规律的消费者行为，从而向消费者推荐商品，获取巨大的经济收益；对科学研究来说，数据分析意味着可能出现新的科学发现；医疗保健领域利用数据分析可以更加便捷地测定临床实验的效果；气象领域利用数据分析可以更准确地发布气象预报；甚至在国家政治选举中，数据分析也起到越来越重要的作用。一般来说，信息量越大、分析用的工具越细致，分析的结果就会越准确，而根据分析结果所采取的行动就越接近成功。但是，对大数据的分析处理会因为数据量的巨大而面临很多挑战。

在进行数据分析处理时，系统一般是把数据全部扫描一遍，计算出用户的查询结果，然后返回结果。但是，我们注意到，在很多场景下，用户并不需要绝对准确的结果。用户需要的很可能只是数据的一个大概轮廓，或者一个大致的走势，尤其是在进行 Group By 这种聚集操作的时候。因此，如果能尽快勾勒出数据轮廓，那么微小的偏差也是允许的。只要分析工具能够返回一个误差可控的反映数据轮廓的结果，就可以满足分析要求。

这样，模糊查询就在大数据分析处理中就有了用武之地。模糊查询以牺牲查询的准确度为代价，换取了处理时间的缩短。随之而来的问题是，如果准确度下降的太严重，那么查询出来的结果对用户而言可用性不高。因此，查询结果的精确度能否由用户来控制？

在线聚集是由 Hellerstein 等人提出的技术。该技术通过对原始数据集进行随机采样，以获取具有良好随机性的样本数据。然后针对样本数据，使用概率统计方法，可以生成当前查询结果的置信度和置信区间。当置信度和置信区间满足了用户的要求时，用户就可以中止查询。通过在线聚集技术，就可以让用户来控制

查询结果的精度。

同时，我们注意到，对于很多聚合类型的查询，两个查询得到的结果之间往往是有交集的。如果能利用好查询之间的交集，让前面查询的中间结果可以被后面的查询使用，那么就可以加快查询的速度。

基于这种模糊查询技术，以及重复利用查询中间结果的想法，本文研究并实现了一种基于模糊查询的大数据分析处理系统，该系统支持聚合类型的查询（Group By 查询），可以在秒级内返回百 G 数据的聚合查询结果。该系统为用户定义了一套查询操作，通过这些接口，用户可以自由地对高维数据进行各种聚合查询。

1.2 模糊查询和大数据分析处理研究情况

由于本文采用了在线聚集技术来支持模糊查询，因此下面主要介绍在线聚集技术的研究情况。

在线聚集技术最早是由 Hellerstein，Hass 和 Wangle 三人提出的[1]。该技术对数据集进行随机采样，然后对样本做近似估计，利用置信度和置信区间保证结果的准确度和误差范围。该技术可以在短时间内返回给用户一个准确度可控的聚集结果。

在线聚集技术被提出后，很多学者对其做了研究和改进。Hellerstein 等人提出的在线聚集方案使用了中心极限定理来计算置信度和置信区间，Hass 扩展了置信区间的计算方法。Hass 将简单边界参数和增量方法引入中心极限定理，然后将此计算方式引入了 AVG,COUNT,SUM,VARIANCE 和 STDEV 等在线聚集操作的置信度和置信区间计算[2]。

Hellerstein 等人提出的在线聚集技术是针对单表的，但是在现实情况下，聚集操作很多时候是需要多表连接的。因此，Haas 等人将在线聚集技术扩展到了多表，提出了 Ripple Join 算法[3]。Ripple Join 算法克服了传统的连接查询中两个均匀随机样本的连接操作结果形成的样本分布不均匀的问题。该算法的主要思想是不断地在各个表中选取样本，将采取的样本进行连接操作，不断迭代循环，直到

最后的置信度和置信区间符合要求。鉴于多表的链接操作会破坏样本选取的独立性，因此 Hass 等人改进了统计估计模型。Luo 等人在 Ripple Join 算法的基础上，提出了改进的 Hash ripple join 算法，通过并发采样以及并发执行多表的排序归并，对 Ripple join 算法进行了扩展，提升了 Ripple Join 算法的性能[4]。

Ripple Join 算法和 Hash Ripple Join 算法需要不断地在各个表中采集样本。当表中数据量较大，或者查询精确度要求较高时，会有大量的样本进入内存，导致内存溢出，算法性能急剧下降。基于此，Jermaine 等人提出了基于磁盘数据的连接算法，保证了内存溢出情况下连接算法的执行效率[5]。

Feifei Li 等人提出了一种新的算法，wander join 算法，来实现多表连接的在线聚集[6]。该算法通过在数据表中随机游走，来实现在线聚集的效果。同时，他们还设计了一个优化器，选择进行随机游走的最佳计划，而不必先验地收集任何统计变量。

韩希先等人分析发现现有的工作无法以既高效又满足给定的任意置信区间方式来处理近似连接聚集，因此提出了一种新的算法--（p, ε ）近似连接聚集查询来有效地返回满足任意置信区间的近似连接聚集结果[7]。

以上的研究成果都是针对单查询的，但是在实际应用中，连续多个聚集查询也是很常见了。为了优化多个聚集查询，Wu 等人设计了 COSMOS 系统。该系统将数据空间进行了划分，保存了诸多中间结果，并用一个有向无环图将多个查询组织起来，以发现多个查询结果之间的交集。通过对中间结果的有效利用，COSMOS 系统对于多个聚集查询拥有更高的效率[8]。

传统的在线聚集方法为了避免执行中随机 I/O 导致的性能下降，假设数据本身近似随机分布于数据文件中，用顺序 I/O 来代替随机 I/O。但是数据随机分布于数据文件的假设在很多实际的应用场景中是不成立的，从而导致查询结果产生很大误差。鉴于此，安明远等人提出了 DDPOA（动态分片在线聚集技术），将整个数据集分片，对各个子数据集独立计算，线性组合子集结果从而得到最终的聚集结果。该技术在完成时间相差不大的情况下，提高了在线聚集的准确度[9]。

为了充分发挥多核的计算能力，Chengjie Qin 等人设计了一个并行在线聚集

框架：PF-OLA[10]。该框架定义了一组通用接口来表示任何估计模型，并且完全抽象了执行细节。该框架还专门为并行的在线聚集设计了新的统计估计器。

近年来，随着大数据技术的兴起，对云环境下海量数据处理的性能要求越来越高，线聚集技术被引入到了分布式系统和云计算中。Wu 等人将在线聚集技术扩展到了 Peer-to-Peer 系统中。其主要思想是将查询发送到对应的节点，然后在该节点进行在线聚集操作，最后在将这些节点所产生的结果整合在一起[11]。程思瑶等人针对 Peer-to-Peer 网络中的时变数据数据，提出了一种 Peer-to-Peer 网络中基于均衡采样的时变数据近似聚集算法[12]。针对云环境下的在线聚集研究基本是针对 Count、Sum 等聚集函数，尚未有对 Max/Min 在线聚集的研究的情况，汪凤鸣等人利用切比雪夫不等式和中心极限定理，通过分位数来衡量 Max/Min 在线聚集的精确度[13]。该方法能够很好的适应大数据环境下的在线聚集，并具有良好的扩展性。

然而，上述系统均采用中心极限定理的近似估计方法，只能对聚集查询和部分统计操作做出近似估计。Laptev 等人设计了 EARL 系统，该系统采用了 bootstrap 方式实现了对任意查询函数的近似估计，增加了在线聚集的灵活性和实用性[14]。

在大数据分析处理方面，Google 提出了 MapReduce 并行编程模型[15]，它将计算作业分解成 Map 和 Reduce 两个阶段，每个阶段可以划分成大量子任务并行执行。然而知己应用中仍然存在诸多问题难以抽象为 MapReduce 模型加以求解。为此，Isard 等人提出了 Dryad 框架，用有向无环图形式定义数据处理流程，增强了并行框架的灵活性和适应性[16]。上述数据处理引擎适合大数据批处理应用，但是无法有效支持实时大数据分析处理。为此，Berkeley 提出了基于内存的分布式数据处理引擎 Spark[17]。Spark 拥有 MapReduce 的优点，但是，Spark 中间数据不再写入 HDFS 而是可以直接保存于内存中，通过 RDD 机制实现对内存数据的有效管理，提升了性能[18]。

上述模型或引擎都是计算出了精确的结果，本文所实现的系统与上述系统的主要不同是本系统将返回一个模糊的查询结果。

1.3 本文工作与贡献

本文研究并实现了基于模糊查询的大数据分析处理系统。该系统支持用户进行各种聚集查询（Group By），同时，该系统对聚集维度的取值数量会有限制。通过对多查询的在线聚集技术所做的优化，该系统可以在秒级内返回几十 G 数据的查询结果。

总体来说，本文的主要工作和贡献如下：

1) 本文设计了一套数据访问接口，其中主要包括一些用户查询操作。通过这些查询操作，用户可以自由地探索高维数据。

2) 本文提出了一种树形结构优化查询过程。鉴于查询之间，尤其是相邻查询之间，往往是有交集的，因此，本文把以前查询检索到的样本和产生的中间结果存储在该树形结构中。随着查询不断进入系统，该树形结构的节点会逐渐分裂，层次也会随之改变。

3) 基于上述树形结构，本文实现了用户接口的查询过程。该查询过程主要包含两个部分，一是在树形结构中的查询，二是在数据源中的查询。在树形结构中查询的主要目的是找到已经在系统中的符合当前查询条件的样本或者中间结果，然后复用这些样本或中间结果；当在树形结构中查询出的结果不能满足用户要求时，需要在数据源中查询。同时，本文还研究了整合多个中间结果的算法，以生成返回给用户的最终结果。

4) 通过实验，验证了本文所提出并实现的大数据分析处理系统的有效性。首先，验证了该系统的查询处理时间和数据量大小无关，而是和数据的分布有关。然后，验证了本文所提出的树形结构的有效性。最后，通过实验验证了返回给用户的置信度和置信区间的准确性。

1.4 论文结构

本文共分七章，论文的组织结构如下：

本章是论文的第一章，介绍了本文的研究背景、研究意义、研究目标，以及针对研究目标本文所做的工作和贡献。

第二章介绍了本文的相关技术，主要包括在线聚集过程，在线聚集的统计估计模型，多表在线聚集。

第三章介绍了本系统的架构，主要包括随机混淆模块、用户查询模块、样本管理模块、查询引擎模块和统计估计模块。

第四章介绍了系统中的样本管理模块。该模块主要是用一棵样本管理树来实现的，同时，该章节中还介绍了一些对于该树的基本操作。

第五章介绍了系统中的查询引擎模块和统计估计模块。查询引擎模块介绍了如何在样本管理树和数据源中进行查询操作。统计估计模块介绍了如何生成最终的统计估计值。

第六章是实验部分。通过实验验证了本系统的有效性。

第七章是对全文的总结，并对未来工作的展望。

1.5 本章小结

本章从研究背景出发，叙述了本课题的研究意义，介绍了本系统的优势。

第2章 在线聚集相关技术

本文中的模糊查询主要是利用在线聚集技术实现的，因此，本章主要介绍有关在线聚集方面的技术。2.1 节介绍了在线聚集技术的基本原则和过程；2.2 节介绍了在线聚集的统计估计模型；2.3 节介绍了多表的在线聚集过程。

2.1 在线聚集的基本描述

在数据分析中，用户经常需要对数据仓库中存放的海量多维数据进行在线分组聚集。如在一个或多个维度上进行含有 Group By 的 SUM 计算，在结果中提炼出有用的信息，然后基于这些信息做进一步的分析。以传统的黑盒模式响应形如 SUM、COUNT、AVERAGE 等的分组聚集查询时，首先要扫描整个源数据集，按 Group By 中的维值对所有数据进行排序或散列，再对排序结果分组，然后顺序处理每个分组的聚集计算，所有数据处理完毕后返回结果。因为整个过程需要大量的磁盘 I/O 和相当长的处理时间，导致响应速度迟缓，所以不适合用于在线分析[19]。

在实际应用中，用户不希望响应时间过长，使用聚集查询主要是为了尽快获得数据集的整体“轮廓”信息，发现其中“感兴趣”的方面作为进一步分析的对象，有时候不必得到准确结果。这种系统快速响应的需求可以通过近似查询处理来实现，通过查询少量的数据获得近似准确的结果，是近似查询处理的基本特点[20]。

在线聚集处理方案，通过扩展常规数据库系统，在查询执行的过程中对数据进行动态抽样，动态计算出当前的近似结果和置信区间，查询结果的误差会随着抽样数量的增加而逐渐减小。其特点是用户可以直接控制查询的执行，交互性能好，结果反馈的速度快，缺点是动态处理的代价较高，完成查询的总时间较长[21]。联机查询处理方案一边处理数据一边反馈结果，响应速度更快，且在实际近似查询的应用中，对于数据量较大的源数据集，用户很少需要完成查询，一般借助于联机查询处理丰富的交互界面，控制查询的执行速度，在查询结束之前就可以提

炼出有用信息并提前终止查询。

2.1.1 在线聚集基本原则

在线聚集的基本思想是不断在源数据集中产生随机的样本，然后用这些样本产生一个估计量，采集的样本越多，产生的估计量也就越精确。对于每个估计量，可以使用一个置信度 c 和误差 ε 来衡量其准确度。假设精准的聚合结果是 v，系统计算出的估计量是 $\bar{v}$，那么有 $P(|v-\bar{v}|<\varepsilon) \geq c$，即，估计量和精确值的结果之差的绝对值小于 ε 的概率要大于置信度 c[22]。

下面以一个典型的聚集查询为例，来说明在线聚集。在数据分析中，我们经常会执行形如下面的聚集语句：

SELECT op(expression(x_i)) FROM T

为了处理该查询，我们在数据表 T 中随机抽取了 k 个样本。我们用 S 来表示样本集，那么近似估计量就可以按照如下方式计算：

1. $\bar{v} = \frac{|T|}{k} \sum_{\forall t_i \in S} c(\exp ression(t_i))$，其中|T|是表 T 的大小。当 op 为 count 时，c(x)=1；当 op 为 sum 时，c(x)=x。

2.当 op 为 avg 时，$\bar{v} = \frac{1}{k} \sum_{\forall t_i \in S} \exp erssion(t_i)$。

表 2.1 学生成绩示例表

Class	Student_id	Score
1	100	87
1	101	68
1	102	97
1	103	88
1	104	77

表 2.1 是一张学生成绩表。表中各学生的平均分是 83.4 分，总分是 417 分。如果我们随机的选取了学号为 100,101,103 的学生，那么我们可以估计出全部学

生的平均分是(87+68+88)/3=81，总分是 81*5=405。并且，随着采取更多的样本，我们得到的估计量也会越来越准确。

图 2.1 展示了在线聚集系统的用户界面。假设一张表中存储了许多学生多门课的成绩，我们想要计算出每门课程的平均成绩。在线聚集将会为我们返回每门课程平均分的估计值和对应的置信度和置信区间，并且可以用进度条展示当前采样的进度。在图 2.1 中，可以看到，系统现在检索了表中 7%的数据作为样本，据此计算出课程 1 的平均分有 90%的可能性落在区间[70.2-1.8，70.2+1.8]内。如果用户对这个结果感到满意，那么用户可以停止课程 1 的聚集过程，这样系统就不会再计算课程 1 的分组。当所有的分组都结束时，该查询也就结束了。

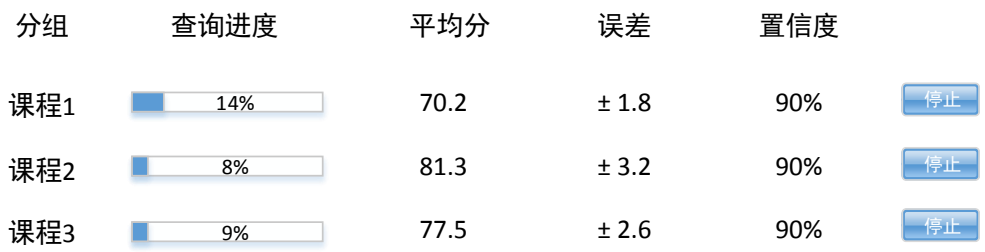

图 2.1 在线聚集用户界面

2.1.2 在线聚集的基本过程

在线聚集的基本过程如图 2.2 所示。给定系统一个查询，系统将会在数据源中随机采样。这些样本被送入查询处理器，来产生结果估计量。同时，这些样本也会被统计分析模型用到，该模型在这些样本上应用中心极限定理（或者霍夫丁不等式，等等），产生置信度和置信区间。结果估计量、置信度和置信区间将会以图 2.1 的方式实时展现给用户。当用户对现在的查询结果满意时，可以终止该组查询。当所有组对应的查询都被终止，则该查询完成。在线聚集的精确度有赖于采样的方式和统计分析模型的有效性[23]。

