分类号 TP274　　　　　　　　　　　　密级 公开
U D C 004.62　　　　　　　　　　　　编号 10741

硕 士 学 位 论 文

论文题目 高效用项集挖掘算法改进研究

研 究 生 姓 名：张晓蝶

指导教师姓名、职称：杨海军 教授

学 科、专 业 名 称：管理科学与工程

研 究 方 向：物流信息系统分析与应用

提 交 日 期：2024 年 5 月 31 日

独创性声明

本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，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。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了谢意。

学位论文作者签名：张晓蝶　　签字日期：2024.5.31

导师签名：　　签字日期：2024.5.31

关于论文使用授权的说明

本人完全了解学校关于保留、使用学位论文的各项规定，同意（选择“同意”/“不同意”）以下事项：

1.学校有权保留本论文的复印件和磁盘，允许论文被查阅和借阅，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文；

2.学校有权将本人的学位论文提交至清华大学“中国学术期刊（光盘版）电子杂志社”用于出版和编入CNKI《中国知识资源总库》或其他同类数据库，传播本学位论文的全部或部分内容。

学位论文作者签名：张晓蝶　　签字日期：2024.5.31

导师签名：　　签字日期：2024.5.31

Research on improving high utility itemsets mining algorithms

Candidate : Zhang Xiaodie

Supervisor : Yang Haijun

摘 要

高效用项集挖掘考虑项的重要性、利润、用户偏好等因素，给项赋予不同的权重以计算项集的效用，更加满足实际应用的需求。高效用项集挖掘领域是当前数据挖掘的研究热点之一。目前，数据的爆炸式增长给高效用项集挖掘算法的研究提出了新的挑战。

通过文献分析发现，采用 utility-list 数据结构的一类高效用项集挖掘算法，可以进一步减少在挖掘过程中的连接操作。本文对使用 utility-list 的经典算法 HUI-Miner 和 ULB-Miner 在连接操作等方面进行改进，分别提出了改进算法 HUIwTMS-Miner 和 ULBwTMS-Miner，并对其进行实验验证。由于单机多核处理器是实现复杂并行计算系统的基础之一，在单机多核处理器环境下的研究可以从微观的角度揭示并行计算的原理。本文在单机多核处理器环境下，利用 Thread Pool 框架分别设计、实现了 ULB-Miner 算法和 ULBwTMS-Miner 算法的并行算法 PULB-Miner 和 PULBwTMS-Miner。主要研究工作如下：

（1）结合投影数据库技术和事务合并策略，分别对经典算法 HUI-Miner、ULB-Miner 进行改进。根据$\succ_T$顺序对事务进行排序，使用剩余效用剪枝对 1-项集进行二次剪枝，减少投影数据库的生成数量，同时对相同事务进行合并。本研究在 SPMF 公开资源库中的 Retail、Pumsb、Connect、Mushroom、eCommerce 数据集上分别设定了 5 个不同最小效用阈值，进行了算法性能对比实验。实验结果显示，HUIwTMS-Miner 算法除在极稀疏数据集 Retail 外的稀疏数据集 eCommerce 和稠密数据集 Pumsb、Connect、Mushroom 上的运行时间明显减少，而内存消耗有所增加；ULBwTMS-Miner 算法在稠密数据集 Pumsb、Connect、Mushroom 上的运行时间明显减少，在除 Retail 外的其他数据集上的内存消耗有所增加。

（2）设计了 ULB-Miner 算法的并行算法 PULB-Miner。在单机多核处理器环境下，利用 Thread Pool 框架实现 PULB-Miner 算法。分别在 Retail、Pumsb、Connect、Mushroom、eCommerce 数据集的不同最小效用阈值、不同线程进行实验，对比分析 ULB-Miner 算法、PULB-Miner 算法的性能。实验结果显示，PULB-Miner 算法在五个数据集上均表现良好，表明该算法具有可行性和有效性。

（3）设计了 ULBwTMS-Miner 算法的并行算法 PULBwTMS-Miner。在单机多核处理器环境下，利用 Thread Pool 框架实现 PULBwTMS-Miner 算法。分别在 Retail、Pumsb、Connect、Mushroom、eCommerce 数据集的不同最小效用阈值、不同线程进行实验，对比分析了 ULBwTMS-Miner 算法和 PULBwTMS-Miner 算法的性能，同时还对比分析了 PULB-Miner 算法和 PULBwTMS-Miner 算法的性能。实验结果显示，PULBwTMS-Miner 算法在运行时间上明显低于 ULBwTMS-Miner 算法，内存消耗略有增加，在稠密数据集 Pumsb、Connect、Mushroom 上的运行时间明显低于 PULB-Miner 算法，内存消耗也减少，表明该算法具有可行性和有效性。

关键词：高效用项集 投影数据库 事务合并 并行计算 Thread Pool

Abstract

High utility itemsets mining considers factors such as the importance, profit, and user preferences of items, assigns different weights to items to calculate the utility of itemsets, and better meets the needs of practical applications. The field of high utility itemsets mining is one of the current research hotspots in data mining. At present, the explosive growth of data poses new challenges to the research of high utility itemsets mining algorithms.

Through literature analysis, it was found that a class of high utility itemsets mining algorithms using utility-list data structures can further reduce the number of connection operations during the mining process. This article improves the classic algorithms HUI-Miner and ULB-Miner using utility-list in connection operations, and proposes improved algorithms HUIwTMS-Miner and ULBwTMS-Miner, respectively, and conducts experimental verification on them. Due to the fact that single machine multi-core processors are one of the foundations for implementing complex parallel computing systems, research in single machine multi-core processor environments can reveal the principles of parallel computing from a microscopic perspective. This article designs and implements parallel algorithms PULB-Miner and PULBwTMS-Miner for ULB-Miner and ULBwTMS-Miner algorithms

using the Thread Pool framework in a single machine multi-core processor environment. The main research work is as follows:

(1) By combining projection database technology and transaction merging strategy, improvements are made to the classic algorithms HUI-Miner and ULB-Miner, respectively. Sort transactions in $\succ_T$ order, use residual utility pruning to perform secondary pruning on the 1-itemset, reduce the number of generated projection databases, and merge the same transactions. This study set 5 different minimum utility thresholds on the Retail, Pumsb, Connect, Mushroom, and eCommerce datasets in the SPMF public resource library and conducted algorithm performance comparison experiments. The experimental results show that the HUIwTMS-Miner algorithm significantly reduces running time on sparse datasets eCommerce and dense datasets Pumsb, Connect, and Mushroom, except for the extremely sparse dataset Retail, while increasing memory consumption; The ULBwTMS-Miner algorithm significantly reduces runtime on dense datasets such as Pumsb, Connect, and Mushroom, while increasing memory consumption on datasets other than Retail.

(2) Designed a parallel algorithm called PULB-Miner for the ULB-Miner algorithm. Implement the PULB-Miner algorithm using the Thread Pool framework in a single machine multi-core processor environment. Conduct experiments on different minimum utility thresholds and threads on the Retail, Pumsb, Connect, Mushroom, and

eCommerce datasets to compare and analyze the performance of ULB-Miner algorithm and PULB-Miner algorithm. The experimental results show that the PULB-Miner algorithm performs well on all five datasets, indicating its feasibility and effectiveness.

(3) Designed a parallel algorithm called PULBwTMS-Miner for the ULBwTMS-Miner algorithm. Implement the PULBwTMS-Miner algorithm using the Thread Pool framework in a single machine multi-core processor environment. We conducted experiments on different minimum utility thresholds and threads on the Retail, Pumsb, Connect, Mushroom, and eCommerce datasets to compare and analyze the performance of ULBwTMS-Miner algorithm and PULBwTMS-Miner algorithm. We also compared and analyzed the performance of PULB-Miner algorithm and PULBwTMS-Miner algorithm. The experimental results show that the PULBwTMS-Miner algorithm has significantly lower runtime than the ULBwTMS-Miner algorithm, with a slight increase in memory consumption. The runtime on dense datasets such as Pumsb, Connect, and Mushroom is significantly lower than the PULB-Miner algorithm, and memory consumption is also reduced, indicating the feasibility and effectiveness of this algorithm.

Keywords： High utility itemset;Projected batabase; Transaction merging; Parallel computing;Thread Pool

目　录

1 绪论 1
1.1 研究背景与意义 1
1.2 国内外研究现状 2
1.2.1 频繁项集挖掘算法 2
1.2.2 高效用项集挖掘算法 4
1.3 主要研究内容 8
1.4 论文组织结构 9
1.5 研究创新点 10
2 相关理论研究 11
2.1 关联规则 11
2.2 预备知识 12
2.2.1 频繁项集 12
2.2.2 高效用项集 12
2.3 并行计算及其框架 15
2.3.1 并行计算概念 15
2.3.2 并行计算框架 16
2.3.2.1 Thread Pool 16
2.3.2.2 Fork/join 18
2.3.2.3 Hadoop 20
2.3.2.4 Spark 23
2.4 数据集介绍 25
2.5 本章小结 25
3 改进的高效用项集挖掘算法研究 27
3.1 Utility-List 数据结构 27
3.2 HUI-Miner 和 ULB-Miner 算法介绍 29
3.2.1 HUI-Miner 算法 29

3.2.2 ULB-Miner 算法 33
3.3 改进策略 39
3.3.1 投影数据库 39
3.3.2 事务合并策略 40
3.4 改进算法介绍 41
3.4.1 改进算法 HUIwTMS-Miner 41
3.4.1.1 HUIwTMS-Miner 算法描述 41
3.4.1.2 实验结果与分析 46
3.4.2 改进算法 ULBwTMS-Miner 49
3.4.2.1 ULBwTMS-Miner 算法描述 49
3.4.2.2 实验结果与分析 52
3.5 本章小结 55
4 并行 ULBwTMS-Miner 算法研究 56
4.1 并行算法设计原则 56
4.1.1 并行基本概念 57
4.1.2 分解技术 57
4.1.3 并行算法模型 58
4.2 并行 PULB-Miner 算法研究 59
4.2.1 Thread Pool 框架下 PULB-Miner 算法设计与实现 60
4.2.2 实验结果与分析 62
4.3 并行 PULBwTMS-Miner 算法研究 65
4.3.1 Thread Pool 框架下 PULBwTMS-Miner 算法设计与实现 65
4.3.2 实验结果与分析 66
4.4 本章小结 72
5 总结与展望 73
5.1 工作总结 73
5.2 工作展望 74
参考文献 75

攻读硕士学位期间发表的论文及科研情况 80

致 谢 81

1 绪论

1.1 研究背景与意义

在信息共享越来越普遍的大数据时代，获取信息的方式更加方便、多样，从这些海量的数据中发现有价值的信息，把这些数据转化成有组织的知识是一种重要需求。从信息处理的角度，利用计算机帮助分析数据和理解数据，基于丰富的数据做出决策，这种追求导致了数据挖掘的诞生。

数据挖掘[1]可以应用到诸多领域，例如：金融数据分析、零售和电信业的数据挖掘、科学与工程数据挖掘、行为分析、入侵检测和预防数据挖掘等等。数据挖掘涉及许多相关研究，例如聚类分析[2]-[3]、分类分析[4]-[5]、模式挖掘[6]-[7]、预测[8]等。其中，模式挖掘一直是研究热点，其主要的研究内容有：频繁项集挖掘[9](Frequent Itemsets Minging,FIM)、关联规则挖掘[10](Association Rule Minging,ARM)、高效用项集挖掘[11](High-Utility Itemsets Mining,HUIM)、序列模式挖掘[12](Sequential Pattern Mining,SPM)等等。其中，频繁项集挖掘和高效用项集挖掘是目前数据挖掘中最重要和最常见的任务，因为它们满足众多领域和应用的要求，也能给决策者提供有价值的信息。

频繁项集挖掘假定事务中每个项的价值都相同,用于发现出现次数最多的项集[13]-[14]。由于其简单高效的特点，频繁项集挖掘应用于购物篮分析[15]、生物信息、行为分析、互联网、异常检测[16]-[20]等多个领域。然而，频繁项集挖掘存在局限性，不足以向决策制定者提供全部有价值信息，因此学者们又提出了数据的效用模型。高效用项集挖掘是在频繁项集挖掘的基础上发展而来的[11]，其不仅考虑项集的频繁程度，还考虑用户偏好、重要性、利润等因素对项集“有效性”影响。项集的效用由它的外部效用（例如，权重/单位利润/兴趣度）和内部效用（例如，购买数量/发生次数/重要性）来衡量。因此，高效用项集挖掘更能满足人们对事物“有用性”的要求，被视为频繁项集挖掘的拓展，成为近年来数据挖掘领域被广泛关注的重要研究课题。

随着数据挖掘研究与应用迅猛发展，新的算法不断出现，算法的数据结构、适用范围也在不断丰富，随之扩展的改进算法也使得挖掘效率有很大提升。但是数据的爆炸式增长，使得我们能够获得的数据倍增，搜索空间呈指数级增长，挖

掘这些数据需要更大的内存和更快的挖掘速度。因此，进一步提升现有算法的性能、制定合适的挖掘方案，从待处理的数据中挖掘高效用项集，提取有价值的信息，依旧是数据挖掘领域需要解决的难题。

1.2 国内外研究现状

1.2.1 频繁项集挖掘算法

Apriori 算法[21]、FP-growth 算法[22]和 Eclat 算法[23]是频繁项集挖掘领域最具有影响的三个算法，许多学者在其基础上，通过优化其数据结构、搜索策略，产生一批改进算法，它们的算法性能和适用范围都有不错的改进效果。

（1）Apriori 算法及其改进算法

Apriori 算法是由 Agrawal 提出的，主要采用 1-项集、2-项集、...、n-项集的自底向上“逐层搜索”的方法，利用预先设定的最小支持度进行剪枝，频繁 k-项集中的项两两组合进行自连接操作，进而得到候选(k+1)-项集，每当有一次候选项集产生就要遍历一次数据库计算支持度，而且在算法的执行过程中，产生的候选项集数量过多，浪费内存空间，降低挖掘效率。

AprioriTID 算法[21]是在 Apriori 算法的基础上改进，仅遍历一次数据库生成 Tid 表来计算候选项集的支持度，使用集合 C_k=(Tid,{X_k})完成，其中每个 X_k 都是一个 k-项集，在标识符为 Tid 的事务中；用逐渐缩小的 Tid 表来代替原有的事务数据库，从而减少所要扫描的数据量。

Ashok 采用了 Apriori 算法和 AprioriTID 算法的基本思想，提出了 Partition 算法[24]。算法分两阶段：第一阶段将数据库划分为若干不重叠的分区，第一次扫描数据库生成分区频繁项集，合并每个分区的频繁项集生成一个潜在的频繁项集；第二阶段第二次扫描数据库，生成潜在频繁项集的实际支持度，识别频繁项集。该算法随事务数量线性扩展。

（2）Eclat 算法及其改进算法

Eclat 算法是由 Zaki 提出的，采用垂直数据表示的方法，将数据按照项集存储，每条记录包括一个项集和包含它的 Tid 集合，利用基于前缀的等价关系将搜索空间划分为较小的子空间，由两个频繁 k-项集求并集得到候选(k+1)-项集，

通过对候选(k+1)-项集的 Tid 集求交集得到(k+1)-项集的支持度计数，判断是否是频繁项集。

由于 Eclat 算法在稠密数据集求交集占用过度空间且产生大量冗余，Zaki 提出了改进算法 Diffsets 算法[25]，用求差集代替候选项之间的交集操作。在数据集较稠密时，避免了存储候选项集的每个事务标识。

（3）FP-Growth 算法及其改进算法

FP-Growth 算法是由 Han 提出来的，可以克服 Apriori 算法重复扫描数据库的不足。该算法采用深度优先遍历的方法，只需要对数据库进行 2 次扫描且不产生候选模式集，采用高度压缩的 FP-Tree 数据结构，根据条件模式基生成条件 FP-Tree，路径上的所有项的组合即所求结果。该算法提出的 FP-Tree 数据结构在频繁项集挖掘领域广泛应用。

Borgelt 在 FP-Growth 算法的基础上改进，提出 Relim 算法[26]，在运行时不必创建频繁模式树，而是通过建立一个事务链表组 TLs(Transaction Lists)来找出所有的频繁项集，该事务链表组中的各项事务链表 TL(Transaction List)保存着头元素相同的各事务的相关信息，并且按照头元素支持度大小递增排列，依次对 TLs 中的每个 TL 进行挖掘。Relim 算法只适合挖掘短模式的数据库，空间利用率高，对最小支持度高或者频繁规则比较少的数据集进行数据挖掘时，可以节约大量的运算时间。

Pei 提出的 H-Mine 算法[13]也在一定程度上改善了在稠密数据集求交集占用过度空间且产生大量冗余的问题。该算法通过遍历数据集将数据转换为新的数据结构 H-struct，根据首项数据对数据进行划分，深度优先遍历，每次只挖掘一个数据分区，最后将每个分区结果进行汇总得出频繁项集。H-Mine 算法有极好的可扩展性且综合性能都优于 Apriori 算法和 FP-Growth 算法。

为了适用于稠密数据集和海量数据集挖掘，Deng 先后提出了三种算法：(1)PPV 算法[27]，提出新的数据结构 Node-list 和 Poc-tree；(2)Prepost 算法[28]，提出新的数据结构 N-list 和 PPC-tree；(3)FIN 算法[29]，提出新的数据结构 Nodeset，利用 Poc-tree 挖掘频繁 1-项集和频繁 2-项集，并初始化 Nodeset，在 k-项集(k＞2)利用超集等价性对集合枚举树[30]进行剪枝，发现频繁项集。数据结构 Node-list 和 N-list 都需要使用预排序和后排序对节点进行编码且不适合两个短项

集生成一个长项集，影响算法效率。FIN 算法解决了编码问题，通过超集等价性可以高度修剪搜索空间，从而避免大量的冗余，在稠密数据集和海量数据集上表现更好。

1.2.2 高效用项集挖掘算法

高效用项集挖掘是 Chan 在 2003 年首次提出来的[11]，不仅考虑事务数据库中项出现的次数，还考虑项的权重，以解决频繁项集中基于支持度挖掘的关键限制。虽然高效用项集挖掘是频繁项集挖掘的拓展，但是高效用项集挖掘算法和频繁项集挖掘算法并不相同。

现有的高效用项集挖掘算法根据是否生成候选项集可分为两阶段算法和一阶段算法。两阶段算法主要通过在第一阶段找到候选的高效用项集，第二阶段进行验证。一阶段算法则直接生成高效用项集。

最典型的两阶段算法--Two-Phase 算法是由 Liu 等人提出的[31]-[32]。该算法首次提出了事务加权效用(transaction-weighted urtilization,TWU)模型，即一个项集的 TWU 值大于等于预先设定的最小效用阈值 min-util，那么该项集就是一个候选项集。算法在第一阶段，遵循水平遍历搜索，使用 TWU 模型生成候选项集。在第二阶段，通过多次扫描数据库计算候选项集的效用值，并与给出的 min-util 进行比较，输出所有的高效用项集。由于 Two-Phase 算法使用 Apriori 算法相同的挖掘原理，也存在与 Apriori 算法同样的缺点，即无法避免多次重复扫描数据库，产生大量候选集，导致计算量庞大和内存空间浪费。