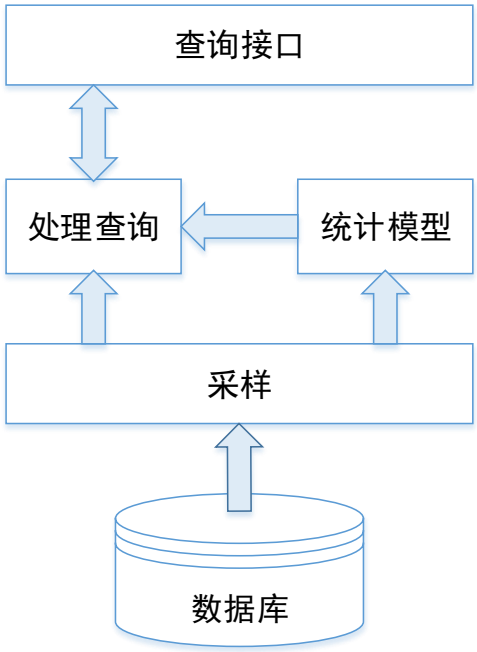

图 2.2 在线聚集基本过程

2.2 统计分析模型

统计模型用来估计近似结果的置信度和误差范围，是在线聚集技术中十分重要的一个部分。

2.2.1 置信区间类型

在统计学中，一个概率样本的置信区间（Confidence interval），是对这个样本的某个总体参数的区间估计。置信区间展现的是，这个总体参数的真实值有一定概率落在与该测量结果有关的某对应区间。置信区间给出的是，声称总体参数的真实值在测量值的区间所具有的可信程度，即前面所要求的“一定概率”。这个概率被称为置信度。

假设在数据集中随机采取了 n 个样本，并且根据这 n 个样本计算出了聚集估计值Y_n，并且正确的聚集结果是 u，对于一个预先指定的置信度$p \in (0,1)$，我们需要得到一个准确的参数ε_n，使正确结果 u 落在区间$[Y_n-\varepsilon_n, Y_n+\varepsilon_n]$内的概率为 p。$[Y_n-\varepsilon_n, Y_n+\varepsilon_n]$则为置信区间，p 为置信度。当$\varepsilon_n$比较大时，表示当前所收集到的样本不够多，因此聚集估计量Y_n距离正确值 u 的误差也比较大。

在 Hellerstein 等人的论文中，提出了三种置信度类型，分别是保守的置信区间（conservative confidence）,大样本置信区间（large-sample confidence）和确定性置信区间（determinstic confidence）[1]。下面分别对三种置信区间进行介绍。

保守的置信区间是指置信区间$[Y_n-\varepsilon_n, Y_n+\varepsilon_n]$包含正确结果 u 的概率大于等于概率 p。这样的区间可以通过 Hoeffding 不等式或者该不等式的扩展不等式计算出来，并且对于所有的$n\geq 1$都是成立的[24][25]。

大样本置信区间是指置信区间$[Y_n-\varepsilon_n, Y_n+\varepsilon_n]$包含正确结果 u 的概率等于概率 p。该区间是基于中心极限定理计算出来的。该置信区间的优势是无论 n 的大小，中心极限定理都成立，因此估计效果比较好。但是在应用大样本置信区间时需要注意的是，大样本置信区间$[Y_n-\varepsilon_n, Y_n+\varepsilon_n]$包含准确结果 u 的概率可能会比名义上的置信度 p 小。但是该置信区间在实际应用中的一个优势是一般会比保守的置信区间短。基于此优势，大样本置信区间在在线聚集中应用比较广泛。

确定性置信区间是指置信区间$[Y_n-\varepsilon_n, Y_n+\varepsilon_n]$包含正确结果 u 的概率为 1。这种置信区间只有在 n 很大时才有有用。因此，该置信区间很少应用到在线聚集系统。

2.2.2 置信区间的计算

假设 R 是一个包含了 m 个元组的数据表，用$t_1, t_2, \ldots, t_m$来代表对应的 m 个元组。现在考虑形式如下的查询：

SELECT AVG(expression) FROM R;

其中 expression 是涉及 R 的属性的算术表达式。用$v(i)(1\leq i\leq m)$表示算术表达式 expression 作用在元组t_i时得到的值。L_i表示在表 R 中随机检索出的元组t_i的索引，即第 i 个在表 R 中检索出的元组时t_{L_i}。假设所有元组在每次检索中都会等概率的被检索到，因此，$P\{L_i=1\}=P\{L_i=2\}=\ldots=P\{L_i=m\}=1/m$对所有的 i 都成立。在 n（$1\leq n\leq m$）个元组被检索出来后，可以得到聚集的近似结果：

$\overline{Y_n} = (1/n)\sum_{i=1}^{n} v(L_i)$。

为了得到一个保守的置信区间，需要预先得到两个常量 a 和 b，并且对于 $1 \le i \le m$ 有 $a \le v(i) \le b$。这两个常量可以在在线聚集过程开始之前离线地计算出来。用 u 来表示该查询的准确结果，即 $u = (1/m)\sum_{i=1}^{m} v(i)$。Hoeffding 不等式如下：

$$P\{|\overline{Y_n} - u| < \varepsilon_n\} > 1 - 2e^{-2n\varepsilon_n^2/(b-a)^2} \qquad \text{公式(2.1)}$$

让公式 2.1 的右侧等于我们预设的置信度 p，就可以解出 ε_n。这样，$\left[\overline{Y_n} - \varepsilon_n, \overline{Y_n} + \varepsilon_n\right]$ 包含正确结果 u 的概率就大于等于概率 p。解出的 ε_n 如下：

$$\varepsilon_n = (b-a)\left(\frac{1}{2n}\ln\left(\frac{2}{1-p}\right)\right)^{1/2} \qquad \text{公式(2.2)}$$

大样本置信区间的计算需要用到中心极限定理，如下是中心极限定理：

中心极限定理：设随机变量 $X_1, X_2, \ldots, X_n$ 互相独立，具有相同的分布，$E(X_k) = u, D(X_k) = \delta^2 > 0 (1 \le k \le n)$，记 $Y_n = \dfrac{\sum_{k=1}^{n} X_k - nu}{\sqrt{n\delta^2}} = \dfrac{\overline{X} - u}{\delta/\sqrt{n}}$，则对于任意实数 x，有 $\lim_{n\to\infty} P(Y_n \le x) = \dfrac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt = \phi(x)$ [26]。

由上述定理可知，均值为 u，方差为 $\delta^2 > 0$ 的独立同分布的随机变量 $X_1, X_2, \ldots, X_n$ 的和 $\sum_{k=1}^{n} X_k$ 的标准化变量的分布函数，当 n 充分大时，有：

$$Y_n = \frac{\sum_{k=1}^{n} X_k - nu}{\sqrt{n\delta^2}} = \frac{\overline{X} - u}{\delta/\sqrt{n}} \sim N(0,1) \text{ 或者 } \overline{X} \sim N(u, \delta^2/n) \qquad \text{公式(2.3)}$$

计算大样本置信区间时，我们不需要预先得到任何常量。因为 n 个样本都是在表 R 中随机检索得到的，因此，可以认为 $\{L_i : 1 \le i \le n\}$ 是独立同分布的变量。算术表达式 expression 作用在表 R 中每个元组，所有元组得到的值 v(i)得方差为：

$\delta^2 = (1/m)\sum_{i=1}^{m}(v(i)-u)^2$。当 n 足够大时，这些样本符合中心极限定理。因此，变量$\sqrt{n}(\overline{Y_n}-u)/\delta$符合均值为 0，标准差为 1 的标准正态分布。$\delta^2$是表中所有元组计算出的标准差，可以用样本的方差对其进行替代。样本的方差为：$T_{n,2}(v) = (n-1)^{-1}\sum_{i=1}^{n}(v(L_i)-Y_n)^2$。因此：

$$P\{|Y_n-u|\le\varepsilon_n\} = P\{|\frac{\sqrt{n}(Y_n-u)}{T_{n,2}^{1/2}(v)}|\le\frac{\varepsilon_n\sqrt{n}}{T_{n,2}^{1/2}(v)}\}\approx 2\phi(\frac{\varepsilon_n\sqrt{n}}{T_{n,2}^{1/2}(v)})-1 \qquad \text{公式(2.4)}$$

$P\{|Y_n-u|\le\varepsilon_n\}$即是我们预设的概率 p，这样，就可以解出$\varepsilon_n$。

2.3 多表聚集

上面所做的研究都是关注于单表的聚集，但是在实际应用中，聚集操作常常发生在多个表的连接操作后。因此，我们需要扩展单表聚集模型，来处理设计多个表的聚集查询。考虑形式如下的查询：

SELECT $op(\exp ression(x_i))$ FROM $T_1,\ldots,T_k$ WHERE $predicate$

为了处理此查询，我们可以从涉及到的各个表中检索样本，然后对这些样本执行连接操作。然而，给定从表$T_1,\ldots,T_k$检索出的$t_1,\ldots,t_k$，有很大的可能性是，这些样本连接在一起不能产生一个有效结果。因此，在线聚集也会因为没有足够多的有效样本而失效[27]。为了解决上述问题，有多种针对在线聚集的表连接算法被提了出来，其中应用最为广泛的是 ripple join 算法。

Ripple Join 算法和嵌套循环连接算法很类似，都是非阻塞的。Ripple Join 的主要思想是迭代地从各个表中检索数据。一个样本被检索出来后，立即将此样本与其他表中的样本进行连接操作[28]。下面以两个表T_1和T_2的连接操作为例，说明 Ripple Join 的过程。同样的方法可以很容易地扩展到更多张表的连接操作。

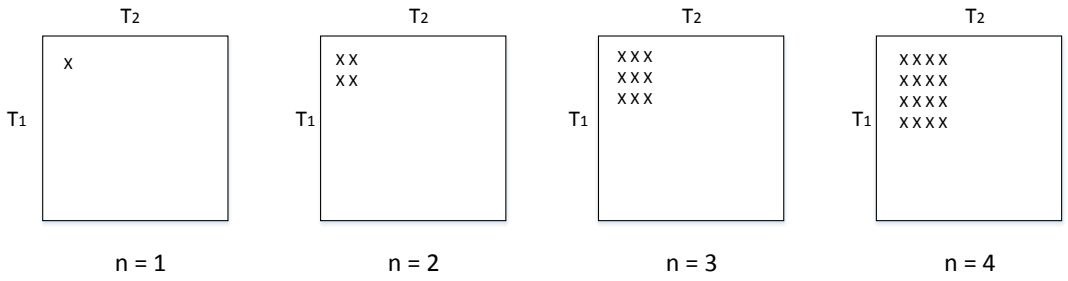

图 2.3 方形 Ripple Join 算法过程

图 2.3 展示了 Ripple Join 算法的的过程。我们用一个矩阵来表示在每个表中采样的进程。在图 2.3 所示的矩阵中，T_1 轴表示从表 T_1 中检索出的样本，T_2 轴表示从表 T_2 中检索出的样本，矩阵中每个位置（t_1，t_2）代表了 $T_1 \times T_2$ 中的一个元组，矩阵中的每个“x”代表了一个现在已经采样的 $T_1 \times T_2$ 对应的一个元素。在图 2.3 中，我们以相同的速率对表 T_1 和 T_2 进行采样。即，在 k 步之后，我们在表 T_1 和 T_2 中都检索出了 k 个样本。通过这些样本可以产生 k^2 个 T_1 和 T_2 笛卡尔积中的结果，这 k^2 个结果中有一部分是有效的，可以被在线聚集使用。

在 Ripple Join 算法中，一旦我们从一个表中检索出一个新的样本，就可以将这个样本与另一张表中检索出的所有样本进行连接。为了这个目的，有一张表中所有检索出的样本都必须被缓存在内存中[29]。因此，Ripple Join 的一个瓶颈就是内存限制。当内存不足，不能够缓存所有样本时，Ripple Join 就会转为正常的嵌套循环连接。

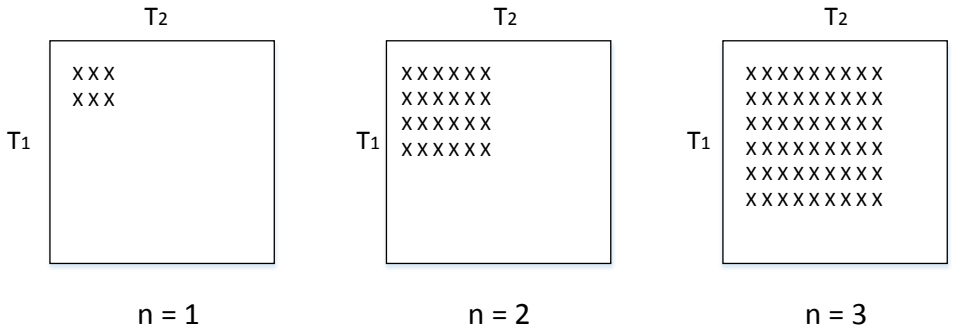

图 2.4 矩形 Ripple Join 算法过程

Ripple Join 的一个优点是可以自适应地调整采样过程。我们可以从某张表中

检索出比其他表多的元组，来提供一个更精准的置信区间。例如，假设每个表T_2中的元组可以与 3 个表T_1中的元组进行连接，但是每个表T_1中的元组只能与 2 个表T_2中的元组进行连接。那么，我们可以将表T_1与表T_2的采样速率比设为 2:3，这样可以产生更多的有效连接结果。图 2.4 展示了此过程。在 Ripple Join 的处理过程中，我们可以基于观察到的数据分布来自适应地改变所涉及的表的采样率[30]。

在 Ripple Join 中，我们使用上一节中的中心极限定理来产生置信区间。但是，与单表不同，当涉及到多张表时，上一节中的模型需要修改。假设$x_i \in T_1$和$y_i \in T_2$，并且(x_i, y_i)是表T_1和表T_2连接后的一个样本。在检索了足够多的样本后，可以假设估计量遵从平均值为准确结果 u 的正态分布。

首先考虑 sum 和 count 查询。对于$x_i \in T_1$，我们定义$u(x_i, T_1)$为：

$$u(x_i, T_1) = \sum_{\forall y_i \in T_2} |T_1| \,\text{expression}(x_i, y_i) \qquad \text{公式(2.5)}$$

$|T_1|$和$|T_2|$分别代表从表T_1和表T_2中检索出的样本的数量，$\text{expression}(x_i, y_i)$代表了$x_i$和$y_i$连接之后的聚集操作（如果$x_i$和$y_i$不能连接，或者连接结果不满足 where 条件，那么$\text{expression}(x_i, y_i)$=0）。则，$u(x_i, T_1)$对于所有的$x_i \in T_1$的平均值就是最终的结果 u。令$\delta_x^2$表示$T_1$中满足连接操作的样本的平方差，则有：

$$\delta_x^2 = \frac{\sum_{\forall x_i \in T_1} (u(x_i, T_1) - u)^2}{|T_1|} \qquad \text{公式(2.6)}$$

类似的，对于表T_2，我们可以计算出$u(y_j, T_2)$和δ_y^2，接下来，可以计算出两张表连接之后的方差δ^2：

$$\delta^2 = \frac{\delta_x^2}{|T_1|} + \frac{\delta_y^2}{|T_2|} \qquad \text{公式(2.7)}$$

在线聚集过程中，假设 X 和 Y 是分别是在表T_1和表T_2中检索出的样本集。用 X 和 Y 来代替上述公式中的T_1和T_2。可以得到：

$$\overline{\delta_x^{\,2}} = \frac{\sum_{\forall x_i \in X} (u(x_i, X) - \bar{u})^2}{|X|}, \quad \overline{\delta^2} = \frac{\overline{\delta_x^{\,2}}}{|X|} + \frac{\overline{\delta_y^{\,2}}}{|Y|} \qquad \text{公式(2.8)}$$

上面$\overline{\delta_x^{\,2}}$中的$\bar{u}$表示样本集中聚集结果的估计量。应用中心极限定理，可以计算出置信度对应的置信区间。

相比 sum 和 count 查询，对 avg 查询结果的估计比较复杂。avg 查询可以看做是 count 和 sum 的一个组合查询（sum 除以 count）。给定$x_i \in T_1$和$y_i \in T_2$，如果x_i和y_i可以连接，并且连接结果满足 where 条件，则令$f(x_i, y_j)$返回 1，否则，令$f(x_i, y_j)$返回 0。给定一个表T_1中的元组x_i，可以定义如下的函数$u'(x_i, T_1)$：

$$u'(x_i, T_1) = \sum_{\forall y_j \in T_2} |T_1| f(x_i, y_j) \qquad \text{公式(2.9)}$$

为了估计平均值，我们需要计算出$u(x_i, T_1)$和$u'(x_i, T_1)$的协方差。用x_{avg}和x'_{avg}分别表示$u(x_i, T_1)$和$u'(x_i, T_1)$的均值。那么协方差如下：

$$\gamma(x_i) = \frac{1}{|T_1|} \sum_{\forall x_i \in T_1} (u(x_i, T_1) - x_{avg})(u'(x_i, T_1) - x'_{avg}) \qquad \text{公式(2.10)}$$

最后，我们可以计算出连接结果的协方差：

$$\delta^2 = \frac{1}{u_c^{\,2}} (\delta_s^{\,2} - 2u\gamma + u^2 \delta_c^{\,2}) \qquad \text{公式(2.11)}$$

在上面的式子中，δ_s和δ_c分别代表 count 和 sum 的对应的结果，$\mathrm{u} = \frac{u_s}{u_c}$，$\delta_c^{\,2}$和$\delta_s^{\,2}$分别是 count 和 sum 对应的方差。在计算中，我们用样本集 X 和 Y 来代替表T_1和表T_2。计算出方差后，就可以对连接结果应用中心极限定理，估计其聚集结果[31]。

Ripple Join 的一个缺点是对内存的使用量比较大。当内存不足时，Ripple Join 会退化为嵌套循环连接，这会导致比较严重的性能下降。Luo 等人采用了并行的基于哈希的 Ripple Join 来解决这个问题，其基本思想是在处理器之间划分元组，每个处理器可以单独处理 Ripple Join，通过整合所有估值获得最终结果。在此基

础上，Jermaine 等人提出了一种基于磁盘的 Ripple Join 算法来解决内存不足的问题。基于磁盘的 Ripple Join 算法有多个阶段。每个阶段内包含了一些列基于哈希的 Ripple Join 过程。该算法进一步提高了多表在线聚集的效率。

2.4 本章小结

本章介绍了在线聚集技术的相关研究工作和相关技术。第一节描述了在线聚集技术的基本原则和过程，第二节详细介绍了在线聚集技术中的统计分析模型，第三章介绍了多表的在线聚集技术。通过总结前人的研究成果，分析存在的不足，为后续研究奠定基础。

第3章 基于模糊查询的大数据分析处理系统架构

本章节详细介绍基于模糊查询的大数据分析处理系统架构。该系统主要是由以下几个部分组成的：随机混淆模块，用户查询模块，样本管理模块，查询引擎模块以及统计估计模块。

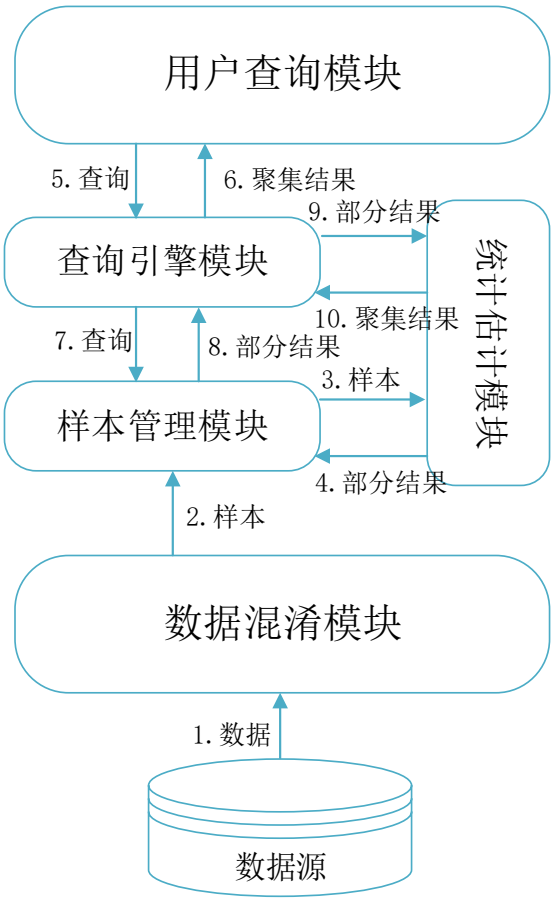

图 3.1 系统模块之间的关系

图 3.1 展示了系统中各个模块之间的关联。数据混淆模块从数据源中读取数据（1），然后对数据进行随机化，生成随机数据集。样本管理模块从数据混淆模块生成的随机数据集里读取数据作为样本（2）。样本管理模块向统计估计模块输入样本（3），经过统计估计模块计算后，返回统计估计量（4）。用户查询模块向用户提供查询接口，向查询引擎传递查询请求（5），并且接收系统产生的最终的

聚集结果（6），查询引擎模块通过在样本管理模块中进行查询（7），得到相应的统计估计量（8），然后与统计估计模块进行交互，将多个统计估计量（9）整合为最终结果（10），最终将该结果返回给用户。

3.1 随机混淆模块

给定一个数据集，随机混淆模块的作用是根据原始数据集生成一个随机样本集。在传统的采样方式中，一般是随机地在数据集中检索样本。但是在该系统的实现中，我们采取了将数据随机存储成随机数据集的方式。这样，顺序地扫描数据集，就可以得到随机数据流。

之所以将原始数据集存储为随机数据集，主要是源于以下几个方面的原因。

一是如果按照原始的随机采样方式，当我们访问一个页时，只能在该页中得到一个样本。但是如果在随机数据集中采样，我们访问一个页时，该页中存储的所有数据都可以作为随机样本。这意味着访问过 k 个页后，顺序扫描随机数据集比随机采样得到的样本数量更多，因此也会产生更小的置信区间和更高的置信度。

二是采用随机数据集可以有效的消除随机 I/O，可以按照顺序来扫面随机数据集。这意味着采用随机数据集可以大大的减小 I/O 开销。在应用中，采用随机数据集顺序扫描获取随机样本的方式要比随机访问数据集中的元素快几个数量级。随机数据集顺序扫描每秒可以获取 532,000 个样本，而随机访问数据集每秒只能得到 316 个样本[32]。

三是当用户设置的查询精度较高时，需要系统扫描整个集，采取顺序扫描随机样本的方式，可以很方便的扫描整个数据集。如果采取随机扫描数据集的方式，很可能会采到重复的样本，而这是不被允许的。去重操作会增加时间和空间的开销，进一步拖慢了该方式的速度。

为了对数据集进行随机化，我们采取了一个简单的策略：首先顺序地扫描文件，当我们扫描文件中每个元组时，我们将该元组放置在随机数据集中一个随机选取的位置。这样，当初始文件扫描结束时，就得到了一个随机数据集。这个随机数据集将会作为在线聚集系统的数据源。

对于涉及到多个数据表的查询，我们需要为连接的结果生成一个随机数据集。可以采取两种方式来生成该随机数据及。一是可以预先计算出两个表的连接结果，然后将该结果随机化；二是采用 Ripple Join 的方式，先对两个表进行随机化，然后迭代地连接两张表。第一种方案为了效率和准确性而牺牲了存储，而第二种方案则为了灵活性牺牲了效率和准确性。

在系统的实现中，我们采取了一种类似于 Ripple Join 连接的方式。为了得到两张表连接后的随机数据集，我们选择一张表作为外表（R），另一张表作为内表（S）。我们按照单表的方式将 R 进行随机化，然后顺序地扫描 R 对应的随机数据集。对于从 R 的随机数据集中扫描出的每个元组，我们查找 S 表的索引，找到对应的元组进行连接。这样就可以产生两张表连接后的随机数据集。

3.2 用户查询模块

在该模块中，我们定义了五种类型的操作。系统开放这五种类型的接口给用户，以便用户探索数据。数据表中的维度被分为了两种，一种是离散型维度，一种是连续型维度，具体的定义将在第四章中介绍。系统会对查询进行一些限制。在查询对应的 SQL 语句中，avg 等聚集函数内出现的只能是连续型维度，where 中的限定条件可以是连续型或者离散型维度，group by 语句后面只能是离散型维度。系统提供的五种类型的查询操作都是只能修改上一次查询对应的 SQL 语句的某个部分。下面以 TPC-H 中 LINEITEM 表为例，说明这五种类型：

第一种是更改选择维度操作。假设系统上一个操作对应查询 select avg(discount) from LINEITEM group by ReturnFlag；更改选择维度，顾名思义，就是将上述 SQL 语句中的 discount 维度换为其他维度，比如：select avg(tax) from LINEITEM group by ReturnFlag。

第二种是下钻操作。假设前一次操作对应的查询是 select avg(discount) from LINEITEM group by ReturnFlag；那么下钻就是在该查询的基础上，将上次的聚合维度固定在某个值上，然后对另一个维度做聚合查询。比如将 ReturnFlag 固定在值‘R’上，然后对维度 LineStatus 聚集：select avg(discount)

from LINEITEM where ReturnFlag = ‘R’ group by LineStatus。

第三种类型是更改条件操作。可以更改最近设置的条件（连续型和离散型）。比如，上一次的操作是：select avg(discount) from LINEITEM where ReturnFlag = ‘R’ group by LineStatus。那么可以更改其限定条件ReturnFlag，比如，select avg(discount) from LINEITEM where ReturnFlag = ‘0’ group by LineStatus。连续型条件的更改也同理。值得注意的是，限定用户只能更改最近设置的条件。

第四种类型是更换聚集维度操作。如果上一个查询是：select avg(discount) from LINEITEM group by LineStatus。如果ReturnFlag也是离散型维度，那么，可以将LineStatus替换为ReturnFlag，即查询变为select avg(discount) from LINEITEM group by ReturnFlag。

最后一种类型是上卷操作。上卷操作指的是去掉最近增加的where条件。假设在select avg(discount) from LINEITEM group by ReturnFlag的上面做了一次下钻操作，即：select avg(discount) from LINEITEM where ReturnFlag=‘0’ group by LineStatus，然后又增加了一个查询条件：select avg(discount) from LINEITEM where ReturnFlag=‘0’and Quantity <= 30 group by LineStatus。由于Quantity是最近增加的条件，在做上卷操作时，可以去掉该条件。但是，出于第四章内存树结构方面的考虑，上卷操作只可以去掉最近添加的限制条件。

本系统中的查询类似于SQL查询，但是，与SQL查询不同的是，查询中增添了两个关键字：Confidence和ErrorBound。和2.1.1中的用户界面有所不同，本系统需要用户预先设置置信度和置信区间。

假设在数据集中随机采取了n个样本，并且根据这n个样本计算出了聚集估计值Y_n，并且正确的聚集结果是u，对于一个预先指定的置信度$p\in(0,1)$，我们计算出ε_n，使正确结果u落在区间$[Y_n-\varepsilon_n,Y_n+\varepsilon_n]$内的概率为p。$[Y_n-\varepsilon_n,Y_n+\varepsilon_n]$则为置信区间，p为置信度。Confidence设置的是置信度。但是由于在查询之前，用户并不知道最终的查询结果，因此也就没有办法给出一个确切的置信区间。因

此，本文使用ErrorBound（$ErrorBound = \frac{\varepsilon_n}{Y_n}$）来衡量估计值的范围，这样，可以得到预设的置信区间为$[(1-ErrorBound)*Y_n,(1+ErrorBound)*Y_n]$。

假设如下是某次查询操作所对应的SQL语句：SELECT avg(quantity) FROM lineitem WHERE discount < 0.05 GROUP BY shipmode CONFIDENCE 0.99 ERRORBOUND 0.01，该查询的目的是计算出每种运输方式中折扣小于5%的商品的总量的近似值Y_n，并且真实结果u以99%的概率落在区间[(1 0.01)*,]内。

3.3 样本管理模块

系统的一个主要的部分是样本管理模块。该模块通过一棵树来管理以前查询所检索到的样本和产生的中间结果。我们姑且称这棵树为样本管理树。这棵树最开始只有一个子节点，随着系统不断接收到查询请求，这棵树也会不断改变形态。该树的主要操作有节点分裂，节点合并，层次转换等。本文将在第四章进行具体介绍。

之所以构建样本管理树，最重要的原因在于，对于在3.2节中定义的五种查询操作，相邻的两个查询操作所查找出的数据集都是有交叠的。当前查询操作是可以利用以前查询所产生的结果的，尤其是上一次的查询操作的结果。Group by操作的过程是将一个数据集按照某个维度的值划分为多个子集的过程，这个过程和树分裂的过程是十分类似的。因此，我们选择用一棵树来保存在线聚集的样本和中间结果。随着查询的进行，获取的样本和中间结果越来越多，树也会分裂出越来越多的节点。

例如，有如下的三个查询（使用了TPC-H中的LINEITEM表作为实例）：Q1是初始查询，Q2在Q1的基础上做了一个下钻操作，Q3在Q2的基础上做了一个上卷操作。

Q1 select avg(discount) from lineitem group by returnflag;

Q2 select avg(discount) from lineitem where returnflag= ‘R’ group

by linestatus;

Q3 select avg(discount) from lineitem group by linestatus;

假设这些查询按照 Q1,Q2,Q3 的顺序执行，那么尽管 Q2 查询所需要的数据集是 Q1 查询所需数据集的一个子集，但是如果只是单独地为 Q1 查询和 Q2 查询分别采样，计算统计估计量，那么没有办法让 Q2 查询复用 Q1 查询采集的样本或计算出的统计估计量。因此，我们注意到，如果在处理 Q1 查询时，将其采集到的样本按照某种方式组织起来（根据 returnflag 属性进行划分），那么在处理 Q2 查询时就可以复用 Q1 查询中已经获取的样本（returnflag 属性等于 r 的那部分）。同样地，我们可以复用 Q3 查询，Q2 查询的产生的结果。但是注意 Q3 查询的情况可能会比较复杂，Q3 查询可以同时复用 Q1 查询和 Q2 查询的结果。如何将 Q1 查询和 Q2 查询的结果组合到一起，供 Q3 查询使用，将在下一章中介绍。

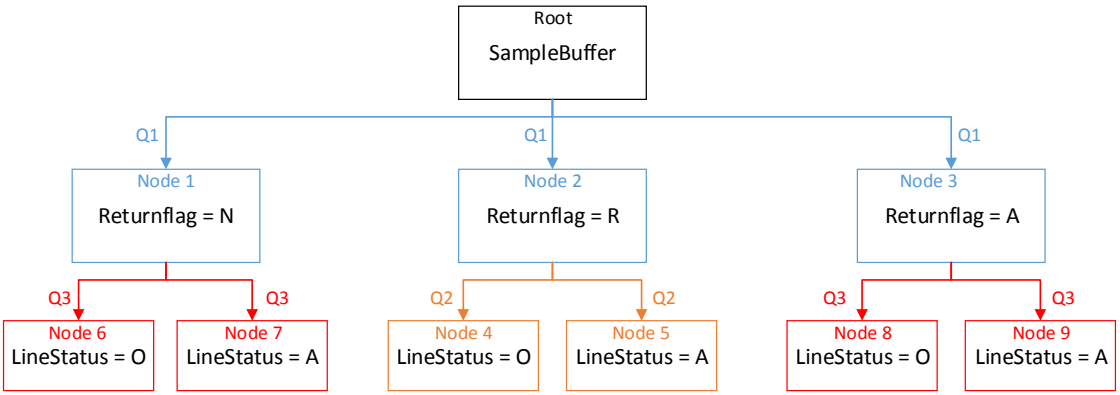

图 3.2 样本管理树示例图

图 3.2 展示了 Q1，Q2 和 Q3 查询过程中树的分裂过程和样本的复用。当 Q1 查询到来时，其聚集维度为 Returnflag，因此，我们按 Returnflag 属性的值将样本进行划分，由根节点分裂出了 Node1,Node2 和 Node3。当 Q2 查询进入系统时，首先根据其查询条件 returnflag=‘R’找到对应的根节点 Node2，然后根据 Q2 查询的聚集维度 LineStatus，对将 Node2 分裂为 Node4 和 Node5，这样，Q2 就复用了查询 Q1 在 Node2 中的样本。最后，当 Q3 进入系统时，由于第一层分裂的维度 Returnflag 并未在 Q3 的限定的 where 条件内，因此 Q3 可以复用第一层分

裂出的所有节点，即Node1，Node2和Node3，同时，Q2分裂出的Node4和Node5可以直接被Q3使用。Node1和Node3则需要继续分裂，分别分裂出Node6、Node7，Node8、Node9。最终，Q3查询需要将Node4、Node6、Node8进行合并，将Node5、Node7、Node9进行合并，以生成最终的查询结果。同时，树的形态也会对应发生改变，具体算法将在下一章介绍。

3.4 查询引擎模块

查询引擎模块需要处理用户查询模块的输入，并生成最终的聚集结果返回给用户。在样本管理树中查找满足条件的样本集，通过样本集计算出查询结果，然后与统计估计模块进行交互，输出查询结果对应的置信度和置信区间。

由于样本管理树中存储了以前查询检索的样本和产生的中间结果，因此查询引擎会首先会在这棵树中进行处理，以重复利用这些样本和结果。这个过程对于3.2 小节中提出的五种用户操作是不相同的。本文将会在第五章分别对这五种用户操作在样本管理树中的查询过程进行介绍。