Ahmed 等人为了避免多次扫描数据库，提出 IHUP 算法[33]。该算法仅需要扫描两次数据库，是基于树的高效用项集挖掘算法，提出了新的数据结构 IHUP-tree。第一次扫描数据库，在 IHUP-tree 的每个节点保存各项的支持度和事务加权效用值，采用模式增长的方式生成候选项集，一定程度上节约了内存，但 IHUP 算法仍然会产生大量的候选项集，影响算法的挖掘效率。

为了解决生成大量候选项集的问题，Vincen S.Tseng 等人提出了 UP-Growth[34] 算法。算法提出了一种紧凑的树结构 UP-Tree 和四种剪枝策略：DGU(Discarding Global Unpromising items)、DGN(Discarding Global Node utilities)、DLU(Discarding Local Unpromising items)、DLN(Discarding Local Node utilities)。通过这四种剪枝

策略，从全局 UP-Tree 和条件 UP-Tree 中删除低效用项，极大减少了候选项集的数量。接着，在 UP-Growth 算法的基础上提出了两个新的剪枝策略：DNU 和 DNN，改进后的算法命名为 UP-Growth+[35]。在该算法中使用节点效用替换项效用，进一步降低上界，可以更有效地过滤低效用项和项集。

Lin 等人提出了 HUP-Growth 算法[35]，该算法集成了两阶段过程和 FP-Tree 的概念，通过两次数据库扫描，利用基于 TWU 值的向下闭合特性并生成压缩的树结构，并且对三种不同的项排序方法生成的树节点数量进行了比较，结果表明，频率排序生成的树结点少于字典排序和 TWU 值排序。

尽管上述算法都应用剪枝策略减小了搜索空间，但依旧无法避免生成含有大量非高效用项集的候选项集，对挖掘效率仍有一定影响。对于这一问题，Liu 等人提出 HUI-Miner 算法[37]。HUI-Miner 算法是第一个一阶段的高效用项集挖掘算法，采用效用列表（Utility-List)数据结构，通过构建所有项的 Utility-List，使用 Utility-List 执行连接操作挖掘所有高效用项集。实验结果表明，HUI-Miner 算法在时间性能和内存占用情况均优于生成候选项集的算法。

Liu 等人在 HUI-Miner 算法的基础上提出 d2HUP 算法[38]。d2HUP 算法将数据库转化为基于树的结构 CAUL(Chain of Accurate Utility Lists)，通过前缀扩展枚举项集，使用 CAUL 可以有效地获得每个枚举项集的效用和效用的上界，以此来修剪搜索空间。算法还引入稀疏数据的递归无关项过滤和密集数据的前瞻性策略，显著提高算法效率。

Fournier-Viger 等人在 HUI-Miner 算法的基础上提出 FHM 算法[39]。FHM 算法构建数据结构 EUCS(Estimated Utility Co-occurence Structure), 用来保存所有 2-项集的 TWU 值，使用 EUCP(Estimated Utility Co-occurence Pruning)策略剪去大量低效用的 2-项集及其超集，极大减少了 Utility-List 的连接次数和运行内存的占用，提高了算法效率。

Zida 提出 EFIM 算法[40]。该算法利用高效的数据库投影 (High-utility Database Projection,HDP)和事务合并（High-utility Transaction Merge，HTM)技术，降低数据库扫描的成本，依赖子树效用(Sub-tree Utility,SU)和局部效用(Local Utility,LU)两个上界进行剪枝。同时，引入一种基于数组的效用计数技术(Fast Utility Counting,FAC), 计算在线性时间和空间的上界。EFIM 算法具有较低内存消耗。

Dawar 提出的 UFH 算法[41]。该算法是一个混合框架，其可以组合任何基于树的递归算法和基于 Utility-List 的算法。在真实的和合成的数据集上进行验证，结果表明，算法在稀疏的数据集上性能优于当前最先进的算法。

Duong 等人为了减小 Utility-List 连接操作的成本，提出了 ULB-Miner 算法[42]，使用设计的效用列表缓冲区结构来有效地存储和检索 Utility-List，并在挖掘过程中重用内存, 结合 EA(Early Abandoning)剪枝策略[43]，对满足特定条件的项不再生成效用列表。算法还介绍了一种在效用列表缓冲区构造效用列表段的线性时间方法。算法在稠密数据集和稀疏数据集上都表现良好。

Wu 等人在 EFIM 算法的基础上提出 HUI-PR 算法[44]。通过构建一种新的树结构，并设计两种剪枝策略 slocU(strict local utility)和 ssubU(strict sub-tree utility), 减少生成的候选项集的数量。该方法生成的搜索空间分支更少，有效地减少了在巨大数据集中的内存使用。

为了进一步减少 Utility-List 存储项集信息占用的时间和内存，Shen 等人提出 EHUIM-DS 算法[45]。EHUIM-DS 算法通过重组事务数据库，设计新的数据结构 ITD(ITems Data)，其中每条记录表示单项的相关事务的所有值，又引入了扩展效用和局部 TWU 作为上界，从宽度和深度上大大减少了搜索空间。

Wu 等人为了改进 Utility-List 构建过程，提出了 UBP-Miner 算法[46]。算法提出一种新的位运算方法 BEO(Bit mErge cOnstruction)来加速 Utility-List 构建过程。此外，还设计了新的数据结构 UBP(Utility Bit Partition)来支持 BEO，并与四种剪枝策略相结合。结果表明，其连接操作的时间复杂度对于数据库的大小是不变的。

表 1.1 对以上经典的高效用项集挖掘算法进行归纳总结。

表 1.1 经典的高效用项集挖掘算法总结

算法	阶段	数据结构	剪枝策略	延伸	特点
Two-phase（2005 年）	两阶段	-	TWU	Apriori	生成大量的候选项集，且需要进行多次数据库扫描
IHUP（2009 年）	两阶段	$IHUP_L$-Tree、$IHUP_{TF}$-Tree、$IHUP_{TWU}$-Tree	TWU	Two-phase	两次数据库扫描，具有可扩展性，适合交互式挖掘

续表 1.1 经典的高效用项集挖掘算法总结

算法	阶段	数据结构	剪枝策略	延伸	特点
UP-Growth（2010 年）	两阶段	UP-Tree	DLU、DLN、DGU、DGN	Two-phase	特别是数据库包含大量的长事务时，表现优异
HUP-Growth（2011 年）	两阶段	HUP-Tree	TWU	Two-phase、FP-Growth	执行时间优于 Two-phase 算法，并验证按照频率排序的树节点少于其他两种方法，不适用于长事务
HUI-Miner（2012 年）	一阶段	Utility-List	TWU、Remaining Utility	Eclat	第一个一阶段算法，不生成候选项集,通过 Utility-List 连接操作进行挖掘
d2HUP（2012 年）	一阶段	Utility-List、CAUL	TWU、Remaining utility upper-bound	HUI-Miner	树结构和 CAUL 消耗更多的内存，在前瞻性策略用处不大的地方，算法效率低下
UP-Growth+（2012 年）	两阶段	UP-Tree	DGU、DGN、DNU、DNN	UP-Growth	特别是数据库包含大量的长事务或设定较低的最小效用阈值时，表现优异
FHM（2014 年）	一阶段	Utility-List、EUCS、Hash-map	TWU、EUCP、Remaining Utility	HUI-Miner	进行 2-项集剪枝，大量减少 Utility-List 连接操作的数量
EFIM（2015 年）	一阶段	Utility-binary array	TWU、LU、SU	UP-Growth	使用 HDP 和 HTM 进行优化，递归投影有时会很耗时，在极稀疏数据集不能很好的扩展
UFH（2017 年）	-	Utility-List、树	-	-	一个可以结合任何基于树的递归算法与基于 Utility-List 的算法的混合框架，适用于稀疏数据集
ULB-Miner（2018 年）	一阶段	Utility-List buffer、EUCS、Hash-map	TWU、EA、Remaining Utility、EUCP	HUI-Miner	在稀疏数据集和稠密数据集都表现良好

续表 1.1 经典的高效用项集挖掘算法总结

算法	阶段	数据结构	剪枝策略	延伸	特点
HUI-PR（2019 年）	一阶段	Utility-Tree	TWU、slocU、ssubU	EFIM	增加了新提出的上界的复杂性，不能有效地修剪候选项集，在稀疏数据集的表现不好
EHUIM-DS (2021 年)	一阶段	ITD	TWU、local TWU、Extension utility	EFIM、d2HUP	两个上界：扩展效用和局部 TWU，从宽度和深度上大大减少了搜索空间
UBP-Miner（2022 年）	一阶段	UBP、EUCS	TWU、Suru、PU、LA、EUCP	HUI-Miner	加快效用列表的构建，但是需要额外的内存

1.3 主要研究内容

高效用项集挖掘算法按照挖掘过程可以划分为两阶段算法和一阶段算法，采用的数据结构主要为树和 Utility-List。通过使用不同的数据结构、搜索策略和剪枝策略，算法性能和适用范围都有所不同。文献[40]中的事务合并策略在算法挖掘过程中根据$\succ_T$顺序将事务进行预先排序，并对相同事务进行合并。事务合并策略可以减小数据库的大小和读取数据库的成本，并在基于树的算法中被证实有效。本文将投影数据库技术和事务合并策略应用到使用 Utility-List 数据结构的 HUI-Miner、ULB-Miner 算法中，并对其效率进行验证。并在单机多核处理器环境下，利用 Thread Pool 框架分别设计、实现了 ULB-Miner 算法和 ULBwTMS-Miner 算法的并行算法 PULB-Miner 和 PULBwTMS-Miner。

本文的研究内容主要总结如下：

（1）本文提出了两个改进算法：基于 HUI-Miner 算法的改进算法 HUIwTMS-Miner（High Utility Itemset Miner with Transaction Merge Strategy)和基于ULB-Miner算法的改进算法ULBwTMS-Miner（Utility-List Buffer for high utility itemset Miner with Transaction Merge Strategy)。首先，根据$\succ_T$顺序对数据库中的事务进行排序；然后，结合投影数据库技术，将基于剩余效用的剪枝策略应用到投影数据库的构建阶段，对 1-项集进行二次筛选，不再对所有超集都不是高效

用的项构建投影数据库，并在构建过程中对相同事务合并；最后，依次挖掘生成的投影数据库。

（2）本文基于并行理论设计 ULB-Miner 算法的并行算法 PULB-Miner（Parallel Utility-List Buffer for high utility itemset Miner)，并验证了其可行性和效率。在单机多核处理器环境下，利用 Thread Pool 框架实现了并行算法 PULB-Miner，并对其性能进行验证。

（3）本文根据并行算法设计相关理论，设计了 ULBwTMS-Miner 算法的并行算法 PULBwTMS-Miner(Parallel Utility-List Buffer for high utility itemset Miner with Transaction Merge Strategy)，并验证了其可行性和效率。在单机多核处理器环境下，利用 Thread Pool 框架实现了并行算法 PULBwTMS-Miner，同时验证了其性能。

1.4 论文组织结构

本文分为五个章节，具体内容如下：

第一章为绪论。介绍本文的研究背景与意义、国内外研究现状、主要研究内容、论文组织结构和研究创新点。

第二章为相关理论研究。介绍了关联规则相关知识、频繁项集和高效用项集的相关定义和性质、数据集和并行计算框架，其中并行计算框架重点介绍了适用于单机多核处理器环境并行的 Thread Pool 框架和 Fork Join 框架，并对适用于多处理机环境并行的 Hadoop 框架和 Spark 框架进行简要说明。

第三章为改进的高效用项集挖掘算法研究。首先介绍 Utility-List 数据结构、HUI-Miner 算法和 ULB-Miner 算法的原理和实现步骤；然后介绍改进策略，主要给出了投影数据库和事务合并策略的相关知识；最后介绍改进算法 HUIwTMS-Miner 和 ULBwTMS-Miner，并分别在 5 个特征不同的标准数据集上选取 5 个不同的最小效用阈值进行对比实验，分析 HUIwTMS-Miner 算法和 ULBwTMS-Miner 算法的时间性能和内存性能，验证算法的有效性。

第四章为并行 ULBwTMS-Miner 算法研究。首先从分解技术和并行算法模型两方面介绍并行算法设计原则；然后对 ULB-Miner 算法进行并行计算做可行性分析，基于并行理论，先后设计了并行算法 PULB-Miner 和 PULBwTMS-Miner，

最后在单机多核处理器环境下，在 Thread Pool 框架上验证了其可行性和性能。

第五章为总结与展望。对本文进行工作总结，同时对未来的研究进行工作展望。

1.5 研究创新点

（1）提出 HUI-Miner 算法和 ULB-Miner 算法的改进算法 HUIwTMS-Miner 和 ULBwTMS-Miner。结合投影数据库技术，将事务合并策略应用到采用 Utility-List 数据结构的算法中。采用 Utility-List 数据结构的算法用 Tid 做索引进行数据挖掘，根据排序后的数据库划分投影数据库，并在投影数据库上对事务进行合并可以大量减少 Utility-List 中 Tid 索引的数量，从而减少 Utility-List 的连接操作，提高算法挖掘效率。对比实验在公开数据集上进行验证分析。

（2）设计实现了 ULB-Miner 算法和 ULBwTMS-Miner 算法的并行算法，并在单机多核处理器环境下，利用 Thread Pool 框架实现验证。

2 相关理论研究

2.1 关联规则

关联规则(Association Rules)是数据挖掘中的一项重要技术，它可以反映数据库中事物之间的相互依存性和关联性，用于从海量数据中挖掘有价值的信息。

设 $L = \{I_1, I_2, \dots, I_m\}(1 \le i \le m)$是项的集合，数据库 $D = \{T_1, T_2, \dots, T_n\}(1 \le j \le n)$，其中每个$T_j$都是非空项集，使得 T⊆L。每个事务$T_j$都有唯一的标识符，记为 TID。关联规则是形如 A⇒B 的蕴涵式，其中 A⊂L，B⊂L，A≠∅，B≠∅，并且 A∩B＝∅。规则 A⇒B 在 D 中成立，具有支持度 sup，其中 sup 是 D 中事务包含 A∪B（即集合 A 和 B 的并或 A 和 B 二者）的百分比。它是概率 P(A∪B)。规则 A⇒B 在 D 中具有置信度 conf，其中 conf 是 D 中包含 A 的事务同时也包含 B 的事务的百分比。这是条件概率 P(B|A)[47]。即：

$$\mathrm{Support}(A \Rightarrow B) = P(A \cup B) \tag{1}$$

$$\mathrm{Confidence}(A \Rightarrow B) = P(B \mid A) \tag{2}$$

同时满足最小支持度阈值(min_sup)和最小置信度阈值(min_conf)的规则称为强规则。为方便计算，用 0%-100%之间的值，而不是 0-1.0 之间的值表示支持度和置信度。

关联规则分析应用场景十分广泛，最典型的是“购物篮分析”[15]。“购物篮分析”可以用来解决以下问题：

1. 发现物品之间的关联性：该过程通过发现顾客放入他们“购物篮”中的商品，从而分析两个物品之间的关联性；

2. 预测消费者行为：当消费者购买某一物品时，可以利用关联规则分析来预测他们会购买哪些其他物品；

3. 改进推荐系统：通过发现消费者可能感兴趣的物品，改进现有的推荐系统，提高推荐系统的准确率；

4. 市场营销分析：对市场上消费者的购买行为进行统计分析，从而提出市场营销策略，加强企业竞争力。

2.2 预备知识

2.2.1 频繁项集

项的集合称为项集，包含 K 个项的项集称为 K-项集。项集的出现频度是包含项集的事务数，简称为项集的支持度计数。

表 2.1 展示的是频繁项集的事务数据库示例，事务数据库 $D=\{T_1,T_2,T_3,T_4,T_5\}$，$L=\{a,b,c,d,e,f,g\}$是数据库 D 中所有项的集合。

表 2.1 事务数据库示例

Tid	Transaction
T_1	a b c d e f
T_2	b c d e
T_3	a c d
T_4	a c e g
T_5	b c e g

定义 1 频繁项集[21]。给定最小支持度阈值 min_sup，如果项集 X 的支持度满足预定义的最小支持度阈值（即项集的支持度满足对应的最小支持度计数阈值），则称 X 是频繁项集。

性质 1 若 X 为频繁项集，Y 为 X 的任一非空子集，则 Y 必为频繁项集，且 $\sup(X)\le\sup(Y)$；若 X 为非频繁项集，Y 为 X 的任一超集，则 Y 必为非频繁项集，且 $\sup(Y)\le\sup(X)$。

2.2.2 高效用项集

设 $D=\{T_1,T_2,\ldots,T_n\}(1\le j\le n)$，是一个具有效用值的事务数据库，$L=\{I_1,I_2,\ldots,I_m\}(1\le i\le m)$是事务数据库 D 中所有项的集合，其中每个$T_j$是一个非空项集，使得 T⊆L，$T_j$表示方式为$\{(x_1{:}\,c_1),(x_2{:}\,c_2),\ldots,(x_v{:}\,c_v)\}$（v 为事务项集长度），

其中$x_k \in L(k = 1,2,\ldots,v)$，$c_k$是项$x_k$在数据库中的数量，记为$iu(I_p, T_j)$。每个事务$T_j$都有唯一标识符，称为 TID。

对于项集中的每个项I_p都有两个效用值：内部效用$iu(I_p, T_j)$和外部效用$eu(I_p)$。表 2.2 和表 2.3 给出一个事务数据库示例。

表 2.2 事务表

Tid	Transaction
T_1	(a,1) (b,5) (c,1) (d,3) (e,1) (f,5)
T_2	(b,4) (c,3) (d,3) (e,1)
T_3	(a,1) (c,1) (d,1)
T_4	(a,2) (c,6) (e,2) (g,5)
T_5	(b,2) (c,2) (e,1) (g,2)

表 2.3 外部效用表

Item	a	b	c	d	e	f	g
eu(ip)	5	2	1	2	3	1	1

相关定义描述如下[31][40][42]：

定义 2 项I_p在事务T_j中的效用，记为$u(I_p, T_j)$。$u(I_p, T_j)$是项I_p在事务T_j中的内部效用$iu(I_p, T_j)$与其外部效用$eu(I_p)$的乘积，即：

$$u(I_p, T_j) = iu(I_p, T_j) \times eu(I_p) \tag{3}$$

定义 3 项集 X 在事务T_j中的效用，记为$u(X, T_j)$。$u(X, T_j)$指存在于事务T_j中的项集 X 包含的所有项的效用和，即：

$$u(X, T_j) = \sum_{I_p \in X} u(I_p, T_j) \tag{4}$$

定义 4 项集 X 在数据库 D 中的效用，记为 u(X)。u(X)是在数据库 D 的事务中存在的项集 X 的效用和，即：

$$u(X) = \sum_{T_j \in D} u(X, T_j) \tag{5}$$

定义 5 事务T_j的效用，记为 $TU(T_j)$。$TU(T_j)$是事务T_j包含的所有项的效用和，即：

$$TU(T_j) = \sum_{I \in T_j} u(I, T_j) \tag{6}$$

定义 6 项集 X 的事务权重效用，记为 $TWU(X)$。$TWU(X)$是指包含项集 X 的所有事务的效应和，即：

$$TWU(X) = \sum_{\substack{X \in T_j \\ T_j \in D}} TU(T_j) \tag{7}$$

定义 7 项集 X 的剩余效用，记为 $RU(X, T)$。对于项集 X，$X \subseteq T$，排列在 X 之后的所有项的集合被记为 T/X，则 $RU(X, T)$为 T/X 的效用和，即：

$$RU(X, T) = \sum_{I \in T/X} u(I, T) \tag{8}$$

定义 8 数据库效用，记为 $TUD(D)$。数据库 D 的总效用是所有事务的效用值的和，即：

$$TUD(D) = \sum_{T_j \in D} TU(T_j, T_j) \tag{9}$$

定义 9 高效用项集，记为 HUI。设定一个最小效用阈值 min_util，如果 $u(X) \geq min_util$，则称 X 为高效用项集。

定义 10 估计效用共现结构（Estimated Utility Co-Occurrence Structure,EUCS）。EUCS 结构是一组三元组形式$(a, b, c) \in L*L*R$，一个三元组(a, b, c)表示 $TWU(\{a, b\}) = c$。EUCS 可以实现为如图 2.1 所示的三角形矩阵。

Item	a	b	c	d
b	52			
c	41	39		
d	51	49	38	
e	40	40	24	40

图 2.1 EUCS

在高效用项集挖掘领域，基于 Utility-List 数据结构的算法中，主要使用的剪枝策略有三种：

剪枝策略 1 TWU 剪枝[31]。通过比较项的 TWU 值和最小效用阈值 min_util 进行剪枝：

（1）如果 TWU＜min_util，则该项及其扩展项集都不可能是高效用项集，删除该项，不再进行任何计算；

（2）如果 TWU≥min_util，则该项及其扩展项集有可能是高效用项集，保留该项，并进一步挖掘。

剪枝策略 2 剩余效用剪枝[37]。通过比较项的 sumIutil+sumRutils 值和最小效用阈值 min_util 进行剪枝：

（1）若 X.sumIutils+X.sumRutils≥min_util，则该项及其扩展项集有可能是高效用项集，进一步挖掘；

（2）若 X.sumIutils+X.sumRutils＜min_util，则该项及其扩展项集都不可能是高效用项集，不再进一步挖掘。

需要注意的是，在挖掘投影数据库时使用剪枝策略 2，删除 sumIutils+sumRutils＜min_util 的项，仅仅是删除了以该项为起始项的低效用项集，并不影响该项在其他投影数据库中作为其他项的可扩展项继续存在。

剪枝策略 3 EUCP 剪枝[39]。如果在 EUCS 中没有元组(x,y,c)满足 c≥min-util 时，那么 xy 及其所有超集都是低效用项集。

2.3 并行计算及其框架

在计算机最初发展的十几年里，从体系结构到系统软件以及应用软件都采用串行计算作为主要的设计和开发模式。面对串行计算的性能瓶颈，一个合理的解决方案就是依靠并行化，并行计算就成为提高计算机系统算力的重要发展方向。

2.3.1 并行计算概念

并行机，也可以称作并行计算机，是指机器在同一时间内能够处理多个指令或多条数据。在并行机上所做的计算，称为并行计算，又称高性能计算或超级计算，是相对于串行计算来说的。并行计算将应用分解为多个子任务，并将其分配给不同的处理器，每个处理器相互协作、并行执行，以实现加速求解或提高求解问题规模的目的。并行计算的主要目标是解决一系列具有挑战性的科学、工程和

商业并行计算问题，并满足对速度和内存资源日益增长的需求[48]。

并行计算可分为时间上的并行和空间上的并行。时间上的并行就是指流水线技术，而空间上的并行则是指用多核处理器、多个处理器及计算机集群构成的多处理机系统同时执行计算，多核处理器、多个处理器通过系统互联（由总线、交叉开关或多级网络实现）部件将多个计算内核或多个处理器连接起来，而多处理机系统通过网络将两个以上的处理机连接起来，达到同时计算同一个任务的不同部分，或者解决单个处理机无法解决的大型问题的目的。两种并行方式都可以有效地提高计算资源的利用能力，从而提高程序的执行性能。

在频繁项集和高效用项集挖掘领域最常用的并行计算方法有：（1）计数分布并行；（2）数据分布并行；（3）候选分布并行[49]；（4）投影数据库并行[50]。文献[49]中论证表明，计数分布并行在绝大部分情况下优于数据分布并行和候选分布并行，综合表现最优。

2.3.2 并行计算框架

并行计算环境从处理器部件的角度可分为单机多核处理器并行、多处理器并行和多处理机并行。其中，单机多核处理器指的是单个 CPU 内部有多个计算核心部件；多处理器指的是一台计算机上存在多个处理器；多处理机指的是一个系统里存在多个不同的计算机构成的集群。本小节主要介绍的并行框架是适用于单机多核处理器环境的 Thread Pool 框架和 Fork/Join 框架；适用于多处理机环境的 Hadoop 框架和 Spark 框架。

2.3.2.1 Thread Pool

对于要处理的任务而言，系统中的线程总是非常有限的资源。为每个或每一批任务创建一个线程以对其进行执行，通常是一件奢侈而不能实现的事情。比较常见的一种方法就是去复用一定数量的线程，由这些线程去执行不断产生的任务。线程池(Thread Pool)本质就是一个复用极其有限的线程处理相对无限任务的并行计算框架[51]。

Thread Pool 包含了若干准备运行的空闲线程，线程在程序运行的开始创建，使用队列对待处理的任务进行缓存，并复用一定数量的工作线程（线程池中的线

程）去取队列中的任务进行执行，执行完成后，如果没有其他任务，线程转入休眠状态，等有任务的时候再唤醒，直到所有任务都执行完毕再关闭线程池。线程池的原理如图 2.2 所示，其中，阻塞队列里的任务是对所有线程共享。

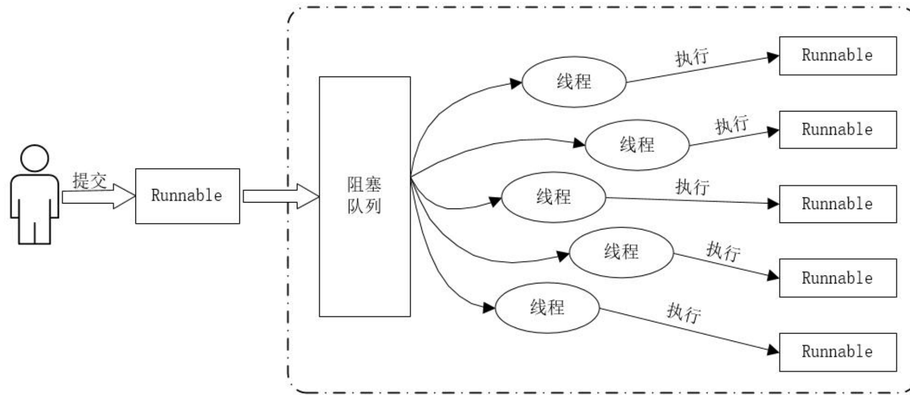


图 2.2 线程池原理图

图 2.2 中的 Runnable 代表可执行状态。线程的运行与否和进程一样都要听命于 CPU 的调度，把这种中间状态称为可执行状态，也就是说它具备执行的资格，但是并没有真正的执行起来而是在等待 CPU 调度。

线程池大小是指线程池中工作线程的数量，对线程池的工作效率有很大影响。线程池大小太大会消耗过多资源，并增加上下文切换。线程池大小太小，又可能导致无法充分利用 CPU 资源，使任务处理的吞吐率过低。合理的线程池大小取决于该线程池所要处理的任务的特性、系统资源状况以及任务所使用的稀缺资源状况等因素。

线程池的调度逻辑可以总结为如图 2.3 所示。

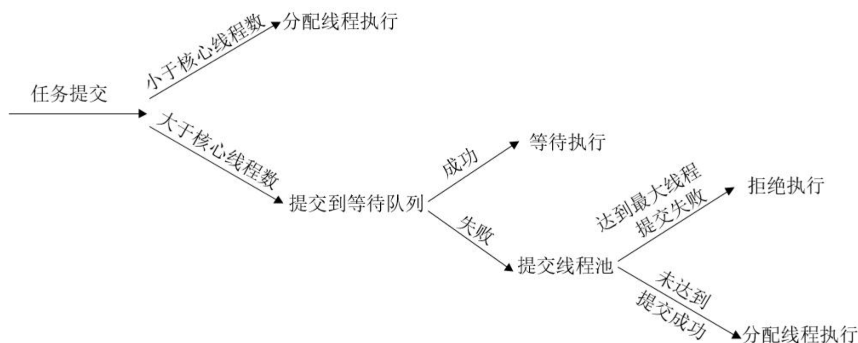


图 2.3 ThreadPoolExecutor 的任务逻辑调度图

类 ThreadPoolExecutor 可以被用来创建线程池，是 Thread Pool 模式的一个可复用实现。在创建了 ThreadPoolExecutor 对象后，可以将 Runnable 对象交给线程池运行。

Thread Pool 模式通过复用一定数量的工作线程去执行不断被提交的任务，节约了线程这种有限而昂贵的资源。Thread Pool 模式还可以带来以下好处：

（1）抵消线程创建的开销，提高响应性。Thread Pool 模式通过事先创建一部分线程使得被提交的任务可以立即被执行而无需等待工作线程的创建，从而提高了响应性。由于一个工作线程可以为多个任务服务，因此创建工作线程的开销被平摊到其执行的所有任务中。一个工作线程执行的任务越多，那么其创建的开销就越明显地被抵消。

（2）提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性。因此，需要线程池来管理线程。

（3）减少销毁线程的开销。Thread Pool 模式是我们避免了频繁的创建工作线程，因此避免了频繁的销毁已停止的线程，从而减少相应的开销。

2.3.2.2 Fork/Join

从 JDK1.7 版本开始，Java 引入了 Fork/Join 框架编程模式用于并行执行任务，该框架是适用于多核处理器上并行编程的轻量级并行框架，可以满足多核时代并发/并行编程的要求，从而更好地提供程序的性能。

Fork/Join 采用了分而治之的思想，通常用于解决那些可以递归地分解为更小任务的问题。Fork/Join 框架编程模式主要由 Fork 和 Join 两个操作构成，Fork 操作主要用于对任务和数据进行划分，是将一个大任务不断分割，直到子任务达到某个条件（如小于某个值）即可执行子任务；Join 操作主要用于对各个子任务的运行结果进行合并，得出大任务的结果。Fork/Join 框架类似于 MapReduce, 可以说是一种设计模式，一种思想，在其他编程语言中也同样适用。

Fork/Join 框架的原理如图 2.4 所示。

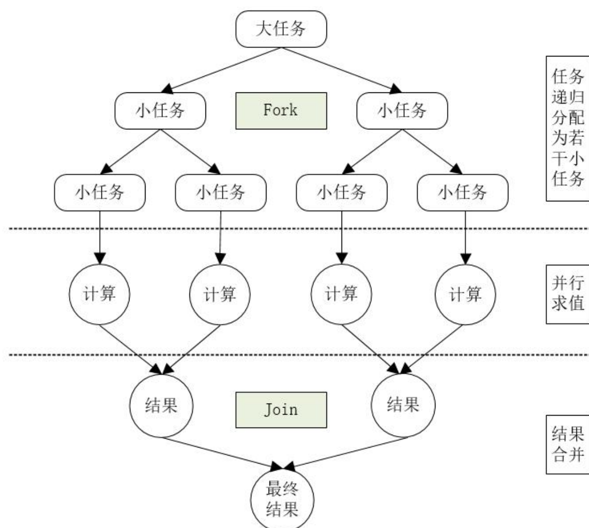


图 2.4 Fork/Join 原理图

Fork/Join 框架强调任务的划分。如果划分任务过多，导致子任务太小，任务的划分和合并操作可能给程序带来额外的开销；相反，如果子任务划分过大，解决起来可能不够简便。在使用 Fork/Join 框架的过程中，首先对任务 ForkJoinTask 进行划分，任务划分的数目可以根据问题的特征以及 CPU 可同时处理的线程数进行设定，然后将划分的任务交给 ForkJoinPool 处理，最后对处理结果进行收集。

Fork/Join 框架的优点是工作窃取算法。工作窃取算法是指当前程序中提前完成任务的线程会去查看是否还有未处理完成的工作，如果有，则在未完成任务的线程的队列中窃取任务来执行，从而帮助未完成的线程尽快完成工作，加快整体任务的完成。工作窃取算法是 Fork/Join 框架的核心，是提高程序性能、保证负载均衡的一种方法。

Fork/Join 框架采用双端队列(Deque)作为任务的存储结构，该队列支持后进先出(Last In First Out,LIFO)的数据 pop 和 push 操作，并且支持先进先出(First In First Out,FIFO)的 take 操作。由某一工作线程创建的子任务仍然会加入到自己的队列中，一般采用 push 操作；工作线程从自己的双端队列中取出任务执行，一般采用 pop 操作；当工作线程需要从其他线程的工作队列中窃取任务执行时，一般采用 take 操作，如图 2.5 所示。

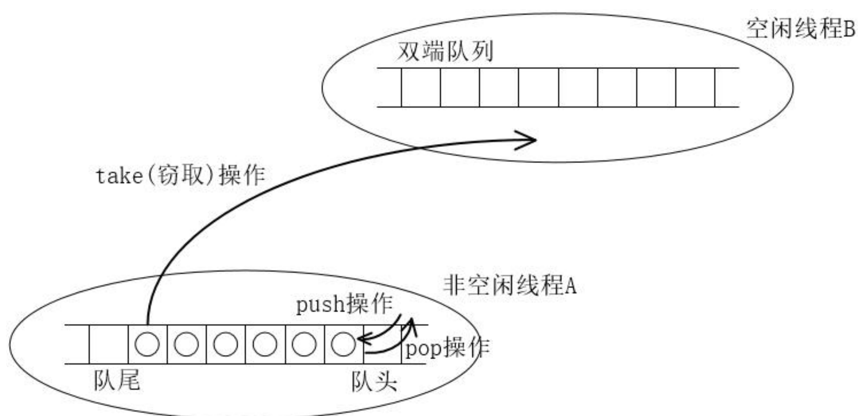


图 2.5 Fork/Join 框架的任务窃取原理图

Fork/Join 框架的任务窃取原理可以简单理解为：当使用 Fork 操作产生新任务时，会把新任务添加到队列的头部，这样可以保证 Fork 出来的新任务可以得到尽快执行；当某个工作线程执行完自己的任务时，就从其他工作线程队列的尾部窃取一个任务执行。

Fork/Join 框架实现了任务定义和任务处理功能的分离，是程序员在实现并行编程的同时，能够集中精力实现业务逻辑相关的内容极大程度上降低了并行程序设计的难度。

2.3.2.3 Hadoop

Hadoop 是最常见的分布式计算框架。狭义上 Hadoop 指的是 Apache 软件基金会的一款开源软件，允许用户在不了解分布式底层细节的情况下，使用简单的编程模型实现跨机器集群对海量数据进行分布式计算处理。

Hadoop 有许多元素组成，其中最核心的两个部分是：（1）Hadoop HDFS[52]（分布式文件存储系统）：处于 Hadoop 最底部，存储 Hadoop 集群中所有存储节点上的文件；（2）Hadoop MapReduce（分布式计算框架）：处于 HDFS 的上一层，是一种用于处理大数据的编程模型。它将数据集切分为多个小任务，每一个小任务都可以通过独立的计算来完成，最终的结果可以根据合并或更新数据来进行聚合。从 Hadoop 的核心部分对 Hadoop 进行如下介绍。

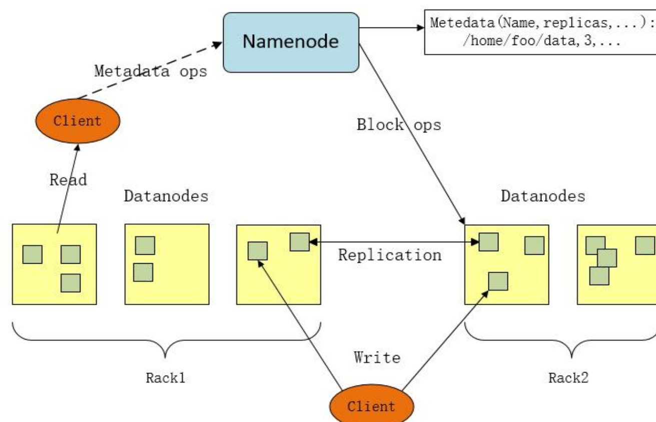


图 2.6 HDFS 架构图

结合图 2.6 所示的 HDFS 架构图，对 HDFS 架构进行如下介绍：

HDFS 采用 Master/Slave 的架构来存储数据，这种架构主要由 Client、Namenode、Datanode 和 Secondary Namenode 这四个部分组成。

（1）Client。Client 就是客户端，文件上传 HDFS 的时候，Client 将文件切分成一个个 Block，然后进行存储。Client 与 Datanode 交互，读取或写入数据；Client 与 Namenode 交互，获取文件的位置信息。Client 可以通过指令管理和访问 HDFS。

（2）Namenode。Hadoop 集群包含一个 Namenode 和大量的 Datanode，Namenode 就是 Master/Slave 架构的 Master，它是管理者，负责管理文件系统名称空间和控制外部客户机的访问。实际上 I/O 事务并没有经过 Namenode，只有表示 Datanode 和块的文件映射的元数据经过 Namenode。

（3）Datanode。Namenode 就是 Master/Slave 架构的 Slave，Namenode 下达命令，Datanode 执行实际的操作存储实际的数据块，执行数据的读/写操作。

（4）Secondary Namenode。Secondary Namenode 是 Namenode 的一个工具，帮助 Namenode 管理元数据信息。Namenode 本身不可避免地具有 SPOF（Single Point Of Failure）单点失效的风险，当 Namenode 出现故障时，Secondary Namenode 辅助恢复 Namenode，并不是马上替换 Namenode 提供服务。