但是，当用户的置信度设置的比较高，或者置信区间设置的比较窄时，查询引擎需要大量的样本来产生满足用户要求的查询结果，这时，在样本管理树中产生的结果是不能满足用户的要求的。在这种情况下，查询引擎需要直接从数据源，即随机数据集中读取样本，用符合条件的样本来更新查询结果。同时，这些新读取的样本也将会插入样本管理树中。

3.5 统计估计模块

统计估计模型主要包含两个部分，一是查询的近似结果，二是该结果在某个置信区间下的置信度。因此，统计估计模块主要有两个功能，一是根据样本生成统计估计模型，二是将多个统计估计模型整合为一个统计估计模型。

当把样本不断的插入样本管理树中的叶子节点时，我们需要利用统计估计模块的第一个功能，生成对应该节点的统计估计模型。

在样本管理树执行查询引擎，找出所有满足查询条件节点后，我们利用统计

估计模块的第二个功能，将这些节点中的统计估计模型进行整合，生成最终的结果。

3.6 本章小结

本章介绍了基于在线聚集的多维数据探索系统的架构，主要包括了五个模块：随机混淆模块，查询解析模块，查询执行模块，样本管理模块和统计估计模块。其中，随机混淆模块和查询解析模块的功能较为简单。在第四第五章中，我们将详细介绍样本的管理，相关的查询算法以及统计估计量的计算。

第4章 样本和中间结果管理

在传统的在线聚集技术中，每当处理一个新的查询，我们就需要在数据源中从头读取数据，一边读取数据一边计算相应的统计估计量。然而，在本系统中，相邻查询之间是有联系的，它们的查询结果有交集。基于这一点，本文设计出了一棵树来存储以前查询所检索出的样本和产生的中间结果，当后续查询进入系统时，我们可以利用这棵树中存储的样本和中间结果加速查询。本章中，我们将介绍这棵树的结构和基于该树的一些基本操作。

4.1 树节点介绍

我们将样本和基于样本的中间结果保存在了树节点中。树中的节点有两种类型，叶子节点与非叶子节点。

在系统刚启动时，树中只有一个根节点。当系统接收到第一个查询，会根据查询进行分裂，分裂出多个子节点。随后，根节点会在随机数据集中读取样本，新读取的样本将会根据根节点的分裂方式插入到恰当的子节点中。随着读取的样本越来越多，系统计算出的估计结果会越来越准确，置信度和置信区间也会越来越接近用户的预设值。当置信度和置信区间满足用户的要求时，根节点停止数据读取操作，向用户返回结果。同时，子节点将会记录本节点保存的所有样本的相关统计估计量，根节点则把所有子节点的统计估计量进行整合，保存在相应的域中。系统接收到下个查询时，根节点的子节点将会进一步分裂。具体的分裂过程将在下文介绍。

这样的分裂方式有两种优势：一是随着分裂的进行，我们得到了越来越多的节点，这些节点中保存着以前查询产生的中间结果，在后面的查询中，可以直接利用这些中间结果；二是即使不能直接利用节点中保存的中间结果，我们也可以从树的分裂信息快速定位到我们需要的样本，加快了检索时间。

4.2 分裂方式

为了对样本和中间结果进行组织，系统使用了样本管理树。对树中的节点进行分裂的过程就是对样本集进行划分的过程。首先需要确定的是样本集需要按哪些维度进行划分。本文选取了两种类型的列作为样本集划分的候选维度。第一种类型是出现在“Group By”语句中的维度，第二种类型是出现在“Where”语句中的维度。这两种类型的维度可以很好地将样本集划分为多个子集。出现在“Group By”语句中的维度的取值一般是离散型的，而出现在“Where”语句中的维度则既可能是离散型，也可能是连续型的。节点按照离散型和连续型维度分裂的方式有所不同。

对于离散型的维度，为了防止产生过多的子节点，本文预设了一个参数 N_g。如果该维度可取的值的个数小于 N_g，可以简单地基于对应的值来进行分裂。否则，将该维度视为连续型维度。

以 tpc-h 中的 LIENITEM 表为例，该表中的 ReturnFlag 维度是离散的，有三个取值（‘N’，‘A’，‘R’），假设参数 N_g 大于 3，当系统中第一个查询是 select avg(discount) from lineitem group by returnflag,那么根节点将会分裂出 3 个子节点，每个节点存储对应条件的样本。

对于连续型的维度，由于其取值一般在某个范围内。因此一种比较直观的分裂方式是将该维度的取值范围均匀划分。然而，这种方式最终很可能导致维度灾难。设想一张有 10 个连续型维度的表，每个维度均匀地划分为 10 个子空间，那么对应的样本管理树需要分列 10 层，每层分裂出 10 个子节点，这样共会产生 10^{10} 个子节点，子节点的数量甚至比数据集的元组数量还要多，已经失去了划分的意义。

因此，对于连续型的维度，我们采取了另一种划分方式，即根据用户的查询负载来划分，即根据系统当前已经处理过的查询对连续型维度进行划分。通过这种划分方式，可以让分裂出的节点正好对应某个查询的结果，鉴于以前出现的查询条件在以后的查询中也很有可能会出现，这样可以更好地利用节点中保存的结

果。

假设在第一个查询 select avg(discount) from lineitem group by returnflag 的后面，紧接着进行了第二个查询：select avg(discount) from lineitem where quantity < 30 group by returnflag，那么我们将会把对应的节点按照 quantity 维度进行分裂，分裂出两个节点，分别对应 quantity < 30 和 quantity>=30 的部分。

由于在随机混淆的时候，我们需要对整个数据集进行扫描，因此，在该阶段，可以统计所有维度的取值，根据其是否超过 N_g，来判断该维度是离散型的还是连续型的。只有离散型的维度才可以出现在 Group By 语句后面，即系统只支持对离散型的维度做聚集查询。

4.3 节点分裂

通过 4.2 节，我们知道了节点可以按哪些维度进行分裂。节点存在两种分裂的方式，按照离散型维度分裂和按照连续型维度分裂。本节将介绍具体的分裂过程。

4.3.1 按离散型维度分裂

样本管理树中的节点有两种形态，叶子节点和非叶子节点。我们分别介绍这两种形态的节点的分裂方式。

叶子节点按照离散型维度的分裂方式比较简单。我们只需要扫描该叶子节点中存储的所有样本，然后检查样本的分裂维度对应的取值，我们为每个取值创建一个子节点，并将样本存入对应的子节点。叶子节点完成分裂后，节点中存储的所有样本就全部下沉到了子节点，同时该节点也由叶子节点变为了非叶子节点。以 tpc-h 中的 LIENITEM 表为例，图 4.1 是一个叶子节点（根节点）按照离散型维度（ReturnFlag）分裂的一个例子。

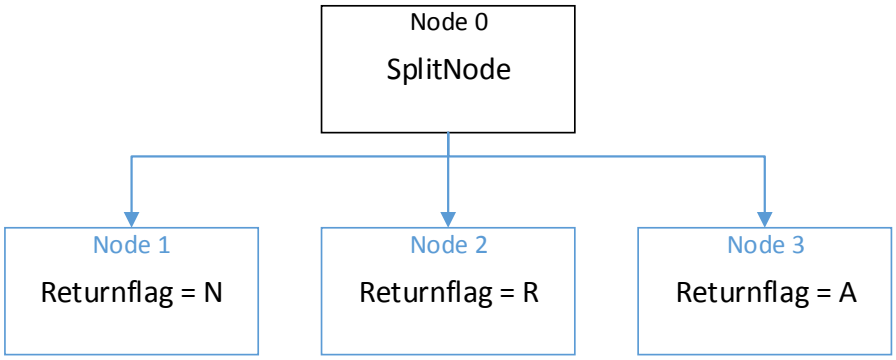

图 4.1 叶子节点按照离散型维度分裂

ReturnFlag 维度有三种取值，分别为‘N’，‘A’，‘R’，我们为每种取值创建一个节点。Root 节点中保存的样本现在全部存在了三个子节点中，Node1 中存储的是 ReturnFlag 维度值为 N 的节点，其它两个子节点也类似。每个子节点都保存了指向父节点的指针，我们在图中为了简介并未画出。

非叶子节点的分裂过程要比叶子节点麻烦。我们需要遍历该节点的所有分支，并确保所有分支都按照选定的维度分裂。具体算法如下：

Algorithm: Non-leaf Node Split by Discrete Demension
Input: TreeNode splitNode, Demension D Output: Queue result
1. queue.add(splitNode) 2. while(!isEmpty(queue)) 3. node = Remove(queue) 4. if node is leaf 5. Split(node, D) 6. if node is split by Demension D 7. Add(result, node) 8. else 9. nodelist = GetChildNodes(node) 10. Add(queue, nodelist)

```
11. return result
```

首先，将要分裂的节点加入一个队列中（1 行）。然后，对该队列进行深度优先遍历（2 行）。首先取出队列中的首节点，当该节点是叶子节点，则对该叶子节点按照维度 D 进行分裂（4-5 行），叶子节点的分裂方式我们在上面已经介绍过。如果该节点已经按维度 D 分裂过，那么把该节点加入结果队列 result 中（6-7 行），否则，将该节点的所有子节点加入 queue 中（8-10 行）。遍历完所有分支后，返回 result 队列（11 行）。这样，就可以遍历以 splitNode 为根节点的树的所有分支，并且把所有分支按维度 D 分裂的节点存储在了队列 result 中。

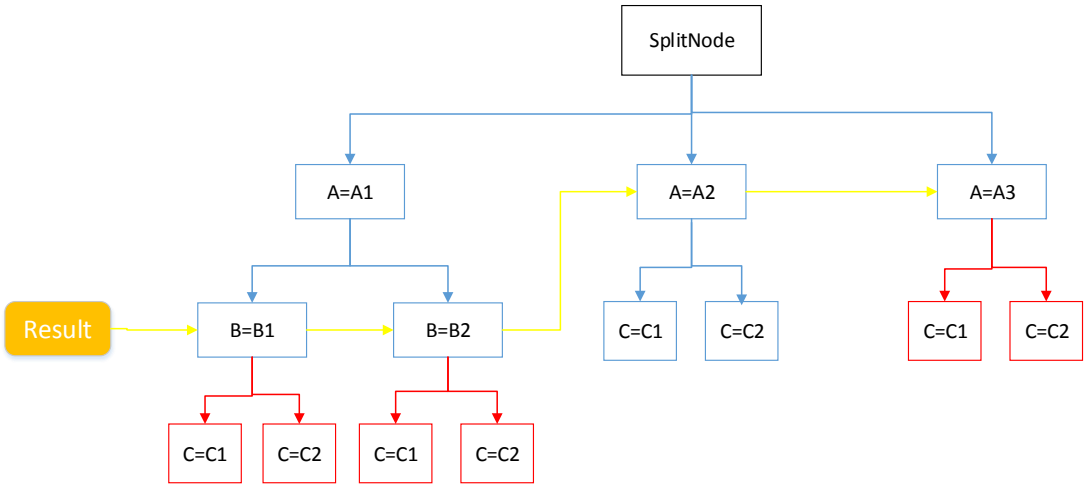

图 4.2 非叶子节点按照离散型维度分裂

图 4.2 是一个非叶子节点分裂的例子。假设表中有三个离散型维度 A、B、C，维度 A 可取值 A1、A2、A3，维度 B 可取值 B1、B2，维度 C 可取值 C1、C2。蓝色节点是初始时 SplitNode 所有的子孙节点，假设现在，我们想让 SplitNode 按照维度 C 进行分裂。由于 SplitNode 最初并非是按照维度 C 分裂的，因此需要遍历 SplitNode 的各个子节点。首先是(A=A1)节点，由于该节点是按照维度 B 分裂的，因此需要再遍历其子节点(B=B1)节点和(B=B2)节点。由于这两个节点都是叶子节点，我们将其按照维度 C 进行分裂。然后将(B=B1)节点和(B=B2)节点加入 Result 队列。同理，遍历到(A=A2)节点时，由于其本身就是按照维度 C 分裂的，因此需要将(A=A2)节点加入 Result 队列。(A=A3)节点也需要分裂并加入 Result 队列。

因此，分裂结束时，SplitNode 的子孙节点如图。红色的节点是新分裂出的节点。黄色的 Result 队列是分裂操作的返回值，该值将会提供给下一章中的查询引擎使用。

4.3.2 按连续型维度分裂

与按离散型维度分裂的情况相同，按连续型维度分裂也要分叶子节点和非叶子节点进行讨论。

在分裂之前，首先要得到分裂点，即在 where 条件中用户设置的范围。例如，假设用户在查询中设置了条件：Discount>=0.03 and Discount <=0.05，那么我们得到两个分裂点分别是 0.03 和 0.05。如果只有一个条件 Discount>=0.03，那么分裂点是 0.03 和 Double.MAX_VALUE；反之，Discount<=0.03 的分裂点是 Double.MIN_VALUE 和 0.03。最大分裂点 Double.MAX_VALUE 和最小分裂点 Double.MIN_VALUE 要特殊考虑。

当叶子节点分裂时，和按照离散型维度的分裂类似，只需要将分裂维度按照两个分裂点进行划分，划分出三个（当有最大分裂点或最小分裂点时是两个）子节点，然后将父节点中存储的样本存储到对应的子节点中。图 4.3 是一个叶子节点按照连续型维度分裂的例子。

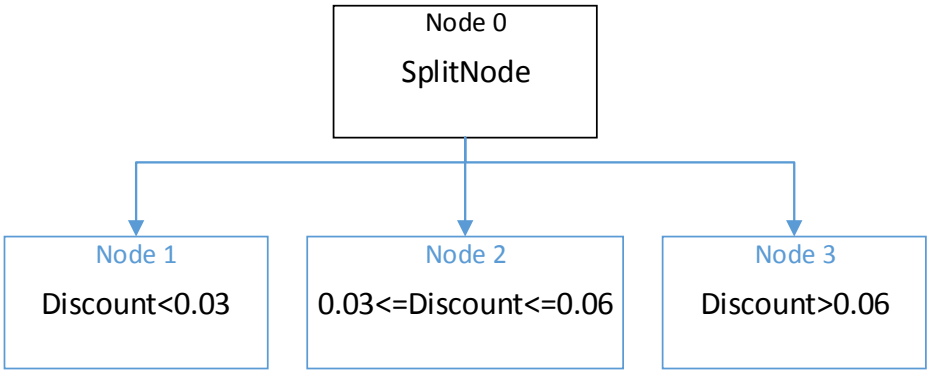

图 4.3 叶子节点按照连续型维度分裂

需要注意的是，如果一个节点已经按照某离散型维度分裂过，当节点再按照该离散型维度分裂时，并不需要做什么操作。但是对于连续型维度，新的分裂点将会导致节点分裂出更多的子节点。

分裂时，首先查询该节点的各个子节点对应的维度取值范围，找到新的分裂点在那个范围内，就将哪个子节点进行分裂。比如，以上面的 SplitNode 为例，假设我们又设置了查询条件 Discount>=0.08，那么需要将 SplitNode 按照新的条件再分裂一次。新的分裂点 0.08 落在了 Node3 对应的范围内（0.06 到 Double.MAX_VALUE），因此，需要将 Node3 分裂，分裂出 Node4 节点，这两个节点是兄弟节点。分裂结束后，SplitNode 及其子节点如图 4.4。

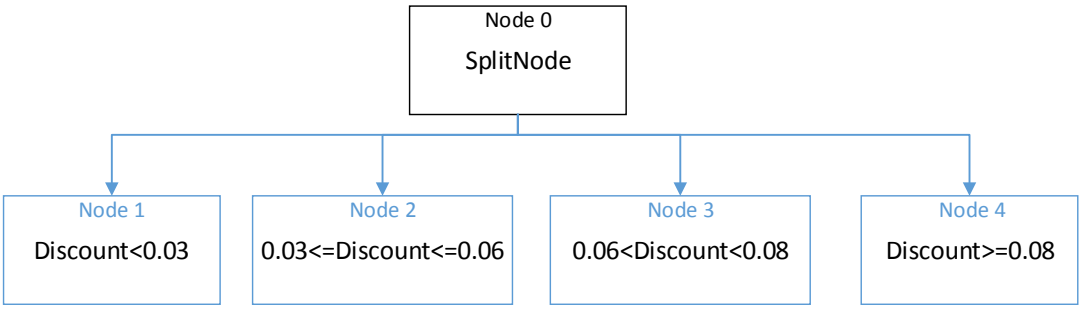

图 4.4 SplitNode 节点再次分裂

非叶子节点按照连续型维度分裂的算法和按照离散型维度分裂的算法过程大致相同，都是将节点的每个分支都按照该维度进行分裂，这里就不再赘述具体的算法。下面是一个例子。假设有连续型维度 A 和 B。SplitNode 节点的初始形态如图 4.5 中蓝色节点所示。现在将 SplitNode 按照条件 B>0.6 进行分裂。