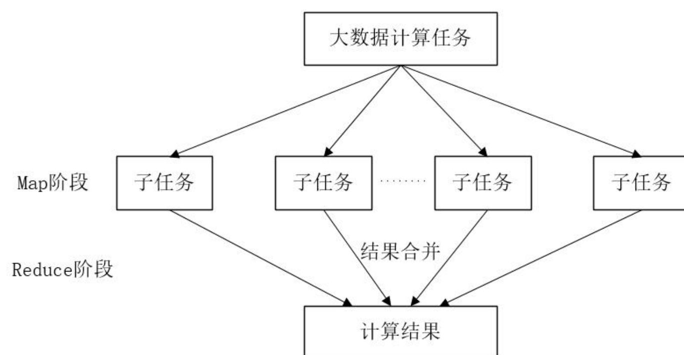


图 2.7 MapReduce 原理图

图 2.7 所示的是 MapReduce 原理图，最简单的 MapReduce 包含 3 个部分，map 函数、reduce 函数和 main 函数。其中 map 函数和 reduce 函数是 MapReduce 的根源，main 函数将作业控制和文件输入/输出结合起来。MapReduce 本身就是用来并行处理大数据集的框架。Map 函数可以将数据的输入域中的每个元素转化为对应的<key,value>列表，Reduce 函数将多个 Map 函数生成的列表进行合并，为每个 key 合并成一个<key,value>数据。MapReduce 可以多个任务同时进行，快速的合并和更新计算结果，极大地简化了处理大数据的过程。

Hadoop 的处理部分 MapReduce 也具有相对的局限性，主要表现在以下几方面：

（1）不能进行复杂的计算。例如图形计算、迭代计算；

（2）实时计算性能差。MapReduce 最初被设计用于批处理数据，不擅长低延迟；

（3）不能进行流计算。流式计算的数据是源源不断产生的，是随时间延续而无限增长的动态数据集合。MapReduce 作为离线计算框架，不能进行流数据处理。

基于 HDFS 架构和 MapReduce 原理，可以总结 Hadoop 的特性优点为：（1）高扩展性：Hadoop 是在可用的计算机集群之间分配数据并完成计算任务的，这些集群可方便灵活的方式扩展到数以千计的节点；（2）成本低：Hadoop 集群允许通过部署普通廉价的机器组成集群来处理大数据，以至于成本很低，看中的是集群整体能力；（3）高效性：通过并发数据，Hadoop 可以在节点之间动态并行的移动数据，使得速度非常快；（4）可靠性：能自动维护数据的多份复制，并

且在任务失败后能自动的重新部署计算任务，所以 Hadoop 的按位存储和处理数据的能力值得依赖。

2.3.2.4 Spark

Spark[53]是基于内存的并行计算框架，与 Hadoop 类似，但是可以弥补 Hadoop 处理部分 MapReduce 的不足。经过多年的发展，Spark 目前已经成为集 Spark Streaming、Graphx[54]、MLlib[55]、SparkSQL 等多功能为一体的生态系统。Spark 可以计算结构化、半结构化、非结构化等各种类型的数据结构，同时也支持 Python、Java、Scale、R 以及 SQL 语言去开发应用程序计算数据，应用面非常广泛，所以，被称之为统一的分析引擎。

Spark 提出了一种基于内存的弹性分布式数据集 RDD，RDD 是分布式内存的抽象概念，可以看作是 Spark 的一个对象，本身可以运行与内存中，如读文件是一个 RDD，结果集也是一个 RDD，不同的分片、数据之间的依赖、key-value 类型的 map 数据也是 RDD。通过对 RDD 的一系列操作完成计算任务，可以大大提升性能。

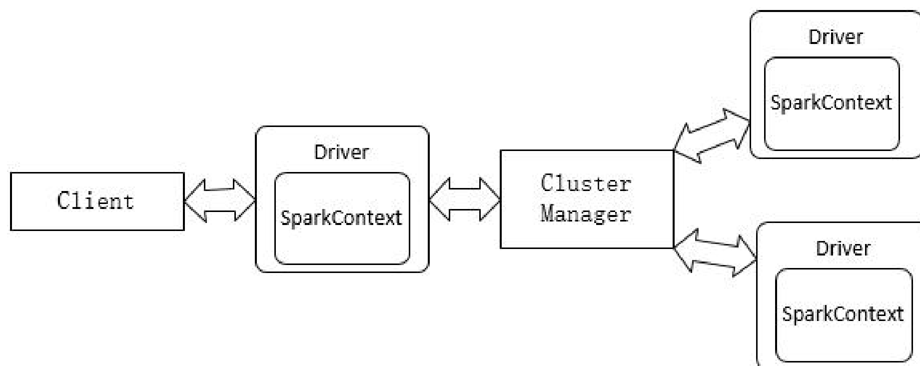


图 2.8 Spark 运行架构图

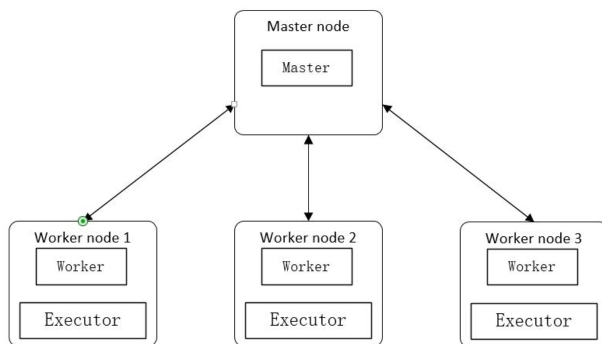


图 2.9 Spark 集群的基本结构

Spark 运行架构图如图 2.8 所示。Spark 集群的基本结构如图 2.9 所示。Spark 程序运行机制为：

（1）Client：用户的客户端，提交应用，Master 节点启动 Driver；

（2）Driver：负责控制一个应用的执行，Driver 向 Cluster Manager 申请资源，并构建 Application 的运行环境，即启动 SparkContext；

（3）SparkContext 向 Cluster Manager 申请 Executor 资源；

（4）Executor 向 SparkContext 申请 Task， SparkContext 将代码发放给 Executor，Executor 负责执行 Task 任务。

在 Spark 应用程序执行过程中，Driver 和 Worker 扮演着重要的角色。Driver 时应用执行起点，负责作业调度，Worker 管理计算节点及创建并行处理任务。

便于对 Spark 框架的进一步了解，对 Hadoop 和 Spark 进行了比较，如表 2.4 所示。

表 2.4 Hadoop 和 Spark 比较

	Hadoop	Spark
类型	基础平台，包含计算、存储、调度	纯计算工具
场景	海量数据批处理（磁盘迭代计算）	海量数据批处理（内存迭代计算、交互式计算）、海量数据流计算
价格	对机器要求低	对内存有要求
编程范式	Map+Reduce,API 较为底层，算法适应性差	RDD 组成有向无环图，API 较为顶层，方便使用

续表 2.4 Hadoop 和 Spark 比较

	Hadoop	Spark
数据存储结构	MapReduce 中间计算结果在 HDFS 磁盘上，延迟大	RDD 中间运行结果在内存中，延迟小
运行方式	Task 以进程方式维护，任务启动慢	Task 以线程方式维护，任务启动快，可批量创建提高并行能力

尽管 Spark 相对于 Hadoop 而言有巨大的优势，但 Spark 并不能完全替代 Hadoop，主要是因为：

（1）在计算层面，Spark 相较 MapReduce 有巨大的性能优势，但至今仍然有许多计算工具基于 MapReduce 框架；

（2）Spark 仅作计算，而 Hadoop 生态圈不仅有 MapReduce，还有 HDFS 和资源管理器 YARN(Yet Another Resource Negotiator)，HDFS 和 YARN 仍然是许多大数据体系的架构核心。

2.4 数据集介绍

本次实验使用的数据均采用数据挖掘专业网站 SPMF（见网址：SPMF: A Java Open-Source Data Mining Library (philippe-fournier-viger.com)）公开数据集中的标准数据集，现将数据集的特征介绍如下：

表 2.5 数据集特征

数据集	大小	事务数量	项数量	平均项数量	密度	特征
Retail	6.83MB	88162	16470	10.3	0.06%	极稀疏
Mushroom	1.03MB	8124	119	23	19.33%	极稠密
Connect	16.20MB	67557	129	43	33.33%	极稠密
eCommerce	1.92MB	14975	3468	11.71	0.34%	稀疏
Pumsb	26.42MB	49046	2113	74	3.50%	稠密

2.5 本章小结

本章介绍了与本文研究内容有关的理论知识和实验使用的数据集。相关理论

知识主要包括关联规则、频繁项集挖掘算法的相关定义和性质、高效用项集挖掘算法的相关定义和性质、并行计算概念及其框架，为本章之后的实验部分奠定了理论基础。

3 改进的高效用项集挖掘算法研究

在高效用项集挖掘算法中，已有多种挖掘技术和优化方法被提出且证实有效。但是不同的挖掘技术和优化方法具有不同的性质和特征，适用于不同的数据结构。目前被广泛使用的数据结构主要分成两种：树和 Utility-List。

HUI-Miner 算法和 ULB-Miner 算法都是基于 Utility-List 数据结构的高效用项集挖掘算法。其中，HUI-Miner 算法是第一个一阶段算法，首次提出 Utility-List 数据结构，被证实优于此前的所有二阶段算法[37]。使用 Utility-List 数据结构的算法不生成候选项集，不执行昂贵的数据库扫描，但是基于 Utility-List 数据结构的算法在创建和维护 Utility-List 时非常耗时，并且可能会消耗大量的内存。在 HUI-Miner 算法的基础上，先后提出了 FHM 算法和 ULB-Miner 算法。ULB-Miner 算法不仅在 HUI-Miner 算法的基础上进行改进，还融合了 FHM 算法中提出的 2-项集剪枝策略 EUCP，在稠密和稀疏数据集上都表现良好。

本章以 Utility-List 为研究中心，分别对 HUI-Miner 算法和 ULB-Miner 算法使用相同策略进行改进，目的是为了减少改进策略对单一算法适用的偶然性。通过投影数据库技术和事务合并策略，合并数据库中的相同事务，减少 Utility-List 的连接次数，并应用基于剩余效用的剪枝策略，减少生成的投影数据库的数量，分别提出了改进算法 HUIwTMS-Miner 和 ULBwTMS-Miner。

3.1 Utility-List 数据结构

在高效用项集挖掘领域中，Utility-List 数据结构被用来维护数据库的效用信息。在算法的挖掘过程中，首先根据数据库构建 1-项集的 Utility-List，之后不再对数据库进行扫描，然后根据 1-项集的 Utility-List 进行挖掘。通过比较两个项集的 Utility-List 中的 Tids 来识别公共事务, 取交集进行连接操作，生成多元 Utility-List。1-项集的 Utility-List 数据结构如图 3.1 所示。

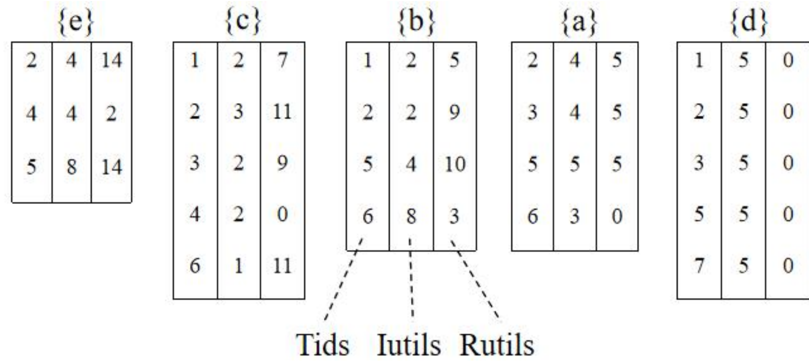


图 3.1 1−项集的 Utility-List

项集 X 的 Utility-List 中的每个元素都包括三个字段：Tid、Iutil 和 Rutil。其中，字段 Tid 表示包含项集 X 的事务 T；字段 Iutil 表示项集 X 在 T 中的效用，即 u(X, T)；字段 Rutil 表示包含项集 X 在 T 中的剩余效用，即 RU(X, T)。

(k+1)−项集可通过两个具有共同前缀的 k−项集取交集生成。构建(k+1)−项集的 Utility-List，可分为两种情况：

（1）共同前缀为空时，表现为两个 1-项集取交集生成 2-项集。图 3.2 所示的是前缀为空的 2-项集的 Utility-List 计算方法。

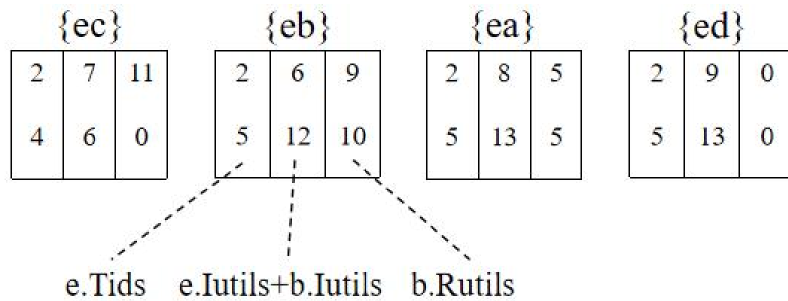


图 3.2 2−项集的 Utility-List

（2）共同前缀不为空时，表现为两个 k-项集(k≥2)取交集生成(k+1)-项集。图 3.3 所示的是以{e}为前缀的 3-项集的 Utility-List 计算方法。

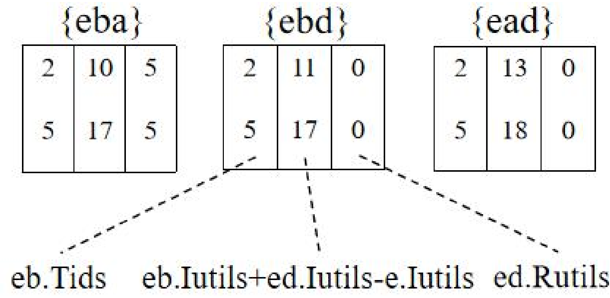


图 3.3 3-项集的 Utility-List

3.2 HUI-Miner 和 ULB-Miner 算法介绍

为了便于对本文中的算法进行详细说明和表述，给出原始数据库 D，如表 3.1 所示。数据库由 7 条事务组成，Transaction 中给出了项在事务中的效用（表 3.1 中给出的效用是内部效用和外部效用的乘积）。设置最小效用阈值 min_util=30。

表 3.1 原始数据库

Tid	Transaction	TU
T_1	b(2) c(2) d(5) g(1)	10
T_2	a(4) b(2) c(3) d(5) e(4)	18
T_3	a(4) c(2) d(5)	11
T_4	c(2) e(4) f(3)	9
T_5	a(5) b(4) d(5) e(8)	22
T_6	a(3) b(8) c(1) f(6)	18
T_7	d(5) g(5)	10

3.2.1 HUI-Miner 算法

HUI-Miner 算法的主要过程展示为 Algorithm 3.1。其中，输入 D 为事务数据库，min_util 为最小效用阈值，输出为高效用项集。

```
Algorithm 3.1 HUI-Miner algorithm
Input:
      D:a transaction database;    //事务数据库
      min_util:a minimum utility threshold;    //最小效用阈值
Output:
      The set of high utility itemsets ;    //高效用项集的集合
1  Scan D to calculate the TWU of 1-item;    //第一次扫描数据库计算 1-项集的 TWU 值
2  Let I* be the list of 1-item such that TWU ≥ min_util;
3  Let ≻ be the total order of TWU ascending values on I*;
4  Scan D again to rebuild the database and build the utility-list of each item i∈I* ;    //第二次扫描数
   据库，根据 I*重构数据库，并构建每个 i∈I*的效用列表
5  Search(∅,I*,min_util);
```

```
Algorithm 3.2 Search Procedure
Input:
      P.UL:the utility-list of itemset P;    //项集 P 的效用列表
      ULs:the set of utility-list of all P's 1-extensions;    //所有 P 的 1-项扩展项集的效用列表集合
      min_util:a minimum utility threshold;    //最小效用阈值
Output:
      all the high utility itemsets with P as prefix;    //前缀为 P 的所有高效用项集
1  foreach utility-list X in ULs do
2      if (X.SumIutils≥min_util) then
3          output the extension associated with X;
4      end
5      if (X.SumIutils+X.SumRutils≥min_util) then
6          exULs=NULL;    //以 Px 作为前缀的 1-项扩展项的效用列表集合
7          foreach utility-list Y after X in ULs do
8              exULs=exULs+Construct(P.UL,X,Y);    //调用 Construct，并更新 exULs
9          end
10         Search(X,exULs,min_util);    //递归调用 Search
11     end
12 end
```

```
Algorithm 3.3 Construct algorithm
Input:
      P.UL:the utility-list of itemset P;    //项集 P 的效用列表
      Px.UL:the utility-list of itemset Px;    //项集 Px 的效用列表
      Py.UL:the utility-list of itemset Py;    //项集 Py 的效用列表
Output:
      Pxy.UL:the utility-list of itemset Pxy;    //项集 Pxy 的效用列表
1  Pxy.UL=NULL;
2  foreach element Ex∈Px.UL do
3      if (∃Ey∈Py.UL and Ex.Tid==Ey.Tid) then    //如果 Px 和 Py 存在相同 Tid
```

```
4            if (P.UL is not empty) then
5                search such element E∈P.UL that E.Tid==Ex.Tid;
6                Exy=<Ex.Tid, Ex.Iutil+Ey.Iutil-E.Iutil, Ey.Rutil>;
7            else
8                Exy=<Ex.Tid, Ex.Iutil+Ey.Iutil, Ey.Rutil>;
9            end
10           append Exy to Pxy.UL;
11       end
12 end
13 return Pxy.UL;
```

Algorithm 3.1 的主过程执行以下步骤：

Step 1:第一次扫描数据库来计算所有 1-项集的事务加权效用 TWU 值，计算结果如表 3.2 所示。创建集合 I*存储 TWU≥min_util 的 1-项集，并按照 TWU 值升序排序（总顺序）。

表 3.2 各项的 TWU 值

Item	a	b	c	d	e	f	g
TWU	69	68	66	71	49	27	10

Step 2:第二次扫描数据库，根据总顺序重构数据库，同时构建 1-项集的 Utility-List。重新构建的数据库如表 3.3 所示，1-项集的 Utility-List 如图 3.1 所示。

表 3.3 重构数据库

Tid	Transaction	TU
T_1	c(2) b(2) d(5)	9
T_2	e(4) c(3) b(2) a(4) d(5)	18
T_3	c(2) a(4) d(5)	11
T_4	e(4) c(2)	6
T_5	e(8) b(4) a(5) d(5)	22
T_6	c(1) b(8) a(3)	12
T_7	d(5)	5

Step 3：调用 Search 过程，Search 过程的伪代码见 Algorithm 3.2。

根据生成的 Utility-List 进行计算，通过比较两个项集的 Utility-List 中的 Tids 来识别公共事务, 取交集生成新的多元 Utility-List，构造方法见 Algorithm 3.3。

对于取交集生成的 k-项集进行挖掘。算法挖掘策略为：

（1）若该项集对应 Utility-List 所有 Iutils 的总和大于等于 min_util，则该项集为高效用项集，输出。

（2） 若该项集对应 Utility-List 的所有 Iutils 与 Rutils 的总和大于等于 min_util，则该项集需要进一步判定。

（3）若该项集对应 Utility-List 的所有 Iutils 与 Rutils 的总和小于 min_util，则该项集及其所有扩展都为低效用项集，对其进行剪枝。

Step 4：直到不能取交集生成任何项集，则算法结束。

HUI-Miner 算法的搜索空间可以表示为一个集合枚举树。根据表 3.2 中项的 TWU 值进行 TWU≥min_util 筛选，并按照 TWU 值升序排序的顺序为：e、c、b、a、d。表 3.1 原始数据库的集合枚举树如图 3.4 所示。构造方法和过程如下：

（1）创建树的根节点∅；

（2）创建 n 个 1-项集作为根的 n 个子节点，从左到右依次按照 TWU 值升序排序：e、c、b、a、d；

（3）在每个节点末尾依次添加 TWU 值大于该节点中最后一项 TWU 值的项，作为该节点的子节点。可以简单地认为，把后面的节点单个的添加到前面的节点中，因为节点都是按照 TWU 值升序排序的：ec、eb、ea、ed、cb...;

（4）重复执行（3），直到创建所有叶子节点。

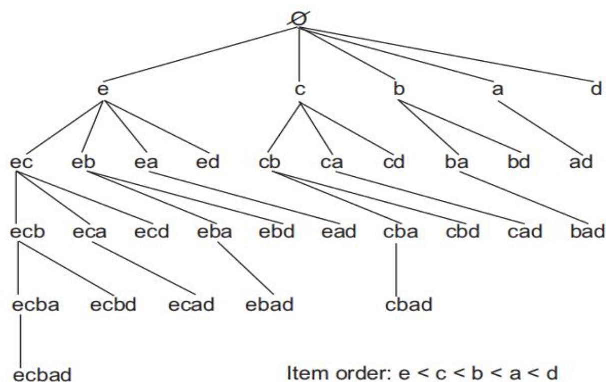


图 3.4 集合枚举树

3.2.2 ULB-Miner 算法

ULB-Miner 算法是在 HUI-Miner 算法的基础上进行改进，主要改变有两点：

（1）提出效用列表缓冲区结构(Utility-List buffer structure)来解决基于 Utility-List 的一阶段算法维护和更新 Utility-List 成本较高的局限；

（2）一种有效的连接操作，旨在在线性时间内在 Utility-List buffer 中创建效用列表段，以减少构建 Utility-List 结构所需的时间。

定义 11 效用列表缓冲区结构。数据库 D 的效用列表缓冲区结构记为 UTLBuf。该结构被设计为一个内存管道，用来存储 Utility-List 中项集的信息。数据库的效用列表缓冲区存储一组（$\mathrm{tid} \in \mathrm{Tid}_D, \mathrm{iutil} \in \mathrm{R}, \mathrm{rutil} \in \mathrm{R}$）形式的元组，这些元组称为数据段。为了快速访问存储在 Utility-List buffer 中的信息，将创建一组索引段，其中索引段 SUL(X)表示项集 X 的信息存储在 Utility-List buffer 中的位置。索引段允许快速访问存储在效用列表缓冲区的数据。

定义 12 索引段，记为 SUL。SUL(X)是数据库中项集 X 的索引段，被定义为具有形式(X,StartPos,EndPos,SumIutil,SumRutil)的元组。其中，StartPos 和 EndPos 分别代表数据段的开始位置和结束位置，SumIutil 和 SumRutil 分别代表项集 X 的效用列表的项效用总和和剩余效用总和。

定义 13 索引段列表。它是一个表示为 SULs_D 的内存通道，并被定义为 $SULs_D = \{\mathrm{SUL(X)}, \mathrm{X} \subseteq \mathrm{I}\}$。

在算法的实现过程中，提出了一种自适应的效用列表段构造过程，这个过程在 Utility-List buffer 的下一个空闲数据段构造一个 Utility-List 并更新 SULs 结构，以便在需要时从缓冲区中快速检索 Utility-List。将自适应的效用列表段构造过程的伪代码见 Algorithm 3.4。

Algorithm 3.4 basic utility-list segment construction procedure

```
Input:
        PPosition:the StartPos of itemset P;      //项集 P 的开始位置
        PxPosition:the StartPos of itemset Px;      //项集 Px 的开始位置
        PyPosition:the StartPos of itemset Py;      //项集 Py 的开始位置
Output:
        PxyPosition:the StartPos of itemset Pxy;      //项集 Pxy 的开始位置
1   PxyPosition=NULL;
```

```
2  countX=SULs(Px).EndPos-SULs(Px).StartPos;    //计算 X 的支持度计数，即不同的 Tid 数
3  for(i=0;i＜countX，i++)do
4      if(∃j ∈[SULs(Py).StartPos,SULs(Py).EndPos] and TIDs[SULs(Px).StartPos+1]=TIDs[j])then
5          if(PPosition≥0)then      //前缀 P 不为空
6                          Search  index  p  ∈  [SULs(P).StartPos,SULs(P).EndPos]  such  that
TIDs[p]=TIDs[SULs(Px).StartPos+1];
7              TIDs[PxyPosition+p]=TIDs[PxPosition+i];
8              Iutils[PxyPosition+p]= Iutils[PxPosition+i] +Iutils[PyPosition+j]- Iutils[PPosition+p];
9              Rutils[PxyPosition+p]= Rutils[PyPosition+j];
10         else
11             TIDs[PxyPosition+p]=TIDs[PxPosition+i];
12             Iutils[PxyPosition+p]= Iutils[PxPosition+i] +Iutils[PyPosition+j];
13             Rutils[PxyPosition+p]= Rutils[PyPosition+j];
14         end if
15     end if
16  end for
17  Update SULs of Pxy;
18  return PxyPosition;
```

ULB-Miner 算法主要过程展示为 Algorithm 3.5[42]。其中，输入 D 为事务数据库，min_util 为最小效用阈值，输出为高效用项集。

Algorithm 3.5 ULB-Miner algorithm

```
Input：
        D:a transaction database;
        min_util:a minimum utility threshold;
Output：
        The set of high utility itemsets ;
1  Scan D to calculate the TWU of 1-item;
2  Let I* be the list of 1-item such that TWU ≥ min_util;
3  Let ≻ be the total order of TWU ascending values on I*;
4   Scan D again to build the initial utility-list buffer UTLBuf,and SULs of item i∈I* and bulid the
EUCS;    //第二次扫描数据库，构建初始化 UTLBuf，SULs 和 EUCS
5  Search(∅,I*,min_util,EUCS,UTLBuf, SULs);
```

Algorithm 3.6 Search Procedure

```
Input：
        P:an itemset;   //项集 P
        ExtensionsOfP:a set of extensions of P ;    // P 的扩展集合
        min_util:a minimum utility threshold;
        EUCS:the EUCS structure;    //估计效用共现结构 EUCS
```

```
        Utility-list buffer UTLBuf:the utility-list buffer structure;    //效用列表缓冲区结构 UTLBuf
        SULs:the summary list;    //索引段列表 SULs
Output:
        The set of high utility itemsets ;
1   for each itemset Px∈ExtensionsOfP do
2       if (SULs(Px).SumIutil≥min_util) then
3           output Px;
4       end if;
5       if (SULs(Px).SumIutil+SULs(Px).SumRutil≥min_util) then
6           ExtensionsOfP← ∅;
7           for each itemset Py∈ExtensionsOfP such that y≻x do    // y≻x 代表项 y 排序在项 x 后
8               if (∃(x, y, c) ∈EUCS such that c≥min_util) then
9                   Pxy←Px∪Py;
10                  if (ULBReusingMemory-Construct(UTLBuf,SULs,P,Px,Py)) then
11                      ExtensionsOfPx←ExtensionsOfPx∪Pxy;
12                  end if
13              end if
14          end for
15          Search(Px,ExtensionsOfPx,min_util,EUCS,UTLBuf, SULs);    //递归调用 Search
16      end if
17  end for
```

Algorithm 3.7 ULBReusingMemory-Construct

```
Input:
        P:an itemset;    //项集 P
        Px:an itemset;    //项集 Px
        Py:an itemset;    //项集 Py
        Utility-list buffer UTLBuf:the utility-list buffer structure;
        SULs:the summary list;
Output:
        Updated UTLBuf,SULs with itemset Pxy;    //根据项集 Pxy 更新 UTLBuf,SULs
1   Let PPnt,PxPnt,PxPnt are three pointers that initially point to UTLBuf at positions
SULs(P).StartPos, SULs(Px).StartPos, SULs(Py).StartPos, respectively.    // PPnt，PxPnt，PxPnt 均为
初始化指针
2   Let EACriterion=SULs[Px].SunIutil+SULs[Py].SunIutil+SULs[Px].SunRutil+SULs[Py].SunRutil
    //EA 剪枝的条件
3   Let insertionPosition=SULs.Last.EndPos;    //插入位置为 SULs 最后位置
4   while (PxPnt not reach SULs(Px).EndPos and PyPnt not reach SULs(Py).EndPos) do
5       if (TIDs[PxPnt]＜TIDs[PyPnt]) then
6           shift PxPnt to the right by 1;
7           Substract EACriterion by(Iutils[PxPnt]+Rutils[PxPnt])
8       else if (TIDs[posX]＞TIDs[posY]) then
9           shift PyPnt to the right by 1;
```

```
10          Substract EACriterion by(Iutils[PyPnt]+Rutils[PyPnt])
11      else
12          if (SULs[P] is not NULL) then
13              while (PPnt not reach SULs(P).EndPos and TIDs[PPnt]≠TIDs[PxPnt]) do
14                  shift PPnt to the right by 1;
15              end while
16          end if
17          if (insertionPosition≥UTLBuf.TIDs.size()) then
18              UTLBuf.TIDs[TIDs.count++]=TIDs[PxPnt];
19              UTLBuf.Iutils[Iutils.count++]=Iutils[PxPnt]+Iutils[PyPnt]-Iutils[PPnt];
20              UTLBuf.Rutils[Rutils.count++]=Rutils[PyPnt];
21          else
22              insertionPosition++;    //重用内存
23              UTLBuf.TIDs[insertionPosition]=TIDs[PxPnt];
24              UTLBuf.Iutils[insertionPosition]=Iutils[PxPnt]+Iutils[PyPnt]-Iutils[PPnt];
25              UTLBuf.Rutils[insertionPosition]=Rutils[PyPnt];
26          end if
27          shift both PxPnt and PyPnt to the right by 1;
28      end if
29 end while
30 if (EACriterion＜min_util) then
31      return false;
32 end if
33 Update SULs[Pxy];
34 return true;
```

Algorithm 3.5 主过程执行以下步骤：

Step 1：第一次扫描数据库来计算所有 1−项集的事务加权效用 TWU 值，计算结果如表 3.2 所示。创建集合 I*存储 TWU≥min_util 的 1-项集，并按照 TWU 值升序排序（总顺序）。

Step 2：第二次扫描数据库，根据总顺序重构数据库，同时构建 1-项集的 Utility-List buffer 和 EUCS 结构。重新构建的数据库如表 3.3 所示，1-项集的 Utility-List buffer 如图 3.5 所示，EUCS 结构如图 2.1 所示。

TIDs	2	4	5	1	2	3	4	6	1	2	5	6	2	3	5	6	1	2	3	5	7
Iutils	4	4	8	2	3	2	2	1	2	2	4	8	4	4	5	3	5	5	5	5	5
Rutils	14	2	14	7	11	9	0	11	5	9	10	3	5	5	5	0	0	0	0	0	0

SULs					
	Item=e	Item=c	Item=b	Item=a	Item=d
	StartPos=0	StartPos=3	StartPos=8	StartPos=12	StartPos=16
	EndPos=3	EndPos=8	EndPos=12	EndPos=16	EndPos=21
	SumIutils=16	SumIutils=10	SumIutils=16	SumIutils=16	SumIutils=25
	SumRutils=30	SumRutils=38	SumRutils=27	SumRutils=15	SumRutils=0

图 3.5 1-项集的 Utility-List buffer

Step 3:深度优先搜索，调用 Search 过程。

Search 过程见 Algorithm 3.6。Search 过程是 ULB-Miner 算法的挖掘方法，包含了算法的剪枝策略和 Utility-List buffer 的维护和更新。

在 Search 过程中，为了减少内存使用，在 Utility-List buffer 中实现内存重用，也就是说，如果 Pxy 不是搜索空间的候选项，那么其存储 Utility-List 的内存将被召回并重用，用于在递归调用 Search 过程中考虑的下一个潜在候选项集，只有在 Utility-List buffer 已满时，才会分配新的内存。包含内存重用的 ULB 构造过程伪代码见 Algorithm 3.7。

（1）对重构数据库进行深度优先挖掘时，首先对项{e}进行展开，依次生成 2-项集{eb}、{ea}、{ed}，2-项集{ec}在 EUCS 结构中的 TWU 值为 24＜30，不满足展开条件，在挖掘过程中被舍弃。随后对 2-项集{eb}进行展开，依次生成 3-项集{eba}、{ebd},再对 3-项集{eba}进行展开，生成 4-项集{ebad}。对于扩展项的效用信息将在 1-项集的 Utility-List buffer 后的空闲数据段构造 Utility-List 并更新 SULs 结构。对于扩展项的 Utility-List buffer 如图 3.6 所示。

TIDs	2	5	2	5	2	5	2	5	2	5	2	5
Iutils	6	12	8	13	9	13	10	17	11	17	15	22
Rutils	9	10	5	5	0	0	5	5	0	0	0	0

SULs						
	Item=eb	Item=ea	Item=ed	Item=eba	Item=ebd	Item=ebad
	StartPos=21	StartPos=23	StartPos=25	StartPos=27	StartPos=29	StartPos=31
	EndPos=23	EndPos=25	EndPos=27	EndPos=29	EndPos=31	EndPos=33
	SumIutils=18	SumIutils=21	SumIutils=22	SumIutils=27	SumIutils=28	SumIutils=37
	SumRutils=19	SumRutils=10	SumRutils=0	SumRutils=10	SumRutils=0	SumRutils=0

图 3.6 扩展项的 Utility-List buffer

根据图 3.6 可以看出，只有 4-项集{ebad}的 SumIutils≥30，输出为高效用项集。3-项集{ebd}不满足展开条件，不再进一步挖掘。

（2）2-项集{eb}的扩展项搜索完成后，对 2-项集{ea}进行展开，在此过程中重用内存，重用内存后的 Utility-List buffer 如图 3.7 所示。

TIDs	2	5	2	5	2	5	2	5	2	5	2	5
Iutils	6	12	8	13	9	13	13	18	11	17	15	22
Rutils	9	10	5	5	0	0	0	0	0	0	0	0

SULs						
	Item=eb	Item=ea	Item=ed	Item=ead	Item=ebd	Item=ebad
	StartPos=21	StartPos=23	StartPos=25	StartPos=27	StartPos=29	StartPos=31
	EndPos=23	EndPos=25	EndPos=27	EndPos=29	EndPos=31	EndPos=33
	SumIutils=18	SumIutils=21	SumIutils=22	SumIutils=31	SumIutils=28	SumIutils=37
	SumRutils=19	SumRutils=10	SumRutils=0	SumRutils=0	SumRutils=0	SumRutils=0

图 3.7 重用内存后的 Utility-List buffer

根据图 3.7 可以看出，3-项集{ead}的 SumIutils≥30，输出为高效用项集。

（3）2-项集{ea}的扩展项搜索完成后，对 2-项集{ed}进行展开,由于 2-项集{ed}无可扩展项，至此，项{e}已展开完成。随后对其他 1-项集依次进行深度优先搜索。

Step 4：直到无任何可扩展项生成，算法结束。

3.3 改进策略

虽然 HUI-Miner 算法和 ULB-Miner 算法挖掘高效用项集的效率已经很出众了，但是使用 Utility-List 求交集会进行大量的 Tid 识别，每生成一个新的 k-项集就会对两个(k-1)-项集进行 Tid 比较。Utility-List 中 Tid 的数量决定着算法的挖掘效率。如果可以将相同的事务进行合并，就可以大量减少 Utility-List 中 Tid 的数量，进一步提升挖掘效率。

3.3.1 投影数据库

投影数据库技术广泛应用于关联规则的并行计算中，将串行的高效用项集挖掘算法进行并行计算时，为确保并行的数据间相互独立，投影数据库技术是一个很好的选择。使用投影数据库技术可以将原始数据库划分为多个投影子数据库，以满足在不同的线程或处理器上进行独立计算。有关投影数据库的定义如下：

定义 14 E(α)。E(α)表示根据深度优先搜索可以用来扩展项集α的所有项的集合。

定义 15 投影数据库[40]。在深度优先搜索时考虑项集α时，当扫描数据库以计算α子树中项集的效用或其效用的上界时，所有项不属于 E(α)都可以被忽略，没有这些项的数据库称为投影数据库。

定义 16 α-T。设有项集α的事务 T 的投影表示为α-T，并定义α-T={i|i∊T∧i∊E(α)}。

定义 17 α-D。项集α对数据库 D 的投影表示为α-D，并定义α-D={α-D|T∊D∧ α-T≠∅ }。

在高效用项集挖掘中，使用投影数据库技术会将原始数据库根据项进行投影数据库划分，一个投影子数据库可以看作是以某项为根节点的子树。使用投影数据库通常会降低数据库的扫描成本，因为随着搜索更大的项集，项集的投影数据库会变得更小，事务长度也会变短，从而减少内存使用。

3.3.2 事务合并策略

挖掘高效用项集所使用的事务数据库由多条事务组成，它们存在 0 或多条相同事务。结合 Utility-List 数据结构的特性，通过合并事务数据库中的相同事务可以减小数据库的大小，从而进一步减小根据事务数据库生成的 Utility-List 的大小，降低 Utility-List 的挖掘成本，提高算法的挖掘效率。

现将事务合并策略相关内容介绍如下：

定义 18 相同事务。如果事务 T_a 与事务 T_b 具有相同的项，即 $T_a=T_b$。需要注意的是，在这个定义中，仅需要两个事务具有相同的项，不需要两个相同的事务具有相同的内部效用值。

定义 19 事务合并。通过一个新的事务 T_m 替换数据库 D 中相同的事务 T_{r1}，T_{r2}，...，T_{rm}，即 $T_m=T_{r1}=T_{r2}=...=T_{rm}$，其中每个项的效用为 $u(i,T_m)=\sum_{k=1}^{m}u(i,T_{rk})$。

定义 20 事务总顺序(Total order on transactions)[40]。$\succ_T$顺序被定义为事务向后读取时的字典顺序。如果有两个事务$T_a=\{i_1,i_2,...,i_m\}$和$T_b=\{j_1,j_2,...,j_k\}$，$\succ_T$顺序由四种情况定义：