首先，由于 SplitNode 不是按照维度 B 进行分裂的，因此，需要遍历 SplitNode 的所有子节点。子节点（A<0.4）按照维度 B 分裂出了两个节点(B<0.3)和(B>=0.3)，由于 0.6 是一个新的分裂点，因此，我们将节点(B>=0.3)进行分裂，分裂出兄弟节点(B>0.6)。(0.4<=A<=0.8)节点和(A>0.8)节点的处理过程类似。

最终，我们将新分裂出三个节点（红色的节点）。查询的最后，要将所有满足查询条件（B>0.6）的节点加入队列 Result 中（图 4.5 中黄色线段串联起来的三个节点），Result 队列将会被第五章中的查询引擎所使用。

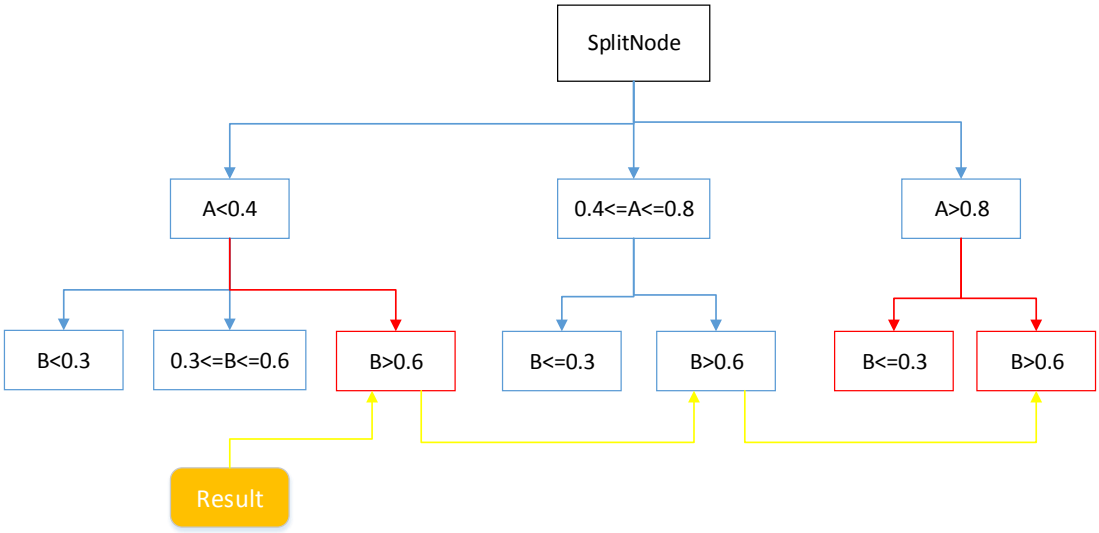

图 4.5 非叶子节点按照连续型维度分裂

4.4 节点合并

4.1 节介绍了节点的分裂方式。但是，在某些情况下，按照用户查询负载来划分连续型维度，会产生过多的子节点。为了减小维护子节点的开销，我们需要对某些节点进行合并。

需要注意的是，此处合并的都是按照连续型维度分裂出的节点。如果按照离散型维度分裂，则分裂出的节点数量一定小于 N_g， N_g 的值一般会设置为一个较小的值，因此没有合并的必要。但是如果用户在多个查询中为连续型维度设置了很多查询条件，那么分裂出的节点数量会很大，导致子节点数量几何级增长。

在合并的过程中，我们倾向于把重复利用率低的那些节点合并。因为这些节点用到的次数比较少，不值得花费时间和空间去维护。为了得到节点的重复利用次数，我们在节点对应的类中增加了一个计数器 Reuse。该计数器在节点被创建时被初始化为 1，节点每次被重复利用，该计数器的值会增加 1。

查找候选合并节点的算法如下：

Algorithm: Get Candidate Merge Node
Input: TreeNode root, int leafNum Output: TreeNodes candidate

```
1. N = GetLeafNodeCount(root)
2. if N < leafNum
3.     return
4. Add(queue, root)
5. while(!isEmpty(queue))
6.     current = RemoveFirst(queue)
7.     if current is split by continuous dimension
8.         for i in (0,GetChildCount(current)-2)
9.             x = GetChild(current,i).reuse + getChild(current,i+1).
reuse
10.                if x < min
11.                    min = x
12.                    candidate = getChild(i), getChild(i+1)
13.     if current is not leafnode
14.         for i in (0,GetChildCount(current)-1)
15.             queue.add(GetChild(current,i))
16 return candidate
```

该算法的输入是树的根节点和一个预设的参数 leafNum，输出是候选的合并节点。第 1 行中，计算出当前树的叶子节点的个数。2-3 行中，如果叶子节点个数少于参数 leafNum，则直接返回，若大于等于 leafNum，则进一步查找恰当的合并节点（4-16 行）。在查找的过程中，需要对树进行遍历，我们用了一个队列来辅助遍历过程（4-5 行）。

对于树中的每个节点，需要判断其是否是按照连续型的维度进行划分的（7 行），如果是，那么就访问该节点的子节点，计算出所有相邻子节点的 Reuse 之和（8-9 行）。如果该和小于现在已经计算出的最小的重用值之和（min），那么更新最小的重用值之和（min）以及候选的合并节点（candidate）（10-12 行）。如果该

节点不是叶子节点，那么就将其所有子节点加入到 queue 中（13-15 行）。遍历结束后，我们返回了候选的合并节点（16 行）。

通过该算法，就可以得到将要合并的节点。接下来，需要将这两个节点进行合并。

合并 node1，node2 两个节点时，首先需要合并其对应的统计估计量，统计估计量的合并方法将会在第五章介绍。接下来，我们将会把 node2 合并入 node1 节点。首先，将 node1 和 node2 的维度取值范围进行合并。然后，将以 node1 为根的子树中，所有的叶子节点存储的样本存入 node1 中。对于 node2 节点也如法炮制。然后，需要更新 node1 的父节点，告诉父节点 node2 子节点已经不存在。最后，删除 node1 的所有子节点和以 node2 为根的子树。

在合并中，我们删除了 node1 的所有子节点以及整个以 node2 为根节点的子树。因为父节点的复用次数必然大于所有子节点的复用次数，node1 和 node2 本身的复用次数就很低，因此，他们的子节点也没有继续维护的必要。

4.5 层次转换

层次转换是一项针对非叶子节点按照离散型维度分裂的辅助操作。当一个非叶子节点按照离散型维度进行分裂后，其后一步的操作很有可能是下钻操作。为了对下一个查询做准备，我们要改变树的层次。

例如，在图 4.2 中，将 SplitNode 按照维度 C 进行了分裂，由于 SplitNode 初始是按照维度 A 进行分裂的，因此需要遍历 SplitNode 的每个分支，确保所有分支按照维度 C 分裂。而用户下一步很有可能是基于维度 C 做下钻操作，但是此时按维度 C 分裂出的节点遍布各个子树，做下钻操作时就需要遍历这些节点，很不方便。为此，我们转换树的分裂层次，让 SplitNode 按照维度 C 进行分裂。

在将 SplitNode 按照维度 C 分裂后，得到了一个队列 Result，该队列保存了 SplitNode 所有分支中按照维度 C 分裂的节点。我们首先要做的是合并统计估计量。

下面是该过程的算法：

Algorithm: Tranfer - Merge
Input: Queue result, Demension d Output: EstimatorList estimatorList
1. for $\forall v \in Value(d)$ do 2. $estimator_v = MakeEstimator()$ 3. for $\forall node \in result$ do 4. for $\forall v \in Value(d)$ do 5. $node_v$ = GetChild(node,v) 6. $estimator_v$ = MergeEstimator($estimator_v$,GetEstimator($node_v$)) 7. for $\forall v \in Value(d)$ do 8. Add(estimatorList, $estimator_v$) 9. return estimatorList

首先，对于维度 d 可以取到的每个值 v，都创建一个对应的统计估计量 $estimator_v$（1-2 行）。对于 Result 队列中的每个节点 node，找到对应于取值 v 的子节点，然后得到该节点的统计估计量（GetEstimator 函数），将该估计量与 $estimator_v$ 进行合并（MergeEstimator 函数）。循环完成后，$estimator_v$ 就整合了 result 队列中所有取值为 v 的子节点的统计估计量（3-6 行）。然后，将这些统计估计量放入队列中，最后返回该队列（7-9 行）。

图 4.6 是 4.3.1 小节中，SplitNode 按照维度 C 分裂后，根据 Result 队列生成统计估计量的过程。因为维度 C 只有两个取值 C1 和 C2，对应的，我们建立了两个统计估计量，Estimator(C=C1)与 Estimator(C=C2)。Estimator(C=C1)通过合并四个红色节点的统计估计量得到，同理 Estimator(C=C2)通过合并四个绿色节点的统计估计量得到。

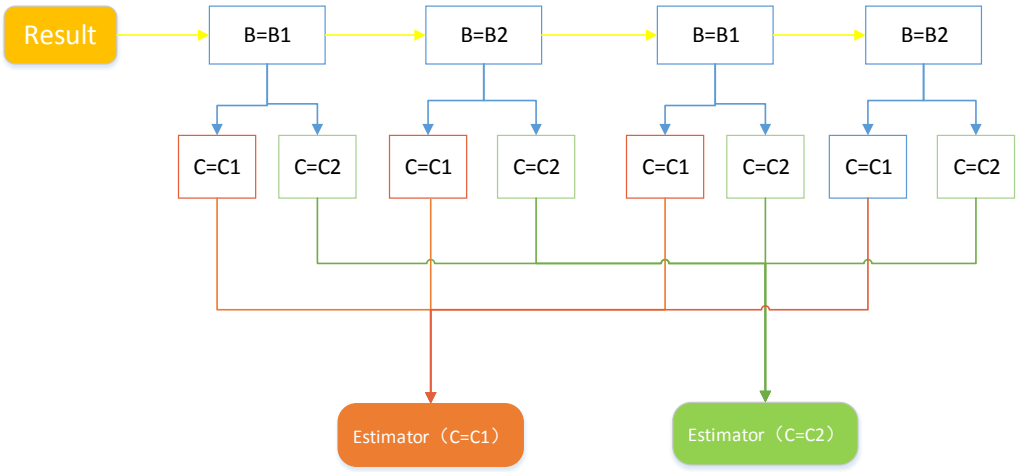

图 4.6 合并统计估计量

当此步骤完成时，就获得了样本中所有满足条件的样本的一组统计估计量。这些统计估计量中，有些置信度和置信区间已经满足了用户的设置的条件，对于这样的统计估计量，可以将结果立即返回给用户。对于还未满足置信度和置信区间的那些统计估计量，我们需要从随机数据集中读取更多的样本，来更新这些结果。在应用中，大部分统计估计量在此时都可以满足用户设置的条件，可以直接返回给用户。

接下来，要调整树的结构。这一步是在返回给用户结果之后进行的，因此，对查询的处理时间没有影响。

下面是该步骤的具体算法：

Algorithm: Tranfer - Change Structure
Input: EstimatorList, Node root, Demension d
1. CopyNode(root, newRoot);SetChild(Parent(root), newRoot) 2. for $\forall v \in Value(d)$ do 3. $estimator_v$ = GetEstimator(estimatorList, v) 4. $node_v$ = MakeNode($estimator_v$) 5. SetChild(newroot, $node_v$) 6. Add(queue,root);Add(newQueue, newRoot)

```
7.      while queue is not empty
8.          node = RemoveFirst(queue);newNode = RemoveFirst(newQueue)
9.          if node is not split by dimension d
10.             CopyChildNodes(node, newNode)
11.             AddFirst(queue, GetChildNodes(node))
12.             AddFirst(newQueue, GetChildNodes(newNode))
13.         else
14.             node = GetChild(node,v)
15.             if node is leaf node
16.                 CopyChild(node,newNode)
17.             else
18.                 UadateNode(node,newNode)
```

该算法的目的是更改树的层级结构，使以 root 为根节点的树，首先按照维度 d 进行分裂。为了辅助更改结构的操作，首先生成一个 root 节点的副本 newroot（1 行），将该节点作为新的 root 节点，然后将 root 节点摘出来，辅助生成树结构。首先，对于维度 d 的每个取值，取出第二步生成的统计估计量，并用该统计估计量生成一个节点 $node_v$（3-4 行）。然后，将节点 $node_v$ 设置为 newQueue 节点的子节点（5 行）。使用 queue 队列来对以 root 为根的树进行遍历，同时，使用 newQueue 队列来辅助生成新的以 $node_v$ 为根节点的树（6 行）。在每次迭代中，我们取得 queue 队列和 newQueue 的队首节点 node 和 newNode（8 行）。接下来判断 node 是否是按照维度 d 进行分裂。如果不是，那么将 node 所有的子节点都进行深复制，并设置为 newNode 的子节点（9-10 行），同时，将 node 节点的所有子节点加入 queue，将 nodeNew 的所有子节点加入 newQueue，后续对这些节点要进一步进行遍历（11-12 行）；如果是，那么将 node 节点更新为 v 值对应的子节点（14 行），然后判断该子节点是否是叶子节点，如果不是，那么将 node 节点的所有子节点拷贝成 newNode 节点的子节点，如果是子节点，那么用 node 节点来更

新 newNode 节点，更新的信息包括统计量，缓存类等。

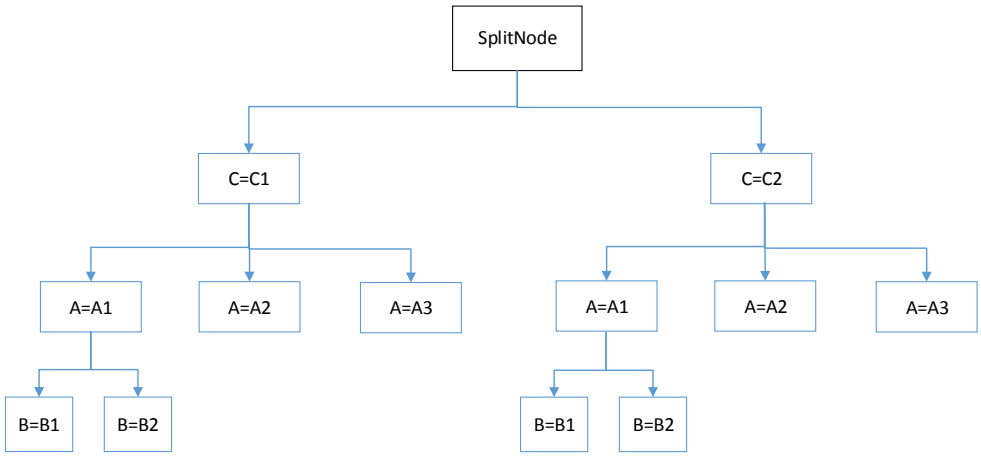

图 4.7 SpiltNode 节点层次转换后的形态

图 4.7 中，展示了 SplitNode 层次转换后的形态。这样 SplitNode 首先就以维度 C 进行了分裂。需要注意的是，上述操作是在得到查询结果后完成的，因此，并不会延长查询的处理时间。

4.6 样本存储

在样本管理树中，我们将样本存在树的叶子节点中。为了构建轻量级的叶子节点，我们为存储样本的缓存专门构建了一个类，这个类的引用被存放在叶子节点中。当叶子节点需要访问自己存储的样本时，节点会通过这个引用找到存储样本的缓存。这样做的好处是，当需要将一个叶子节点 node1 中的样本全部存入另一个叶子节点 node2 时，只需要将 node2 中的指向样本存储的引用设置成 node1 中的引用，然后清空 node1 中对应的引用即可，而不必进行真正的数据存储，节省了 I/O 时间。

令一个需要注意的问题是，某些查询的条件会设置的较为苛刻，这里的苛刻有两种情况：一是查询的 where 语句比较多，或者 where 语句的选择范围很窄，这样就会导致查询的选择度很低，样本集中满足查询条件的样本数量少；二是用户的置信度设置的太高（比如大于 99.9%），或者误差界限设置的太小（比如小于 0.1%），这时，需要数量巨大的样本来满足用户的精度要求。这两种情况都会导致

管本管理树中样本数量的激增。

然而，系统中内存空间是有限的。当系统中的内存不足以存储全部样本时，需要考虑将内存中的信息交换到磁盘中。因此，我们在节点中增加一个域，用来管理存储在磁盘中的样本。同时，还要增加一个布尔型的标志 isInMemory,该标志指示当前节点的样本是存储在内存中（True）还是磁盘中（False）。

在系统中，我们采用了 LRU 置换算法，将最久没访问过的叶子节点中的样本存入磁盘。通过一个链表串联起来的哈希表来实现 LRUCache，保存叶子节点的访问顺序。当节点创建,或者被访问时,将该节点置于 LRUCache 的表头。当 LRUCache 的结构发生改变时，会自动检查当前的内存使用状态。当剩余可用内存与最大可用内存的比值小于某参数 r 时，将会触发置换功能，将 LRUCache 中最后面的节点对应的样本存储到外存中，同时，将该节点的 isInMemory 置为 false。

4.7 本章小结

本章介绍了一种树结构，该树结构用来存储在线聚集过程检索出的样本（叶子节点中保存的样本）以及以前查询所产生的中间结果（节点中保存的统计估计量）。之后，我们介绍了这个树结构中，节点应该按照什么维度进行分裂，具体的节点分裂算法，节点合并算法，层次转算算法，以及样本在叶子节点中的存储。

第5章 查询引擎和统计估计量

第四章介绍了样本管理树，并且介绍了几种该树的基本操作。当一个查询进入系统时，首先会在样本管理树中查找，查到满足查询条件的样本和中间结果，然后将这些样本产生的统计估计结果与中间结果进行合并，生成一组查询结果。生成的查询结果达不到用户设置的精度时，再从随机数据集中读取更多的样本。下面，结合 3.2 小节提出的五种基本操作，和上一章中的树结构，介绍如何由用户的查询生成最终的查询结果。本章包括两部分，5.1 介绍了查询引擎的工作方式，5.2 介绍了统计估计量的计算与合并。

5.1 查询引擎

查询引擎的功能是根据用户的查询请求，计算出相应的查询结果，然后返回给用户。在本小节中，主要讨论查询引擎如何处理查询请求。对于查询请求的处理主要分两个部分，首先是在样本管理树中查询，并生成结果；当上面的结果不能满足用户要求时，再到随机数据集中查询。

5.1.1 在样本管理树的查询过程

在 3.2 节中介绍了用户可以进行的五种查询，在本节中，将分别介绍这五种查询如何在样本管理树中进行处理，以及每种查询下样本管理树的结构如何变化。

需要注意的是，在样本管理树中进行查询操作时，有一个焦点节点的概念。焦点节点最初为根节点，并且会随着查询而发生变化。在一个查询中，如果一个节点是焦点节点，那么我们只需要在以焦点节点为根节点的子树中处理该查询即可。

（1） 更改选择维度操作。

在系统中，当用户更改选择维度时，样本管理树的结构不会发生改变，焦点节点也不会改变。样本管理树的结构是否发生变化取决于查询中 where 和 group by 语句。在更改选择维度操作中，where 后面的限制条件和 group by 后面的聚

合维度都没有发生变化。因此，在该操作中，样本管理树的结构不会发生改变。同理，焦点节点也不会发生改变。

我们在每个节点中都保存了所有连续型维度的统计估计量。当更改选择维度时，只需要根据上一次查询得到的 Result 节点队列，重新合并对应维度的统计估计量，就可以得到初始的聚合结果。

如果初始的聚合结果就可以满足用户设置的置信度和置信区间，那么就将此结果返回。否则，需要在数据源中继续扫描数据。这个步骤对于五种操作都是适用的，在后面的操作中就不再赘述。

（2）下钻操作

对于下钻操作，需要将上次的聚合维度固定在某个值上，然后对另一个离散型维度做聚合查询。因此，对于下钻操作，需要首先要转换焦点节点，新的焦点节点是原来的焦点节点的子节点。然后，将新的焦点节点按照另一个维度分裂，得到 Result 节点队列。之后，可以按照 4.5 节中的合并统计估计量算法得到初始的聚合结果。最后，需要对焦点节点做一次层次转换操作，焦点节点首先按照最新的聚合维度分裂。

（3）更改条件操作

更改条件操作指的是更改上一次设置的条件（连续型或者离散型）。当更改限定条件时，首先需要改变焦点节点。新的焦点节点是新的条件对应的节点（如果是更改的是连续型条件，那么可能需要对焦点节点进行分裂）。然后，对新的焦点节点按照聚合维度进行分裂。分裂后，如果需要的话，还需要进行一次层次转换操作。

（4）更换聚集维度操作

该操作指的是在前一个查询的基础上，更改其 group by 语句后面的离散型维度。对于该操作，焦点节点也是不需要改变的。需要将焦点节点按照新的维度进行分裂，通过分裂后得到的 Result 队列，可以得到聚合结果。

（5）上卷操作

该操作指的是在在前一个查询的基础上，删除最后一个添加的 where 条件。

这时，需要将焦点节点变为原来焦点节点的父节点。然后在新的焦点节点中查询对应分裂条件的节点，将满足条件的节点加入 Result 队列，计算出聚合结果。最后，我们需要在新的焦点节点上做一次层次转换操作。

5.1.2 在数据源中查询

5.1.1 小节介绍了五种基本操作如何在样本管理树中查询到可以复用的统计估计量或者样本。在大部分情况下，样本管理树中的样本是可以满足用户的查询精度要求的。但是，在两种情况下，样本管理树中的样本可能不足。一是用户进行了过多次下钻操作，导致查询中的 where 语句很多，查询的选择度就会很低。这时，扫描很多样本才能找到一个满足条件的样本。二是用户设置的置信度太高（比如 99.9%）或者置信区间范围太小（比如 ErrorBound 为 0.001），这时，需要很多的满足条件的样本才能满足用户的查询精度。因此，在这两种情况下，系统需要大量的样本，内存管理树中现有的样本很可能不能满足要求，此时，系统需要在数据源中直接读取数据。

上文所定义的五种操作在数据源中的查询过程是相同的，不同的只是这些操作所对应的查询语句。因此，系统首先要解析出查询对应的限制条件，即 where 语句中的各个条件，这些条件将帮助系统对样本进行筛选。当在随机数据集中检索样本时（会从上一次结束的位置开始检索），系统会判断扫描到的元组是否满足刚才解析出的查询条件。当满足时，该元组就可以用来更新在内存管理树中生成的初始的统计估计量，直到满足用户的精度要求。

需要注意的是，系统在更新统计估计量时采用了批量更新的方式。如果系统每得到一个符合用户查询要求的元组，就用该元组去更新统计估计量，将会导致大量无效的计算。系统中设置了一个参数 S_{batch}，并将满足条件的元组缓存起来，只有当这些元组的数量大于 S_{batch} 时，系统才会用这些元组对统计估计量做更新。S_{batch} 的值也不能设置的过大，过大将会导致统计估计量更新的太慢。

这样，通过不断的检索更多样本，系统将会逐渐缩小置信区间，直到置信区间满足用户的要求，这时，就可以将结果返回。

还有一点是，这些所有新采集出的样本，系统要将它们插入到样本管理树中。由于该操作是和计算查询结果无关的，因此系统可以在返回用户查询结果后进行此操作，这样，可以保证用户得到一个更快的响应时间。

在扫描时，系统将这些元组加入到样本管理树 N=Root 节点的缓存中。在将查询结果返回给用户后，系统将这些元组插入样本管理树。插入的过程就是沿着树中节点逐渐下沉，根据节点的分裂维度，找到对应的子节点，此处就不再赘述。最终，每个元组都会插入到对应的叶子节点。在将元组插入叶子节点时，需要更新该叶子节点对应的统计估计量。和上面一样，此处系统也将元组缓存在起来，只有当缓存中元组的数量大于 S_{batch} 时，才批量地将缓存中的元组插入叶子节点的存储空间，然后再更新统计估计量。

由于父节点中的统计估计量是所有子节点的统计估计量的整合，当子节点更新统计估计量时，父节点对应的统计估计量也相应的改变了。因此，当子节点批量插入样本，更改统计估计量后，需要沿着到 Root 的路径，去更新自己所有祖先节点的统计估计量。

5.2 统计估计量

在样本管理树中，每个节点保存着该节点的所有后继节点保存的样本的统计估计量。本节将介绍这些统计估计量是如何计算出来的。5.2.1 小节介绍叶子节点的统计估计量的计算。5.5.2 小节介绍如何将多个统计估计量进行整合，也就是非叶子节点的统计估计量的计算。

5.2.1 叶子节点的统计估计量

统计估计量包含两个部分：聚合结果和置信度与置信区间。在叶子节点中，估计量和置信区间的计算是根据节点中插入的样本计算出来的。本小节中，将介绍聚合函数 AVG 的估计量和置信区间的计算。

对于每个叶子节点，需要记录当前已经插入该节点的样本数量，系统用 N 来表示该数量。对于表 $R=\{c_1,c_2,\ldots,c_k\}$，如果 c_i 可以出现在一个查询的聚合函数中，那么我们称之为聚合维度。例如，在 TPC-H 中的 LINEITEM 表中，ReturnFalg 不

是一个聚合维度，而 Discount 就是一个聚合维度。我们用集合 C 来代表一个表中的聚合维度的集合。

对于一个数据表 R={c_1,c_2,…,c_k}，如果 $c_i \in C$，那么就在节点中记录所有 c_i 可能的聚合结果。比如，假设节点 n_i 中已经插入了 N 个样本。当前 c_i 列的聚合值是 avg(c_i)=V_{avg}。当检索出一个数据表 R 中的元组 t={v_0,v_1, …,v_k}，如果 t 被插入了节点 n_i，那么就可以更新节点的聚合结果 avg(c_i):

$$avg(c_i) = \frac{V_{avg} N + v_i}{N+1} \qquad \text{公式(5.1)}$$

通过这种方式，可以得到集合 C 中所有维度的聚集结果。这些结果将会在不同的查询中被重复使用。

除了聚合结果外，系统还要计算出对应该聚合结果的置信度和置信区间。在 2.2.2 节中，我们介绍了三种置信区间类型，由于基于中心极限定理的大样本置信区间在样本量较大时估计效果最好，因此，系统采用了该种置信区间。

在 2.2.2 节中，有：

$$P\{|X_i - u| \le \varepsilon_n\} \approx 2\phi(\frac{\varepsilon_n \sqrt{N}}{\delta^2}) - 1 \qquad \text{公式(5.2)}$$

上式中，X_i 是插入节点 n_i 中的元组 t 在维度 c_i 上的取值，u 是所有 X_i 的平局值，δ^2 是所有 X_i 的方差。等式左侧是用户设置的置信度 p，ε_n 是误差界限。因此，为了计算出置信度和置信区间，除了样本数量 N，还需要有 X_i 的方差 δ^2。

假设现在已经有样本 $X_1, X_2, \ldots, X_n$，那么可以得到对总体方差的估计：

$$\begin{aligned} \overline{\delta_n}^2 &= \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \overline{X})^2 \\ &= \frac{1}{n-1}\sum_{i=1}^{n}({X_i}^2 - 2X_i\overline{X} + \overline{X}^2) \\ &= \frac{1}{n-1}(\sum_{i=1}^{n}{X_i}^2 - 2\overline{X}\sum_{i=1}^{n}X_i + n\overline{X}^2) \end{aligned} \qquad \text{公式(5.3)}$$

在上式中，$\overline{X}$ 为所有样本的均值，该值即为上文计算出的 $avg(c_i)$。因此，为

了方便的更新δ^2，我们还要在节点中存储所有样本的平方和$\sum_{i=1}^{n} X_i^2$，记为S_n^2，以及所有样本的和$\sum_{i=1}^{n} X_i$，记为S_n。当有新的样本X_{n+1}插入到节点中，可以更新δ^2为：

$$\overline{\delta_{n+1}}^2 = \frac{1}{(n+1)-1}[(S_n^2 + X_{n+1}^2) - 2avg(c_i)(S_n + X_{n+1}) + (n+1)avg(c_i)^2] \quad \text{公式(5.4)}$$

这样，就可以估计出总体的方差。然后就可以计算出对应某置信度的误差界限，从而计算出置信区间。

因此，在节点中，除了要记录聚集结果的估值，还要记录两个变量，一是样本的数量，另一个是样本对应域的平方和。后两个值可以帮助我们计算出样本方差，从而应用中心极限定理得到对应置信度下的误差界限。

5.2.2 统计估计量的合并

在两种情况下，系统需要对统计估计量进行整合。第一种情况是父节点要整合所有子节点的统计估计量。由于在样本管理树中，节点中的统计估计量是该节点的所有子孙节点中存储的样本所形成的统计估计量，对于非叶子节点，它对应的统计估计量就是它所有子节点的统计估计量的整合。因此，当子节点的统计估计量发生改变时，系统要更新其有所祖先节点的统计估计量。第二种是当查询覆盖了多个节点时，需要重用这些节点所对应的统计估计量，相应的，系统也要将这些统计估计量整合起来，以形成最终的查询结果。

每个节点中的样本（非叶子节点是其子孙节点中的子节点中存储的样本）都是一个独立的样本集。系统采取了带有权重的方式来整合这些统计估计量。

首先是聚合结果，假设节点 n_i 中的聚合结果是 X_i，那么，整合后的聚合结果如下计算：

$$X = \sum_{i=0}^{k} w_i X_i + \sum_{i=0}^{k}\sum_{j=0}^{k} \mathrm{cov}(X_i, X_j) \quad \text{公式(5.5)}$$

w_i 是赋予节点 n_i 的权重，并且 $\sum w_i = 1$。$\mathrm{cov}(X_i, X_j)$ 是两个聚合结果之间的协方差。因为在系统的样本管理树中，任意两个节点间都没有相同的样本，因此，不同节点的聚合结果之间的协方差是很小的。为了简化上述公式，可以直接将协方差部分去掉。假设节点 n_i 的方差是 δ_i^2，可以用如下的公式来计算最终的方差[5]：

$$\overline{\delta^2} = \sum w_i^2 \delta_i^2 \qquad \text{公式(5.6)}$$

能够最小化方差的最优化权重可以通过下面的公式得到[5]：

$$w_i = \frac{1}{\delta_i^2 \sum \frac{1}{\delta_i^2}} \qquad \text{公式(5.7)}$$

我们注意到，当整合多个节点的统计估计量的时候，有时，会由于有点节缺乏足够多的样本而低估方差。$\sum w_i = 1$，而 $\sum w_i^2 \le 1$。一些样本比较少的节点可能会产生一个较小的 δ^2，而按照上面的公式，这种节点将会被赋予较大的权值，这样，就会使最终的方差估计量产生偏差。这一点在当查询涉及到很多节点时会表现的很明显。因此，我们需要对这点做出修正。

在上一小节中计算出的方差是一个估计值。我们用了样本的方差来替代整个数据集的方差。通过分析，可以得到一个更好的对方差 δ^2 的估计[33]：

$$\delta^2 \sim \frac{\delta_s^2 \chi^2}{K-1} \qquad \text{公式(5.8)}$$

上式中，δ_s^2 是由样本计算出来的方差，χ^2 服从卡方分布，K 是节点的数量。通过展开 χ^2，可以得到 δ^2 的最终估计值：

$$\delta^2 = \overline{\delta}^2 \times \frac{1}{K-1} \sum_{i=1}^{K} \frac{(X_i - X)^2}{\delta_i^2} \qquad \text{公式(5.9)}$$

用方差来计算聚合结果的置信度和置信区间。假设最终的准确结果是 $\overline{X}$，基于中心极限定理，如果 X 是由大量的样本计算出的，那么 $\overline{X} - X$ 服从正态分布。因此，我们有：

$$P(\overline{X} - X < \varepsilon) \cong 2\phi(\frac{\varepsilon\sqrt{\overline{N}}}{\overline{X}}) - 1 \qquad \text{公式(5.10)}$$

其中，$\overline{N}$ 是查询所检索出的所有样本，$P(\overline{X} - X < \varepsilon)$ 是查询的置信度（Confidence），ε 是误差。

5.3 本章小结

本章介绍了系统中查询引擎的实现。查询引擎首先会在第四章的内存管理树中进行查找，本章介绍了五种用户操作在内存管理树中查找并生成结果的方式；内存不足时，查询引擎会直接从数据源中进行数据访问，这个过程对五种查询方式都是相同的。然后，本章介绍了如何用样本生成统计估计结果，包括估计值、确定置信度下的置信区间。最后，本章介绍了如何将多个统计估计结果进行合并。

第6章 实验结果及分析

本系统的两个最重要的指标分别是系统处理查询的时间和结果的有效性。本章将通过实验，重点验证这两项指标的有效性。

6.1 实验配置

6.1.1 运行环境

本系统是用 Java 语言完成的。运行环境如下：

表 6.1 机器软硬件配置

项目	内容
中央处理器	Intel Xeon E5-2660
内存	16GB
JDK	JDK1.8
操作系统	Linux Ubuntu 14.04 LTS server

6.1.2 实验数据

TPC Benchmark™H（TPC-H）是一个决策支持基准。它由一组面向业务的即席查询和并发数据修改组成。选择查询和填充数据库的数据具有广泛的行业相关性。这个基准说明了决策支持系统，它们检查大量数据，执行高度复杂的查询，以及解决关键业务问题。

本文使用了 TPC-H 的 10G 数据来测试系统的性能。TPC-H 数据集中一共有 8 张表，本文选取了其中维度比较多的 LINEITEM 和 ORDERS 表对系统性能进行测试。