（1）如果T_a和T_b相同，且T_b的 Tid 大于T_a的 Tid，则$T_b\succ_T T_a$。

（2）如果 k＞m 且对于任意整数 x(0≤x＜m)，都有$i_{m-x}=j_{k-x}$，则$T_b\succ_T T_a$。

（3）如果存在一个整数 x(0≤x＜min(m,k))和任意整数 y(x＜y＜min(m,k))，使得$j_{k-x}\succ i_{m-x}$，且$i_{m-y}=j_{k-y}$，则$T_b\succ_T T_a$。

（4）除去上述三种情况，均有$T_a\succ_T T_b$。

性质 2 假设有一个$\succ_T$排序的数据库 D 和一个项集α，相同的事务将连续出现在投影数据库α-D 中。

为了确保可以有效实现事务合并，使用反向读取数据的方法，事先根据总顺序对原始数据库中的每个事务进行排序，与算法执行的其他操作相比，这个成本通常可以忽略不计，因为它只执行一次。但是如果我们只在原始数据库上应用事务合并策略，那么这种优化效果有限，不会使算法的速度更快，需要结合投影数据库技术进行合并，因为投影数据库比原始数据库更小，事务更短，更容易有相同的事务。

3.4 改进算法介绍

本节将本章 3.2 中提到的事务合并策略和投影数据库技术分别加入到 HUI-Miner 算法和 ULB-Miner 算法中，并将基于剩余效用的剪枝策略应用到投影数据库的构建阶段，对项进行二次筛选，删除所有超集都不是高效用的项，将潜在的高效用项生成投影数据库，并对相同事务进行合并，以此来减少算法在挖掘过程中 Utility-List 取交集的计算量，提高算法挖掘效率。

在本节介绍的两个改进算法中，事务合并策略的具体应用主要分为两个步骤：

（1）对原始数据库根据$\succ_T$顺序对每条事务进行排序；

（2）在划分投影数据库时，与先前一条事务比较，如果事务相同，则将该事务与先前一条事务进行合并，如果事务不相同，则将该事务设置为先前一条事务，直到投影数据库都构建结束。

3.4.1 改进算法 HUIwTMS-Miner

3.4.1.1 HUIwTMS-Miner 算法描述

HUIwTMS-Miner 算法的主要过程展示为 Algorithm 3.8。其中，输入 D 为事务数据库，min_util 为最小效用阈值，输出为高效用项集。

Algorithm 3.8 HUIwTMS-Miner algorithm

Input：
　　D:a transaction database;
　　min_util:a minimum utility threshold;
Output：
　　The set of high utility itemsets ;
1 Scan D to calculate the TWU of 1-item;
2 Let I* be the list of 1-item such that TWU ≥ min_util;
3 Let ≻ be the total order of TWU ascending values on I*;
4 Scan D again to rebuild the database and build the utility-list of each item i∈I*; //第二次扫描数据库，重构数据库并构建初始 utility-list
5 Sort transactions in D according to$\succ_T$; //数据库排序
6 delete empty transactions; //删除空事务
7 **foreach** i in I* **do**
8 　**if** (i.SumIutils+i.SumRutils≥ min_util) **then**
9 　　Scan D to creat projected transaction i-D and use projected transaction merging;

```
            //构建投影数据库并将相同事务合并
10          rebuild the utility-list according i-D;
11      end if
12  call the preSearch procedure;
13 end
```

Algorithm 3.9 preSearch Procedure

```
Input:
      ULs:the set of utility-list of all 1-item;     //所有 1-项的效用列表集合
      min_util:a minimum utility threshold;
Output:
      newprefix:P.item;     //新前缀 P
1   utility-list P =ULs.get(0);
2   if (P.SumIutils≥min_util) then
3     output;
4   end
5   if (P.SumIutils+P.SumRutils≥min_util) then
6     exULs=NULL;
7     foreach utility-list X after P in ULs do
8       exULs=exULs+Construct(P.UL,X,Y);
9     end
10    Search(X,exULs,min_util);
11 end
```

Algorithm 3.8 的实现过程进行如下表述：

Step 1:第一次扫描数据库来计算所有 1−项集的事务加权效用 TWU 值，计算结果如表 3.2 所示。创建集合 I*存储 TWU≥min_util 的 1-项集，并按照 TWU 值升序排序（总顺序）。

Step 2:第二次扫描数据库，根据总顺序重构数据库，并对重构数据库中的事务排序。同时构建 1-项集的 Utility-List。重新构建的数据库如表 3.3 所示，1-项集的 Utility-List 如图 3.1 所示。

根据反向读取数据的方法，并按照定义 15 给出的事务总顺序对重构数据库中的事务进行排序，在此过程中将空事务删除。排序后的数据库如表 3.4 所示。

表 3.4 排序后的数据库

Tid	Transaction	TU
T_4	e(4) c(3)	6
T_6	c(1) b(8) a(3)	12
T_1	c(2) b(2) d(5)	9
T_3	c(2) a(4) d(5)	11
T_5	e(8) b(4) a(5) d(5)	22
T_2	e(4) c(3) b(2) a(4) d(5)	18

Step 3：构建投影数据库，在过程中对相同事务进行合并，并重构各项的Utility-List。

在构建投影数据库时，使用剪枝策略 2 对 1-项集进行二次剪枝。根据图 3.1 所示的 1-项集的 Utility-List，满足 sumIutils+sumRutils≥min_util 条件的项有{e、c、b、a}，分别构建它们的投影数据库，如表 3.5-3.8 所示。以 e-D 为例，生成 1-项集的 Utility-List 见图 3.8 所示。

表 3.5 e-D

Tid	Transaction	TU
T_4	e(4) c(3)	6
T_5	e(8) b(4) a(5) d(5)	22
T_2	e(4) c(3) b(2) a(4) d(5)	18

表 3.6 c-D

Tid	Transaction	TU
T_4	c(2)	2
T_6	c(1) b(8) a(3)	12
T_1	c(2) b(2) d(5)	9
T_3	c(2) a(4) d(5)	11
T_2	c(3) b(2) a(4) d(5)	14

表 3.7 b-D

Tid	Transaction	TU
T_6	b(8) a(3)	11
T_1	b(2) d(5)	7
T_M	b(6) a(9) d(10)	25

表 3.8 a-D

Tid	Transaction	TU
T_6	a(3)	3
T_N	a(13) d(15)	28

其中，表 3.7 中的 T_M 是在构建投影数据库时，将 T_5 和 T_2 的投影事务进行合并后赋予的新的 Tid, 表 3.8 中的 T_N 是在构建投影数据库时，将 T_3、T_5 和 T_2 的投影事务进行合并后赋予的新的 Tid。为了方便辨别，在本文中选取了字母作为下标进行标示。

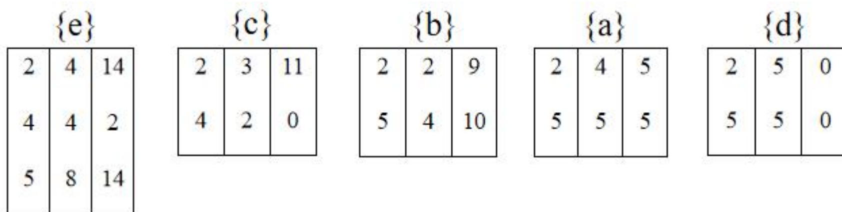


图 3.8 1-项集的 Utility-List

Step 4：预挖掘高效用项集, 见 Algorithm 3.9。

在 HUIwTMS-Miner 算法中为每个项都构建了投影数据库，分别对每个投影数据库进行挖掘。但是在投影数据库中会包含重叠数据，如果按照 HUI-Miner 算法中挖掘重构数据库的方法来挖掘投影数据库，会造成部分项集在多个投影数据库中的重复计算，而且这种计算不是建立在全局数据的基础上，没有科学的方法计算重复比例，导致挖掘结果混乱，没有可用性。

对于在投影数据库上挖掘高效用项集，首先要对 1−项集进行预挖掘。对 X-D

进行预挖掘时，只挖掘 X 及以 X 项为起始项的 2-项集，其中 2-项集的构造方法见 Algorithm 3.3。以 e-D 为例，生成 2-项集的 Utility-List 见图 3.9 所示。

{ec}

2	7	11
4	6	0

{eb}

2	6	9
5	12	10

{ea}

2	8	5
5	13	5

{ed}

2	9	0
5	13	0

图 3.9 2-项集的 Utility-List

Step 5：挖掘高效用项集。

对预挖掘生成的 2−项集进一步挖掘，挖掘方法见 Algorithm 3.2，构造方法见 Algorithm 3.3。

以 e-D 为例，对算法的挖掘过程进行展示：

（1）依次对满足 sumIutils+sumRutils≥min_util 条件的 2-项集进行扩展，生成的 3-项集 Utility-List 见图 3.3 所示。3-项集{ead}的 sumIutils≥min_util，则项集为高效用项集，输出。

（2）继续对满足 sumIutils+sumRutils≥min_util 条件的 3-项集继续扩展，生成 4-项集的 Utility-List 见图 3.10 所示。4-项集{ebad}的 sumIutils≥min_util，则项集为高效用项集，输出。

{ebad}

2	15	0
5	22	0

图 3.10 4-项集的 Utility-List

Step 6：对生成的每个投影数据库进行 Step 4 和 Step 5 的挖掘过程，直到所有的投影数据库都被挖掘，算法结束。

3.4.1.2 实验结果与分析

为了评估 HUIwTMS-Miner 算法的性能，将 HUIwTMS-Miner 算法和 HUI-Miner 算法在表 2.5 中提到的五个数据集上运行，分别在运行时间和运行内存方面进行比较。

（1）实验设置

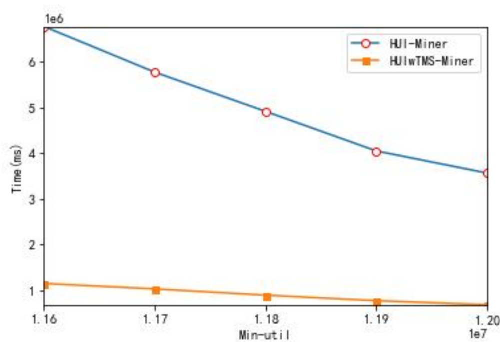
实验环境：（a）硬件：I7-12700KF，32GB 机带 RAM，固态 500G 硬盘，12 核心 20 线程的电脑；（b）软件：IDEA；（c）Windows 11 操作系统；（d）编程语言：Java；（e）Java 运行环境：JDK 21.0.3。

本文实验所用的算法均是精确挖掘算法，所以算法在同一数据集的同一 min_util 上挖掘的高效用项集数量是相同的。为了确保实验的的可信度，采用单一变量的方法进行对比实验，在每个数据集的不同最小效用阈值 min_util 下分别进行 5 次实验，记录实验结果并取平均值。

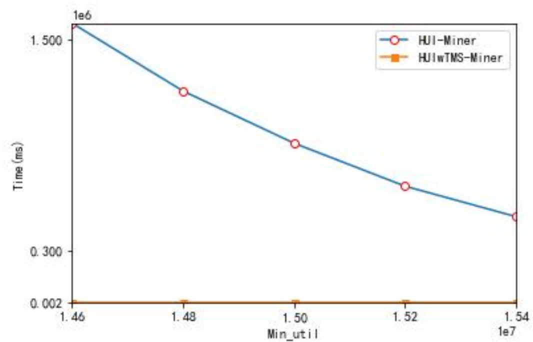
本文中所涉及到的所有实验均是在该实验设置下进行，在后面的实验中不再进一步介绍。

（2）运行时间的性能评估

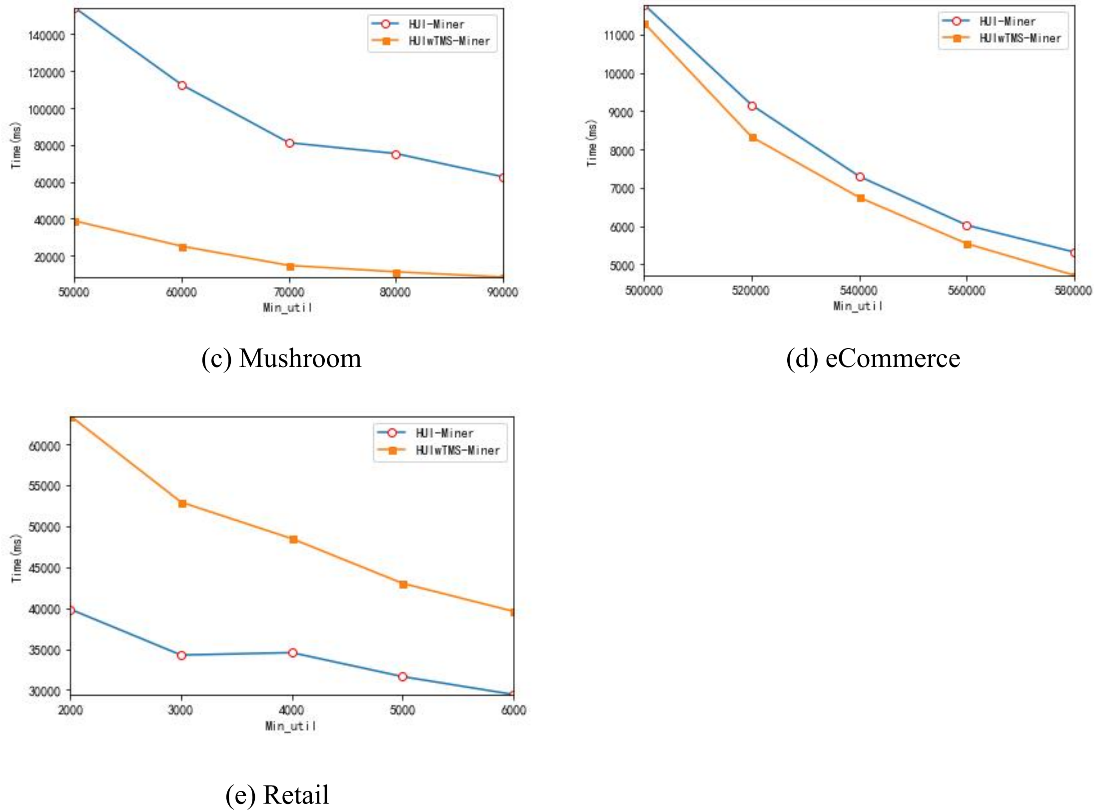
根据 HUI-Miner 算法和 HUIwTMS-Miner 算法在不同数据集上的运行时间绘制折线图，如图 3.11 所示。



(a)Pumsb



(b) Connect



(c) Mushroom (d) eCommerce

(e) Retail

图 3.11 HUI-Miner 算法和 HUIwTMS-Miner 算法的运行时间

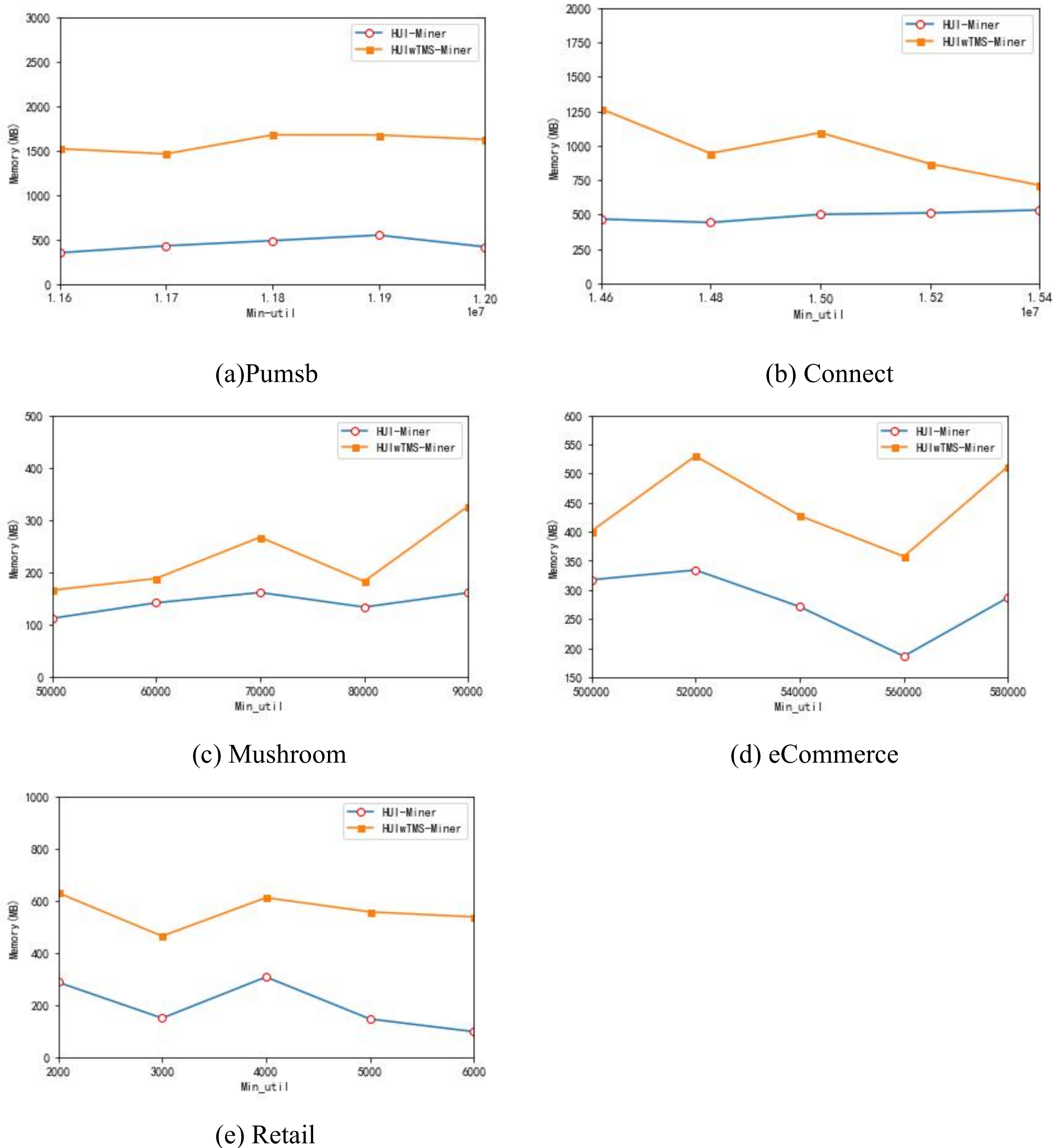
从图 3.11 可以很直观地看出，HUIwTMS-Miner 算法在 Pumsb、Connect、Mushroom 这三个数据集上的运行时间都有很大程度的缩减，在 eCommerce 数据集上有小程度缩减。其中时间缩减最明显的数据集是 Connect，结合表 2.5 给出的数据集特征可以看到，Connect 数据集包含 129 个项，事务平均长度为 43，密度(density)为 33.33%，属于极稠密数据集。在该数据集中，每个项出现的频率极高，出现相同事务的可能性较大，通过事务合并策略可以合并相当大部分的事务，极大程度上减少算法挖掘过程中的计算量。进一步计算 HUIwTMS-Miner 算法相对于 HUI-Miner 算法的运行时间变化幅度，将同一数据集上的不同最小效用阈值 min_util 的变化幅度取平均值进行表示，在 Pumsb、Connect、Mushroom 和 eCommerce 这四个数据集上的运行时间缩减幅度分别为：81.90%、99.76%、81.21%、8.03%。