本文在系统中的查询操作对应的 SQL 语句是由下面两个模板生成的：

查询模板一（单表）：

SELECT avg(c_i)

FROM LINEITEM

WHERE [discount > x and discount < x+0.03] | [extendedprice > x and extendedprice < x+30000] | [quantity > x and quantity < x+30]

GROUP BY [returnflag] | [linestatus] | [shipmode]；

其中 c_i 可取的维度为 quantity、discount、extendedprice，x 为对应维度上的某些随机值。

查询模板二（多表）：

SELECT avg(c_i)

FROM LINEITEM L， ORDERS O

WHERE L.orderkey = O.orderkey and

[discount > x and discount< x+0.03] | [extendedprice> x and extendedprice < x+30000] | [totalprice > x and totalprice < x+300000]

GROUP BY [returnflag] | [linestatus] | [shipmode] | [orderpriority]；

其中 c_i 可取的维度为 quantity、discount、extendedprice、totalprice。x 为对应维度上的某些随机值。

查询模板一的数据集是 LINEITEM 表，查询模板二的数据集是 LINEITEM 表和 ORDERS 表。本文对两个模板分别进行了实验。参照上面的模板，本文通过两个模板各模拟出了 500 次查询操作，每个查询操作都是 3.2 小节中介绍的五种基本查询操作类型之一。出于性能和实用性，本文对这些查询的选择度进行了限制，实验中这些查询的选择度最低为 0.005。

6.1.3 实验设置

本文通过 500 个查询操作的平均处理时间来衡量系统的查询处理效率。对每个查询，要预设置两个参数，分别是置信度和误差界限。本文分别用 c 和 ε 来表示置信度和误差界限。假设系统返回的聚集结果为$\overline{V}$，准确的查询结果为 V，那么，$P(\frac{|\overline{V}-V|}{V}<\varepsilon)=c$。本文在系统中默认设置的 c 和 ε 分别是 95%和 0.02。在每个查询中，对于 group by 语句后面的维度的每个取值，本文都会计算其对应的置信度和误差界限。当达到用户要求的准确度时，该取值对应的查询结束。当

所有取值对应的查询都结束，系统把查询结果返回给用户。为了防止树的节点过多，本文设置模板一中叶子节点数量的最大值 N_g 为 1200，模板二中叶子节点数量的最大值 N_g 为 1800，S_{batch} 的值设为 2000。

为了体现树样本管理树的效果，本文设置了一项对比实验。在对比实验中，让查询之间只共享样本，即对于查询在随机数据集中检索到的样本，系统将其存储在内存中（内存不足时存储在磁盘中）。每当处理新查询时，系统首先扫描已经在内存（或者磁盘）中的样本，当这些样本不足时，再去扫描随机数据集，本文记这种方式为 OLA-Buffered；记上文所提出的方式为 OLA-Tree。OLA-Tree 与 OLA-Buffered 的主要区别在于 OLA-Tree 采用了树来组织样本和中间结果，以供后面的查询使用。

6.1.4 用户界面

用户界面如图 6.1 所示。左边栏是数据导航栏，用于提示用户当前数据源中的离散型和连续型维度有哪些。在查询操作栏中，用户可以进行五种查询操作，每种查询用户都需要设置一些查询条件。右侧的当前查询栏把当前用户的查询按照 SQL 语句的形式表示了出来。右下是查询结果栏，该栏显示的是对应当前查询的查询结果。在右上角的 Settings 选项中，用户可以选择数据源，或者对各项参数进行设置。

图 6.1 系统界面

6.2 实验结果与分析

6.2.1 数据集大小对实验结果的影响

在第一个实验中，本文测试了不同数据量下系统响应查询的速度。本文分别在 2G、4G、6G、8G 和 10G 的数据集上进行实验，来评估系统的执行速度。同时，本文还在 Mysql 上进行了同样的查询，以得到精确的查询结果。图 6.2 和图 6.3 分别显示了查询模板 T1 和 T2 实验结果。

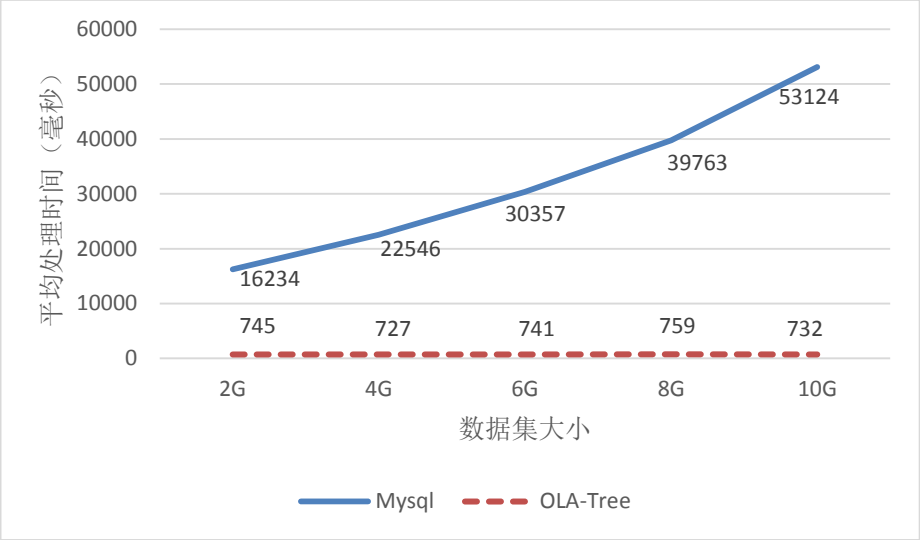

图 6.2 数据量大小对处理时间的影响（查询模板一）

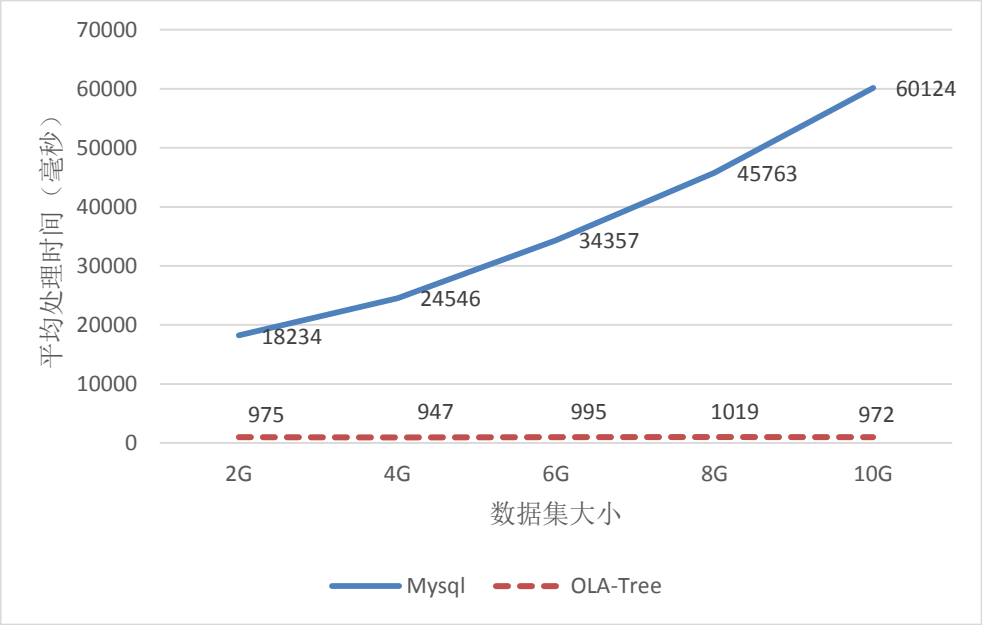

图 6.3 数据量大小对处理时间的影响（查询模板二）

从上述结果可以看出来，OLA-Tree 的处理时间要远小于数据库计算精确值的处理时间。这种时间上的差异主要是由于处理数据量大小的不同，OLA-Tree 使用了采样技术，因此处理的数据量要比计算精确结果处理的数据量小。在上面的实验中，将置信度设为 0.95，误差界限设为 0.02，OLA-Tree 产生的结果已经足够接近正确结果了。

另一个值得注意的点是 Mysql 处理查询的时间随着数据量的增加大致呈线性增长，并且随着数据量的增加其走势越来越陡峭；而 OLA-Tree 则对数据量的变化不敏感，处理时间基本没有变化。之所以出现这种结果，是因为在 OLA-Tree 中，当置信度和误差界限被固定，采样的数量就取决于数据的方差。虽然数据量增大，但是数据的分布没有变化，因此方差也没变。因此，尽管数据量增大，但 OLA-Tree 所采的样本的数量几乎是一样的，处理时间也就不会发生太大变化。

6.2.2 置信度和误差界限对查询时间的影响

在系统中，用户需要预先指定两个参数，分别是置信度 c 和误差界限 ε 。在本小节中，我们研究这两个参数对实验结果的影响。

首先是置信度。置信度是真实的结果落在系统计算出的结果区间的概率。当置信度增大，系统需要检索更多的样本，处理时间也会响应延长。我们将误差界限设为 0.02，测试在不同置信度下系统处理查询的平均时间。结果如图 6.4 和图 6.5 所示。

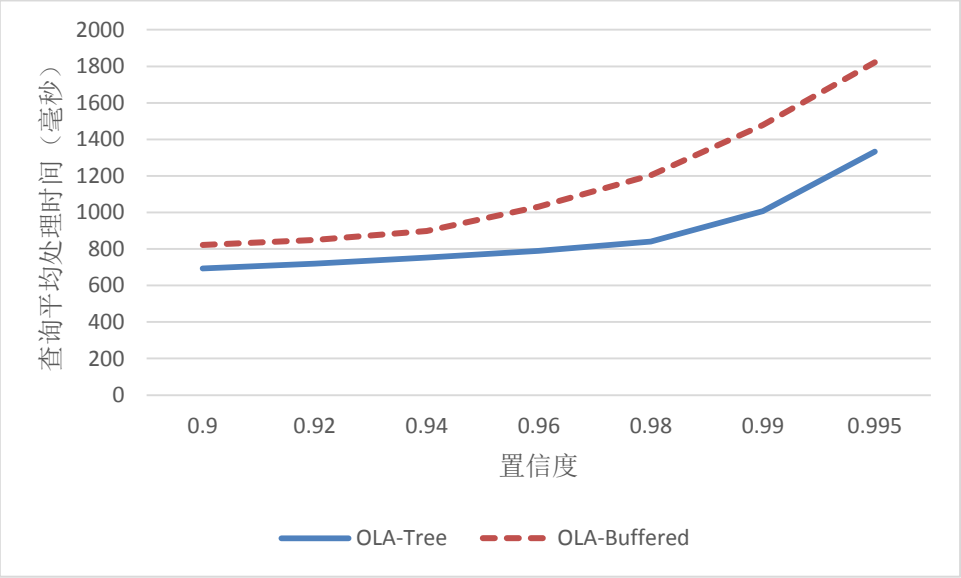

图 6.4 不同置信度下的处理时间（查询模板一）

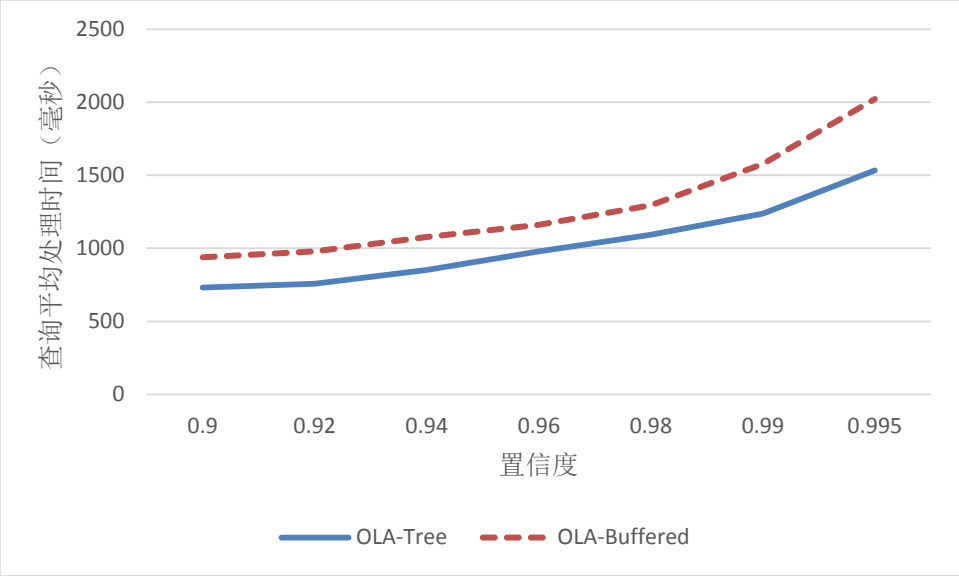

图 6.5 不同置信度下的处理时间（查询模板二）

通过图 6.4 和 6.5，可以发现，随着置信度的增大，无论是 OLA-Tree 还是 OLA-Buffered，其处理时间都明显地增大了。但是，OLA-Tree 的处理时间还是要优于 OLA-Tree。这是主要是因为 OLA-Tree 在计算结果时利用了树节点中记录的中间结果。

需要注意的是，对于时间曲线，开始时较为平缓，但是当置信度从 0.99 变为 0.995 时，曲线变得陡峭。这是因为，采样的数量和 $z(\frac{1+p}{2})$ 是成正比的，$z(\frac{1+p}{2})$ 指的是正态分布概率为 $\frac{1+p}{2}$ 时所对应的值。当 p 越接近 1，$z(\frac{1+p}{2})$ 的增长就越快。

然后是误差界限。系统通过误差界限来计算置信区间。假设将误差界限设置为 ε，系统返回的聚合结果为 X，那么系统得到的置信区间就是[(1- ε)X，(1+ ε)X]。因此，如果用户想得到一个小的置信区间，那么用户就应该将误差范围设置的小。图 6.6 和 6.7 是把置信度固定在 0.95 时，设置不同的误差界限对系统处理查询的时间的影响。

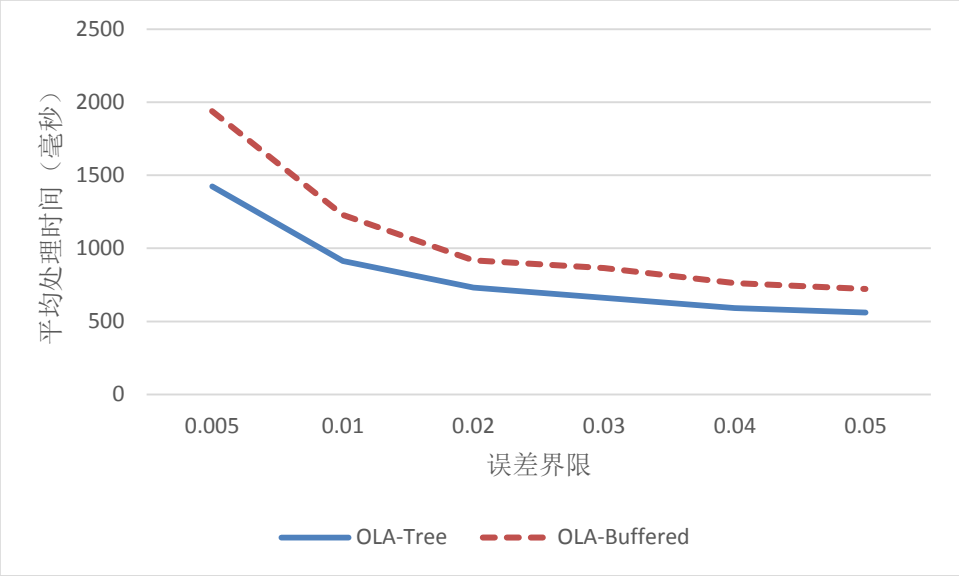

图 6.6 不同误差界限下的处理时间（查询模板一）

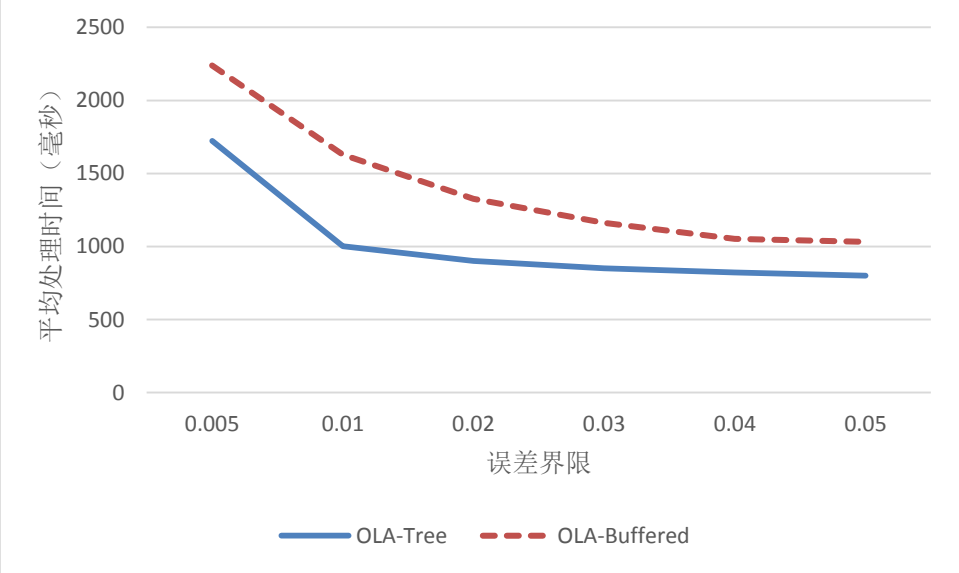

图 6.7 不同误差界限下的处理时间（查询模板二）

可以看到，实验结果和预期是相符的。当误差界限增大的时候，查询的平均处理时间也会缩短。并且，OLA-Tree 的处理时间优于 OLA-Buffered 的处理时间。