但是，HUIwTMS-Miner 算法在 Retail 数据集上的运行时间并不乐观，不仅没有缩短时间，反而高出原算法 HUI-Miner 的运行时间。经过进一步计算，HUIwTMS-Miner 算法在 Retail 数据集上的运行时间的变化幅度是增长了 44.86%。

这是因为在密度仅为 0.06%的 Retail 数据集中，项出现的频率相对较少，出现相同事务的可能性较小，通过事务合并策略减少的运行时间不能够抵消事务排序合并、构建和挖掘投影数据库所增加的时间。

（3）运行内存的性能评估

根据 HUI-Miner 算法和 HUIwTMS-Miner 算法在不同数据集上的运行内存绘制折线图，如图 3.12 所示。



(a)Pumsb (b) Connect

(c) Mushroom (d) eCommerce

(e) Retail

图 3.12 HUI-Miner 算法和 HUIwTMS-Miner 算法的运行内存

从图 3.12 可以很直观地看到，在五个数据集上，HUIwTMS-Miner 算法运行时占用的内存都高于 HUI-Miner 算法的运行内存。这是因为在算法中将一个原始数据库划分为多个小的投影数据库，虽然每个投影数据库都小于原始数据库，但是项会在不同的投影数据库中重复出现，导致存储投影数据库占用过多的内存。进一步计算 HUIwTMS-Miner 算法相对于 HUI-Miner 算法的运行内存增加幅度，将同一数据集上的不同最小效用阈值 min_util 的变化幅度取平均值进行表示，那么在 Pumsb、Connect、Mushroom、eCommerce 和 Retail 数据集上的运行内存增加幅度分别为：262.32%、101.34%、57.32%、62.66%、231.12%。

（4）总结

综合考虑五个数据集的特征，结合运行时间和运行内存的性能评估可以得出结论：HUIwTMS-Miner 算法除了不适用于密度极低的数据集，也就是不适用于极稀疏的数据集，在其他数据集上，包括稀疏数据集和稠密数据集都能取得较好的结果，尤其是在稠密数据集上。HUIwTMS-Miner 算法在一定程度上增加内存占用的同时，可以大幅度减少运行时间。相对于内存的增加，运行速度的提升更加明显。根据实验分析得出结论，HUIwTMS-Miner 算法是有效且可行的。

3.4.2 改进算法 ULBwTMS-Miner

3.4.2.1 ULBwTMS-Miner 算法描述

ULBwTMS-Miner 算法主要过程的伪代码见 Algorithm 3.10。其中，输入 D 为事务数据库，min_util 为最小效用阈值，输出为高效用项集。

Algorithm 3.10 ULBwTMS-Miner algorithm

```
Input:
        D:a transaction database;
        min_util:a minimum utility threshold;
Output:
        The set of high utility itemsets ;
1   Scan D to calculate the TWU of 1-item;
2   Let I* be the list of 1-item such that TWU ≥ min_util;
3   Let ≻ be the total order of TWU ascending values on I*;
4   Scan D again to rebuild the database and build the utility-list of each item i∈I*;
```

```
5  Sort transactions in D according to≻T;
6  delete empty transactions;
7  foreach i in I* do
8      if (i.SumIutils+i.SumRutils≥ min_util) then
9          Scan D to creat projected transaction i-D and use projected transaction merging;
           //扫描数据库 D 构造投影数据库并进行投影事务合并
10         build the initial utility-list buffer UTLBuf, and SULs and build the EUCS according i-D;
           //根据投影数据库构建初始化 UTLBuf，SULs 和 EUCS
11     end if
12  call the Search procedure;
13 end
```

Algorithm 3.11 Search Procedure

```
Input:
        P:an itemset;
        ExtensionsOfP:a set of extensions of P ;
        min_util:a minimum utility threshold;
        EUCS:the EUCS structure;
        Utility-list buffer UTLBuf:the utility-list buffer structure;
        SULs:the summary list;
Output:
        The set of high utility itemsets ;
1  if (P=NULL) then
2      itemset Px=i;
3      if (SULs(Px).SumIutil≥min_util) then
4          output Px;
5      if (SULs(Px).SumIutil+SULs(Px).SumRutil≥min_util) then
6          ExtensionsOfP← ∅;
7          for each itemset Py∈ExtensionsOfP such that y≻x do
8              if (∃(x, y, c) ∈EUCS such that c≥min_util) then
9                  Pxy←Px∪Py;
10                 if (ULBReusingMemory-Construct(UTLBuf,SULs,P,Px,Py)) then
11                     ExtensionsOfPx←ExtensionsOfPx∪Pxy;
12                 end if
13             end if
14        end for
15        Search(Px,ExtensionsOfPx,min_util,EUCS,UTLBuf, SULs);
16     end if
17  else
18     for each itemset Px∈ExtensionsOfP do
19        if (SULs(Px).SumIutil≥min_util) then
20            output Px;
21        end if;
```

```
22          if (SULs(Px).SumIutil+SULs(Px).SumRutil≥min_util) then
23              ExtensionsOfP← ∅;
24              for each itemset Py∈ExtensionsOfP such that y≻x do
25                  if (∃(x, y, c) ∈EUCS such that c≥min_util) then
26                      Pxy←Px∪Py;
27                      if (ULBReusingMemory-Construct(UTLBuf,SULs,P,Px,Py)) then
28                          ExtensionsOfPx←ExtensionsOfPx∪Pxy;
29                      end if
30                  end if
31              end for
32              Search(Px,ExtensionsOfPx,min_util,EUCS,UTLBuf, SULs);
33          end if
34      end for
35  end
```

Algorithm 3.10 的实现过程进行如下表述：

Step 1:第一次扫描数据库来计算所有 1−项集的事务加权效用 TWU 值，计算结果如表 3.2 所示。创建集合 I*存储 TWU≥min_util 的 1-项集，并按照 TWU 值升序排序（总顺序）。

Step 2:第二次扫描数据库，根据总顺序重构数据库，并对重构数据库中的事务排序。同时构建 1-项集的 Utility-List。排序后的数据库如表 3.4 所示，1-项集的 Utility-List 如图 3.1 所示。

Step 3：使用剪枝策略 2 对 1-项集进行二次剪枝，对满足条件的项{e、c、b、a}分别构建投影数据库，在过程中对相同事务进行合并，如表 3.5-3.8 所示。根据投影数据库构建初始 Utility-List buffer 和 EUCS。

Step 4：调用 Search 过程挖掘高效用项集, 见 Algorithm 3.11。

由于在 ULBwTMS-Miner 算法中使用了投影数据库技术，在挖掘过程中，注意区分前缀 P 是否为空这两种情况。以挖掘 e-D 为例，通过初始 Utility-List buffer 构建项集 Pxy，构造方法见 Algorithm 3.7，过程分为两步：

（1）初始化前缀 P 为 NULL，ExtensionsOfP 为所有 1-项集，此时，为了满足只挖掘以{e}为根节点的子树中的所有节点，赋值 Px=e，Py 为所有排序在{e}后面的 1-项集，依次生成 2-项集{ec、eb、ea、ed}，以保证在递归调用 Search 过程的所有前缀 P 中都包含项{e}且以项{e}为起始项，并更新 ExtensionsOfP；

（2）前缀 P 不为 NULL，递归调用 Search 过程，项集 Px 可以赋值为

ExtensionsOfP 中的所有项。如果 Px 满足 SumIutil≥min_util 条件，则为高效用项集，输出；如果 Px 满足 SumIutil+SumRutil≥min_util 条件，进一步扩展，直到无法生成扩展项，该投影数据库挖掘结束。

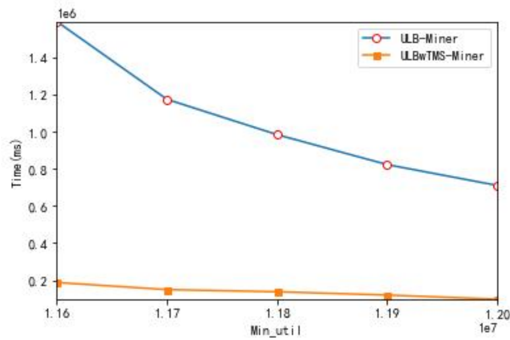
Step 5：直到所有的投影数据库都被挖掘，算法结束。

3.4.2.2 实验结果与分析

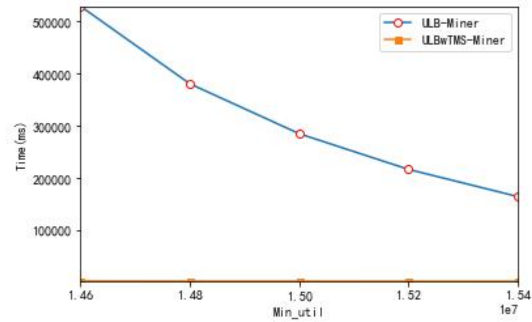
为了评估 ULBwTMS-Miner 算法的性能，分别将 ULBwTMS-Miner 算法和 ULB-Miner 算法在表 2.5 中提到的五个数据集上运行，分别在运行时间和运行内存方面进行比较。

（1）运行时间的性能评估

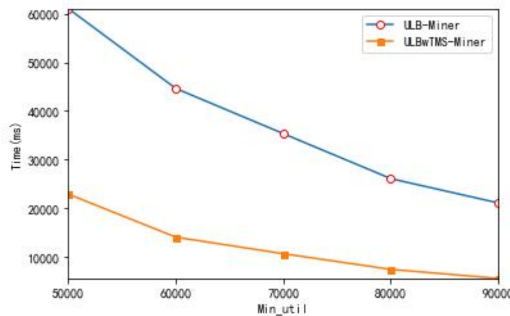
根据 ULB-Miner 算法和 ULBwTMS-Miner 算法在不同数据集上的运行时间绘制折线图，如图 3.13 所示。



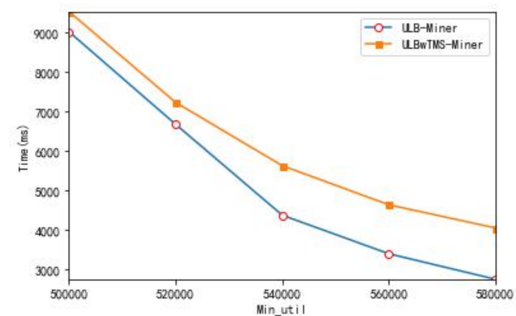
(a)Pumsb



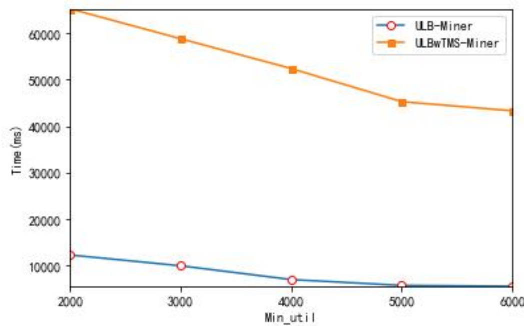
(b) Connect



(c) Mushroom



(d) eCommerce



(e) Retail

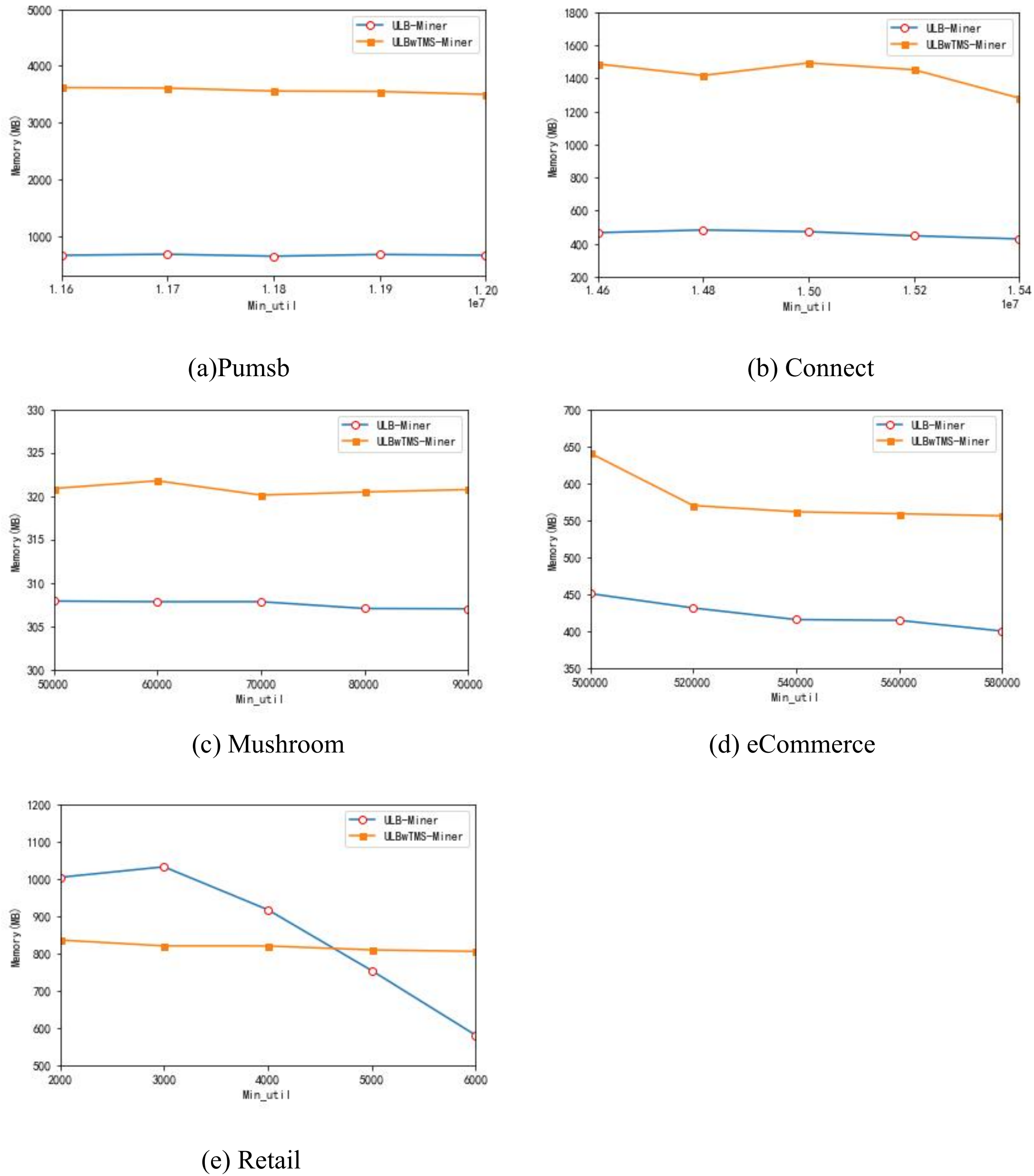
图 3.13 ULB-Miner 算法和 ULBwTMS-Miner 算法的运行时间

从图 3.13 可以很直观地看出，ULBwTMS-Miner 算法在 Pumsb、Connect、Mushroom 这三个数据集上的运行时间都较 ULB-Miner 算法有明显的缩减。进一步计算 ULBwTMS-Miner 算法相对于 ULB-Miner 算法的运行时间变化幅度，将同一数据集上的不同最小效用阈值 min_util 的变化幅度取平均值进行表示，在 Pumsb、Connect、Mushroom 这三个数据集上的运行时间缩减幅度分别为：86.71%、99.14%、69.34%。

但是，ULBwTMS-Miner 算法在稀疏数据集 eCommerce、Retail 上的运行时间高出 ULB-Miner 的运行时间，尤其是在密度仅为 0.06%的 Retail 数据集。一般情况下，项出现的频率和数据集的密度成正比。在稀疏数据集中，通过事务合并策略减少的运行时间不能够抵消事务排序合并、构建和挖掘投影数据库所增加的时间。经过进一步计算，ULBwTMS-Miner 算法在 eCommerce、Retail 数据集上的运行时间分别增长了 25.25%、591.70%。

（2）运行内存的性能评估

根据 ULB-Miner 算法和 ULBwTMS-Miner 算法在不同数据集上的运行内存绘制折线图，如图 3.14 所示。



(a)Pumsb (b) Connect

(c) Mushroom (d) eCommerce

(e) Retail

图 3.14 ULB-Miner 算法和 ULBwTMS-Miner 算法的运行内存

从图 3.14 可以很直观地看到，在 Pumsb、Connect、Mushroom、eCommerce 这四个数据集上，ULBwTMS-Miner 算法运行时占用的内存都明显高于 ULB-Miner 算法。进一步计算 ULBwTMS-Miner 算法相对于 ULB-Miner 算法运行内存的变化幅度，在 Pumsb、Connect、Mushroom 和 eCommerce 数据集上的运行内存增加幅度分别为：439.58%、210.21%、4.31%、36.61%。由于在 Retail 数据集上的特征不明显，通过计算，得出 ULBwTMS-Miner 算法的运行内存缩减幅度为 0.37%。

（3）总结

综合考虑五个数据集的特征，结合运行时间和运行内存的性能评估可以得出结论：ULBwTMS-Miner 算法不适用于稀疏数据集，在稠密数据集上表现良好，ULBwTMS-Miner 算法是有效且可行的。

3.5 本章小结

Utility-List 是高效用项集挖掘领域广泛使用的数据结构，本章旨在改进 Utility-List 的连接操作，即通过合并事务数据库中相同的事务来减小数据库，减小根据数据库生成的 Utility-List 的大小，以此减少在高效用项集挖掘算法的计算过程中 Utility-List 的连接成本。因此，本章对使用 Utility-List 数据结构的经典算法 HUI-Miner 和 ULB-Miner 进行改进，分别提出改进算法 HUIwTMS-Miner 和 ULBwTMS-Miner。为了发挥事务合并策略的最大作用，首先在全局数据库上进行事务排序，以确保在生成投影数据库时，相同事务总是出现在先后的两条事务；然后，继 TWU 剪枝后，使用剪枝策略 2 对 1-项集进行二次筛选，不再对低效用项生成投影数据库，减少生成投影数据库的数量；最后，在构建投影数据库时进行事务合并。

本章还针对新提出的两个算法设计了两组对比实验：（1）对比分析 HUI-Miner 算法和 HUIwTMS-Miner 算法的性能，证明了 HUIwTMS-Miner 算法的可行性和有效性；（2）对比分析 ULB-Miner 算法和 ULBwTMS-Miner 算法的性能，证明了 ULBwTMS-Miner 算法的可行性和有效性。