和置信区间类似，在图 6.6 和图 6.7 中，可以发现，当误差界限从 0.01 减小到 0.005 的时候，系统的查询处理时间急剧增加。这是因为查询需要的样本数量是和误差界限的倒数的平方呈正相关的。误差界限缩小为原来的一半，需要的样本数量就变为原来的四倍。另一方面，当把误差界限设为 0.005 时，某些选择度

低的查询需要检索大量的样本，内存不足以存储这些样本，从而需要存储在磁盘中，这也会拉慢系统的处理速度。当误差界限设置的更小时，系统的性能会更加恶化，甚至退化为全数据集扫描。

但是，一般情况下，0.01-0.05 的误差界限已经可以产生足够好的结果了。在这个误差界限范围内，系统的处理时间还是很有优势的。

6.2.3 聚合结果的准确性

在本实验中，需要验证系统计算出的置信度 c 和误差界限 ε 的准确性。

首先，我们考察对应不同置信度下的误差界限 ε 是否准确。在查询中，系统将误差界限 ε 的值固定为 0.02。在 6.2.1 小节的实验中，已经用 Mysql 计算出了每个查询的精确结果 v，假设系统产生的估计结果为$\bar{v}$，那么可以计算出真实的误差界限：$\frac{|\bar{v}-v|}{v}$。我们可以将真实的误差界限与 0.02 做比较。图 6.8 是用查询模板一与查询模板二共一千个查询生成的实验结果。

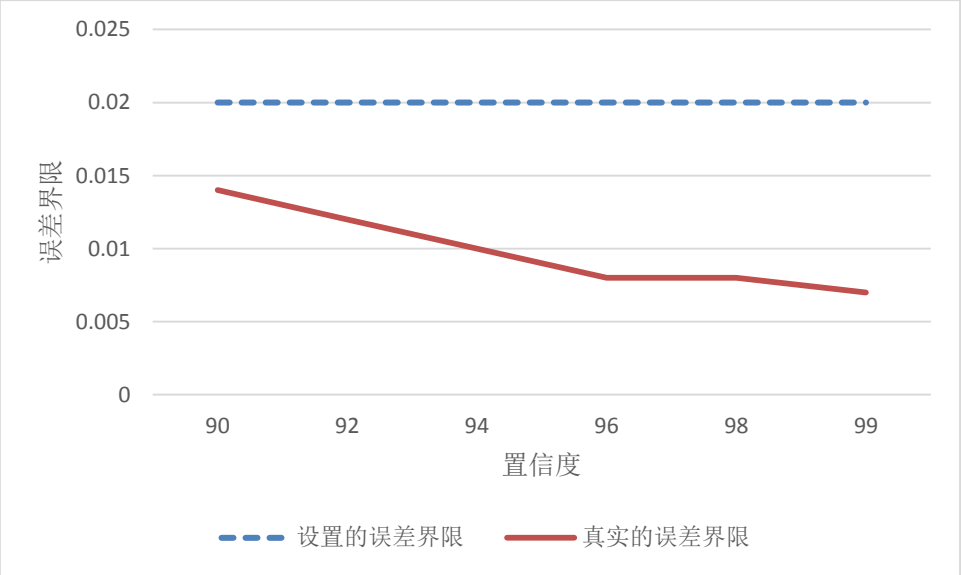

图 6.8 不同置信度下的误差界限

通过图 6.8 可以看出，真实的误差界限是一直小于设置的误差界限的。之所以出现这种结果，是因为在很多查询中，使用了系统中的全部样本，这是多于查询所需要的样本的数量的。

接下来，我们验证置信度是否准确。在查询中，系统将置信度固定为 90%，然后计算出在不同的误差界限下的真实置信度。假设我们将误差界限设置为 ε，实验 6.2.1 中用 Mysql 计算出的精确结果为 v，系统产生的估计结果为$\bar{v}$。那么对于模板一和模板二的共 1000 个查询，统计真实结果 v 落在区间 $[(1-\varepsilon)\bar{v}, (1+\varepsilon)\bar{v}]$的概率 p。p 即为真实的置信度。图 6.9 是实验结果（实验中，为了防止选择度很低的查询产生的大量样本对实验结果造成较大偏差，本实验修改了一些查询，将选择度较低的查询集中在比较靠后的位置）。

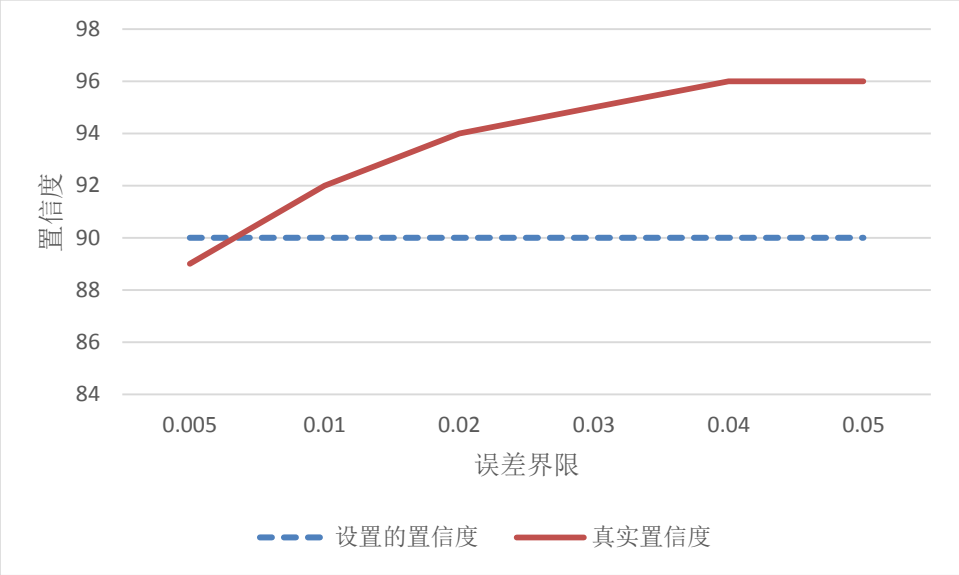

图 6.9 不同误差界限下的置信度

由实验结果，可以看出，当误差界限较小时，真实的置信度可能略低于用户设置的置信度。但是随着误差界限逐渐增大，我们的真实置信度也会随之增大，并且稳定地大于用户设置的置信度。

6.3 本章小结

本章通过实验验证了本文提出的技术的有效性。首先，本章验证了，在用户设置的置信度和置信区间不是太严苛的情况下，该技术相比于数据库对数据量大小的敏感度低。然后，本章与直接缓存样本，没有采用树结构的在线聚集方式做了对比，验证了系统在处理时间方面的优势。最后，通过实验，验证了系统产生

的置信度和误差界限的有效性。

第7章 总结与展望

7.1 本文主要工作与贡献

本文研究并实现了一种基于模糊查询的大数据分析处理技术，通过该系统，用户可以快速地勾勒出数据的轮廓。

本文的主要贡献如下：

1) 本文设计了一套数据访问接口，其中主要包括几种用户查询操作。通过这些查询操作，用户可以自由地对高维数据进行各种聚集查询。
2) 本文提出了一种树形结构优化查询过程。鉴于查询之间，尤其是相邻查询之间，往往是有交集的，因此，本文把以前查询检索到的样本和产生的中间结果存储在该树结构中。随着查询不断进入系统，该树结构的节点会逐渐分裂，层次也会随之改变。
3) 基于上述树形结构，本文实现了用户的接口的查询过程。该查询过程主要包含两个部分，一是在树形结构中的查询，二是在数据源中的查询。在树形结构中查询的主要目的是找到已经在系统中的符合当前查询条件的样本或者中间结果，然后复用这些样本或中间结果；当在树形结构中查询出的结果不能满足用户要求时，系统需要在数据源中查询。同时，本文还研究了整合多个查询中间结果的算法。
4) 通过实验，验证了本文所提出并实现的大数据分析处理系统的有效性。首先，验证了该系统的查询处理时间和数据量大小无关，而是和数据的分布有关。然后，验证了本文所提出的树形结构的有效性。最后，通过实验验证了系统返回给用户的置信度和置信区间的准确性。

7.2 未来研究工作展望

本文基本上实现了模糊查询大数据的目标，然而在应用中仍有需要改进的地方。具体问题和未来的研究方向主要有以下几点。

1）系统当前只能处理对应单句 SQL 语句的查询。但是在实际情况中，有很多情况下是需要嵌套查询的。因此，如何将让系统支持嵌套查询将会是该系统的一个重要的拓展方向。

2）本系统现在是基于单机实现的。但是在实际应用中，大数据往往分布式存储在多台服务器中。如何将系统拓展为分布式实现，以适应更大规模的数据，也是一个值得考虑的问题。

3）本系统在整合多个中间统计估计量时，使用了一种近似算法。因此，如何更有效的整合中间统计结果，也是本系统将要深入研究的问题。

参考文献

[1] Hellerstein, J.M., Haas, P.J.,Wang, H.J.: Online Aggregation. In: SIGMOD Conference 1997, pp. 171–182.

[2] P. J. Haas. Large-sample and deterministic confidence intervals for online aggregation. In: Proceedings of the Ninth International Conference on Scientific and Statistical Database Management (SSDBM), 1997.

[3] Haas, P.J., Hellerstein, J.M.: Ripple Joins for Online Aggregation. In: SIGMOD Conference 1999, pp. 287–298.

[4] Luo, G., Ellmann, C.J., Haas, P.J., Naughton, J.F.: A Scalable Hash Ripple Join Algorithm. In: SIGMOD Conference 2002, pp. 252–262 .

[5] Jermaine, C., Dobra, A., Arumugam, S., Joshi, S., Pol, A.: A Disk-Based Join With Probabilistic Guarantees. In: SIGMOD Conference 2005, pp. 563–574

[6] Wander Join: Online Aggregation via Random Walks. Feifei Li, Bin Wu, Ke Yi, Zhuoyue Zhao. SIGMOD 2016, June 26-July 01, 2016, San Francisco, CA, USA

[7] 韩希先，杨东华，李建中．海量数据上的近似连接聚集操作[J]．计算机学报．2010（10）：1919-1936.

[8] Wu, S., Ooi, B.C., Tan, K.L.: Continuous Sampling for Online Aggregation over Multiple Queries. In: SIGMOD Conference, pp. 651–662 (2010).

[9] 安明远，孙秀明，孙凝晖．动态分片在线聚集[J]．计算机研究与发展.47(11):1928-1935, 2010.

[10] PF-OLA: a high-performance framework for parallel online aggregation. C Qin, F Rusu. Distributed and Parallel Databases, Springer, September, Volume 32, Issue 3, pp 337–375, 2014

[11] Wu, S., Jiang, S., Ooi, B.C., Tan, K.L.: Distributed Online Aggregation. PVLDB 2(1), 443–454 (2009).

[12] 程思瑶，姜守旭，李建中．Ｐ２Ｐ网络中时变数据的近似聚集方法化软件学报．2009（07）：1800-1811.

[13] 云环境下的 Max/Min 在线聚集技术研究. 汪凤鸣，慈祥，孟小峰. 《小型微型计算机系统》, 36(10):2177-2182, 2015.

[14] Nikolay Laptev, Kai Zeng, Carlo Zaniolo. Early Accurate Results for Advanced Analytics on MapReduce. VLDB Endowment, Vol. 5, No. 10.

[15] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters[J]. Communications of the ACM, 2008, 51(1):107-113.

[16] Michael Isard, Mihai Budiu, Yuan Yu.Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. EuroSys'07, March 21–23, 2007, Lisboa, Portugal: ACM, 59-72.

[17] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica. Spark: Cluster Computing with Working Sets. Proceedings of the 2nd USENIX conference on Hot topics in cloud computing(HotCloud). 2010.

[18] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Proceedings of the 9th USENIX conference on Networked Systems Design and Implement(NSDI). San Jose, CA:USENIX Association, 2012.

[19] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In SIGMOD, pages 539–550, 2003.

[20] Acharya, S., Gibbons, P.B., Poosala, V.: Congressional Samples for Approximate Answering of Group-By Queries. In: SIGMOD Conference, pp. 487–498 (2000).

[21] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. R. Narasayya. Overcoming limitations of sampling for aggregation queries. In ICDE, 2001.

[22] C. Jermaine, A. Pol, and S. Arumugam. Online maintenance of very large random samples. In SIGMOD Conference 2004, pages 299–310.

[23] M. Riedewald, D. Agrawal, and A. E. Abbadi. pcube: Update-efficient online aggregation with progressive feedback and error bounds. In SSDBM, 2000.

[24] W.Hoeffding. Probability inequalities for sums of bounded random variables. J.Amer.Statist., 58: 13-30,1963.

[25] P.J.Hass. Hoeffding inequalities for join-selectivity estimation and online aggregation. IBM Research Report RJ 10040, IBM Almaden Research Center, San

Jose, CA, 1996.

[26] P. Billingsley. Probability and Measure. Wiley, New York, second edition, 1986.

[27] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. PVLDB, 2(1):277–288, 2009.

[28] Spiegel Joshua, Polyzotis Neoklis. Tug synopses for approximate query answering. ACM Transactions on Database Systems, 2009,34(1):3.

[29] P. J. Haas, J. F. Naughton, and A. N. Swami. On the relative cost of sampling for join selectivity estimation. In Proc. Thirteenth ACM SIGACT-SIGMOD-SIGART Symp. Princi-ples of Database Sys., pages 14{24. ACM Press, 1994.

[30] W. Hou, G. Ozsoyoglu, and B. Taneja. Processing aggregate relational queries with hard time constraints. In Proc. 1989 ACM SIGMOD Intl. Conf. Managment of Data, pages 68-77. ACM Press, 1989.

[31] Acharya, S., Gibbons, P.B., Poosala, V., Ramaswamy, S.: Join Synopses for Approximate Query Answering. In: SIGMOD Conference 1999, pp. 275–286

[32] A. Jacobs. The pathologies of big data. Commun. ACM, 52(8):36–44, 2009.

[33]J. K. Patel and C. B. Read. Handbook of the normal distribution. 1996.

攻读硕士学位期间主要的研究成果

致谢

转眼研究生生涯即将结束，在这毕业之际，我心里真是感慨万千。在刚踏入浙大校园的时候，我就告诉自己，一定要珍惜这来之不易的机会，在学校里绝不虚度光阴，争取学习到更多有用的知识，为之后的人生道路打下坚实基础。在实验室的这几年里，我学到了很多，认识了许多志同道合的同学。我非常感谢在求学路上给予我帮助的老师们和同学们。

感谢我的导师伍赛老师在专业方向上给我的指导，帮我指明了未来发展道路。感谢寿黎但老师和陈刚老师，在科研方面给了我很多宝贵的意见。感谢陈珂老师在各方面工作以及生活上的关心和指导。感谢唐颖红老师和郑倩老师在实验室日常生活中，给我提供帮助。

感谢实验室的庞志飞、吴参森、陈鸿翔等师兄在学习以及生活中给我的帮助和支持。感谢同届的于志超、张也、冯杰、刘伟、吴联坤、王改革、李邦鹏、王伟迪、胡凡、钱宇、朱华、朱清华、周俊林、吴晓晓等同学，你们在生活和学习中给我带来了无数欢乐。还有其他实验室的师弟师妹们，有了你们的陪伴，我的生活变得更加充实和愉快。

感谢我的父母对我学习和生活的支持，以及无声无息的爱。感谢你们无私的付出，感谢你们教会了我坚持和努力，感谢你们带给我的一切。

最后，感谢评阅、评议论文的各位专家学者，在百忙之中抽出时间给我指导。

金明健

2017 年 1 月 15 日