4 并行 ULBwTMS-Miner 算法研究

在当前大数据环境下，传统的串行高效用项集挖掘算法已难以适应数据规模的巨量化增长，采用并行计算技术的高效用项集挖掘算法的研究工作越来越多。并行高效用项集挖掘算法基于并行计算的思想，将大规模的数据分成若干个部分并行处理，提高算法的挖掘效率，满足挖掘需求。

多核处理器信息处理是一种帮助充分利用现代多核处理器的技术，这种处理器以合理的成本广泛使用[56]-[57]。在多核处理器环境下进行并行计算能够充分利用计算机资源，减少线程空闲带来的资源浪费。目前 PC 计算机大多都至少有 4、8 或者更多的物理核，为并行计算提供了便利条件。虽然在高效用项集挖掘领域多使用基于 Spark、Hadoop 的多机集群并行，但是多核处理器本身是基础，从微观的角度进行研究，有助于理解在 Spark、Hadoop 的多机集群系统上的运行原理。

第三章提出的两个改进算法都是串行算法，使用 utility-list 数据结构，在挖掘方法上也相似，本章内容将选择 ULBwTMS-Miner 算法进行并行研究，设计并实现了并行算法 PULBwTMS-Miner。为了更全面的分析 PULBwTMS-Miner 算法的性能，在本章中先是设计实现了 ULB-Miner 算法的并行算法 PULB-Miner，并将实验结果一并引入到 PULBwTMS-Miner 算法的性能分析中。

4.1 并行算法设计原则

在并行计算中，数据或任务的划分方法、并行算法的选择、并行框架的选择都十分重要，甚至会影响并行计算的可行性。

简单地说，并行算法将给定的问题分解为若干个尽可能独立的子问题，然后使用多个处理器在并行机上联合求解问题，最终得到原始问题的解决方案。算法设计是使用计算机解决问题的关键部分[58]。

将一个大的计算划分为许多小的计算，并将其分配给不同的处理器进行并行执行，这是并行算法设计中的两个关键步骤。为了在不同的并行架构或并行编程模型中实现最佳性能，通常需要选择不同的任务划分方式和并行算法模型。本节主要介绍算法设计原理中的分解技术和并行算法模型的相关知识。

4.1.1 并行基本概念

任务(Task)是由程序员定义的计算单元。任务分解将计算分解为具体而明确的小任务，每项小任务都是独立的、相互关联的。进行任务分解，多任务同时执行是减少解决整个问题所需时间的关键。

分解问题所获得的工作量和大小决定了分解的粒度(granularity)。将任务分解成大量的细小任务称为细粒度(fine-grained)，而将任务分解成少量的大任务称为粗粒度(coarse-grained)。

并发度(degree of concurrency)是一个基于分解粒度相关的概念。在并行程序中，可以在任意时刻同时执行的最大任务数量称为并发度。由于任务间的相互依赖性，在大部分情况下，最大并发度总是小于总的任务数量。

进程是执行任务的逻辑计算代理。处理器是实际执行计算的硬件部件。在多数情况下，关于并行算法设计的文字中提到进程时，进程和处理器之间都有一一对应的关系，并且可以假定进程的数量和并行计算机上的 CPU 的数量一样多[58]。

4.1.2 分解技术

分解技术可以概括为四类：探测性分解(exploratory decomposition)、递归分解(recursive decomposition)、推测性分解(speculative decomposition)、数据分解(data decomposition)。

在这四种分解技术中，最常使用的是递归分解和数据分解，因为这两种分解技术能够适用于绝大部分的问题，而探测性分解和推测性分解仅适用于特定性质的问题，具有专用性。现将四种分解技术介绍如下：

1. 递归分解。采用分而治之的策略，首先将问题划分为一组独立的子问题，子问题递归采用相似的划分方法得到更小的子问题。

2. 数据分解。在大数据挖掘领域，数据分解是最常用的分解技术。任务的分解主要分为两个步骤：首先，进行数据划分；然后，在数据划分的基础上，将计算进行任务划分。通过数据分解划分的每一任务执行的计算都成为最终解的一部分。根据数据在任务中出现的时机，主要划分如下：

（1）划分输出数据：只有每个输出都可以作为输入函数进行计算，才能对

输出数据进行划分。输出数据的一个划分，自动将问题划分为几个任务，每个任务负责计算输出数据的部分；

（2）划分输入数据：为输入数据的每个部分创建任务，以尽可能多地利用本地数据执行计算；

（3）划分输入和输出数据：在可能对输出数据进行划分的情况下，对输入数据的划分可能导致附加的并发性；

（4）划分中间结果数据：有的算法通常由多级计算结构组成，中间某级计算的输出作为下一级计算的输入部分。对于这些算法，可以通过划分算法中间级的输入数据或输出数据进行分解。

3. 探测性分解。在探测性分解中，将搜索空间划分为多个小部分，然后并发搜索这些小的部分，只要在任何一个部分中找到希望的结果，其他未完成的任务就可以终止，计算结束。

4. 推测性分解。程序中往往存在很多可能非常重要的分支，这些分支的选择往往取决于上一任务的计算结果，在这种情况下，往往适用推测性分解，当前任务执行计算时，其他任务可以同时执行下一步。

5. 组合分解。递归分解、数据分解、探测性分解、推测性分解，这四种分解方法并不是相互排斥的，往往可以组合应用。

4.1.3 并行算法模型

算法模型是通过选择适当的分解技术和策略构建并行算法的一种典型方法。常见的算法模型主要分为五种：数据并行模型(data-parallel model)、任务图模型(task graph model)、工作池模型(work pool model)、主-从模型(master-slave model)、流水线模型(pipeline model)。

1. 数据并行模型。数据并行模型是最简单的算法模型之一。首先将数据进行划分，每个任务把类似的操作并发的运用到不同数据上。在数据并行模型中，一个重要特点是数据并发度会随着问题规模的增加而增加，在面临更大的问题时可以使用更多的进程来有效地解决。

2. 任务图模型。任何并行算法的计算都可以看作为一个任务依赖图，在任务图模型中使用任务之间的相互关系来提升本地性或减少交互开销。如果某一问

题中与任务对应的数据量远大于与任务相对应的计算，则通常用任务图模型来解决。

3. 工作池模型。工作池模型也叫任务池模型(task pool model)，任务和进程存在多对多的关系，没有必要预映射任务到指定进程。如果在某一问题中，与任务相关的数据量远小于计算量时，通常使用工作池模型。

4. 主-从模型。主-从模型存在一个或多个主进程，并由他们产生任务并分派给从进程，也就是工作者进程。

5. 流水线模型。流水线是生产者和消费者链，它可能是线性链，也可能是一个有向图。对于流水线中的每一个进程，都可以理解为上一个进程数据的消费者和下一个进程数据的生产者。

6. 混合模型。算法模型间并不是相互排斥的，可以用多个模型解决一个问题。

4.2 并行 PULB-Miner 算法研究

Thread Pool 框架和 Fork/Join 框架下是在单机多核处理器环境下进行并行计算的两个不错的框架选择。在实验前期，分别在 Thread Pool 框架和 Fork/Join 框架下进行了可行性分析。

ULB-Miner 算法在单机多核处理器环境下进行并行计算，主要可以分为三个阶段：

（1）挖掘 TWU≥min-util 的项，重构数据库；

（2）将重构数据库进行划分，在多个线程上挖掘局部高效用项集；

（3）各线程挖掘结果合并，输出全局高效用项集。

在上面三个阶段中，最核心且计算量最大的是第二阶段，第一阶段和第三阶段的计算占比较小。如果可以实现第二阶段的并行计算，ULB-Miner 算法进行并行计算就具有可行性。在多个线程上精确挖掘高效用项集，最重要的是确保各线程间数据独立，既能满足全局最小效用阈值 min-util，又不会重复计算数据。

首先在 Thread Pool 框架下对第二阶段进行考虑。Thread Pool 框架没有设定数据或任务的划分方法，可以根据算法的逻辑自行编程划分策略。在高效用项集并行计算领域，通常使用投影数据库技术将原始数据库划分为一个个子数据库，

以确保在不同线程上挖掘结果的独立性和准确性。所以，在 Thread Pool 框架下对 ULB-Miner 算法进行并行计算逻辑上可行。

然后，在 Fork/Join 框架下对第二阶段进行考虑。根据图 2.3 所示的 Fork/Join 框架原理，通过 Fork 操作对数据进行划分，将原始数据库一分为二，二分为四，四分为八，等等，直到子数据库中的事务数达到预先设置的某个数值为止，划分结束。将划分的子数据库分配给各线程，开始局部挖掘高效用项集。Fork 操作划分出的子数据库相当于原始数据库中的某一块数据，属于局部数据，不能通过设置的全局 min-util 来进行剪枝，而且部分数据会在各个线程重复生成局部 Utility-List，无法精确挖掘高效用项集。所以，在 Fork/Join 框架下对 ULB-Miner 算法进行并行计算逻辑上不可行。

通过在 Thread Pool 框架和 Fork/Join 框架下进行可行性分析，本节的并行 ULB-Miner 算法将在 Thread Pool 框架下进行实验。

4.2.1 Thread Pool 框架下 PULB-Miner 算法设计与实现

本节将 ULB-Miner 算法结合 Thread Pool 框架进行并行计算，设计实现了 ULB-Miner 算法的并行算法 PULB-Miner。

PULB-Miner 算法的主要过程展示为 Algorithm 4.1.其中，输入 D 为事务数据库，min_util 为最小效用阈值，输出为高效用项集。

Algorithm 4.1 PULB-Miner algorithm

Input：

D:a transaction database;

Thread:a number of threads;

min_util:a minimum utility threshold;

Output：

The set of high utility itemsets ;

1 Scan D to calculate the TWU of 1-item;

2 Let I* be the list of 1-item such that TWU ≥ min_util;

3 Let ≻ be the total order of TWU ascending values on I*;

4 Scan D again to rebuild the database and build the utility-list of each item i∈I*;

5 **foreach** i in I* **do**

6 **if** (i.SumIutils+i.SumRutils≥ min_util) **then**

7 Scan D to creat projected transaction i-D;

8 Assign the projected transaction to each free thread and build the initial utility-list buffer

UTLBuf, and SULs and build the EUCS;
9 **end if**
10 call the Search procedure;
11 **end**

PULB-Miner 算法的主过程执行以下步骤：

Step 1: 第一次扫描数据库来计算所有 1−项集的事务加权效用 TWU 值, 创建集合 I*存储 TWU≥min_util 的 1-项集，并按照 TWU 值升序排序（总顺序）。

Step 2: 第二次扫描数据库，根据总顺序重构数据库，同时构建 1-项集的 Utility-List。

Step 3：使用剪枝策略 2 对 1-项集进行二次剪枝，对满足条件的项{e、c、b、a}分别构建投影数据库。

Step 4：空闲的线程从任务队列中获取投影数据库，并根据投影数据库构建初始 Utility-List buffer 和 EUCS，调用 Search 过程挖掘高效用项集, 见 Algorithm 3.11。

Step 5：直到每个投影数据库都挖掘完成，将各线程挖掘的结果合并输出，算法结束。

将 Step 4 和 Step 5 的流程绘制成 Thread 并行流程图，如图 4.1 所示。

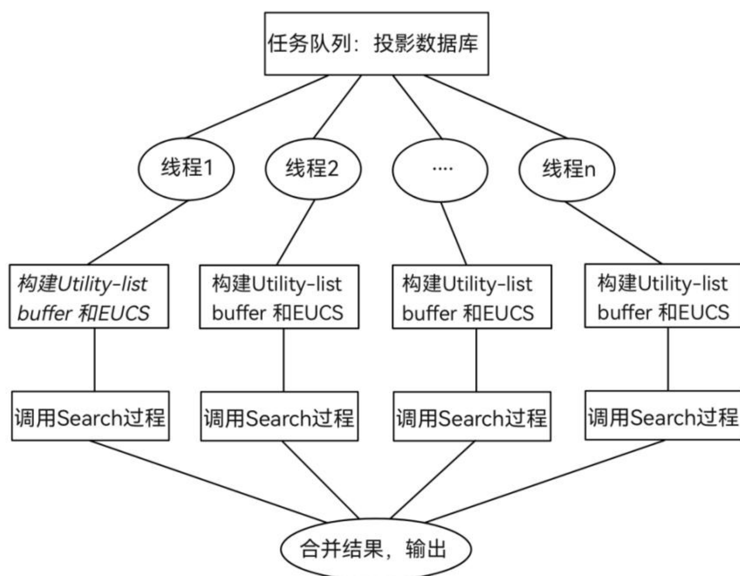


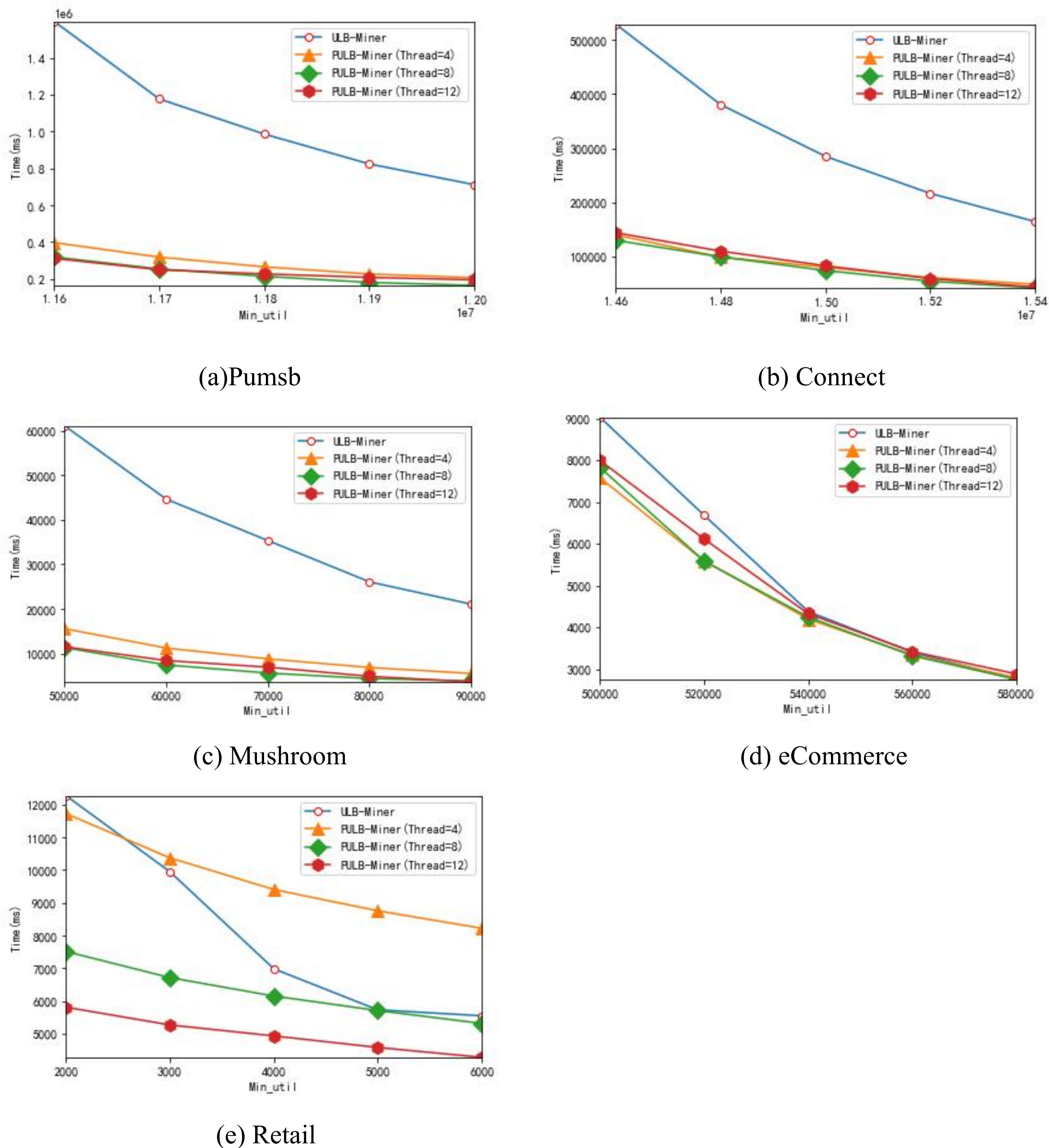
图 4.1 Thread 并行流程图

4.2.2 实验结果与分析

为了评估 PULB-Miner 算法的性能，在表 2.5 中提到的五个数据集上分别在 4 线程、8 线程和 12 线程下进行 5 次实验，记录实验结果并取平均值, 分别在运行时间和运行内存方面与 ULB-Miner 算法进行比较。

（1）运行时间的性能评估

根据 ULB-Miner 算法和 PULB-Miner 算法在 4 线程、8 线程和 12 线程下的不同数据集上的运行时间绘制折线图, 如图 4.2 所示。



(a)Pumsb　　(b) Connect

(c) Mushroom　　(d) eCommerce

(e) Retail

图 4.2 ULB-Miner 算法和 PULB-Miner 算法的运行时间

从图 4.2 可以很直观地看出，除了在 4 线程上的 Retail 数据集的运行时间有所增长，PULB-Miner 算法的运行时间都较 ULB-Miner 算法有明显程度的缩减。

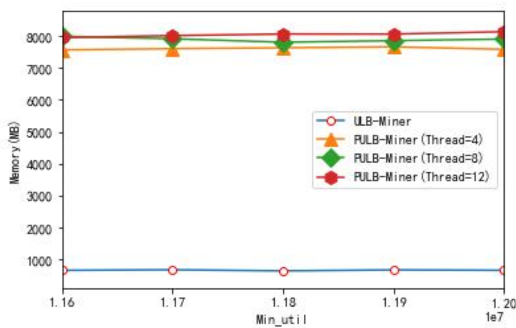
进一步计算PULB-Miner算法相对于ULB-Miner算法的运行时间的变化幅度，将同一数据集上的不同最小效用阈值 min_util 的变化幅度取平均值进行表示，绘制为表 4.1。

表 4.1 运行时间的变化幅度

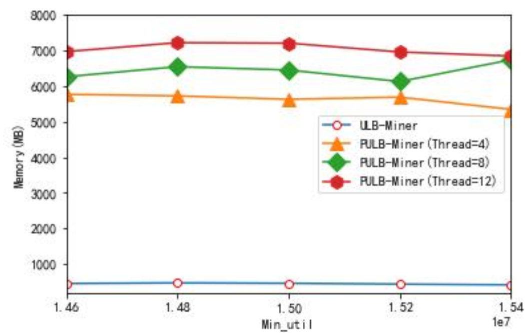
变化幅度 / 数据集 / 线程	Pumsb	Connect	eCommerce	Retail	Mushroom
4	-72.89%	-72.49%	-7.38%	27.64%	-74.43%
8	-78.27%	-74.63%	-7.04%	-17.21%	-82.82%
12	-76.66%	-72.35%	-3.19%	-34.11%	-81.74%

（2）运行内存的性能评估

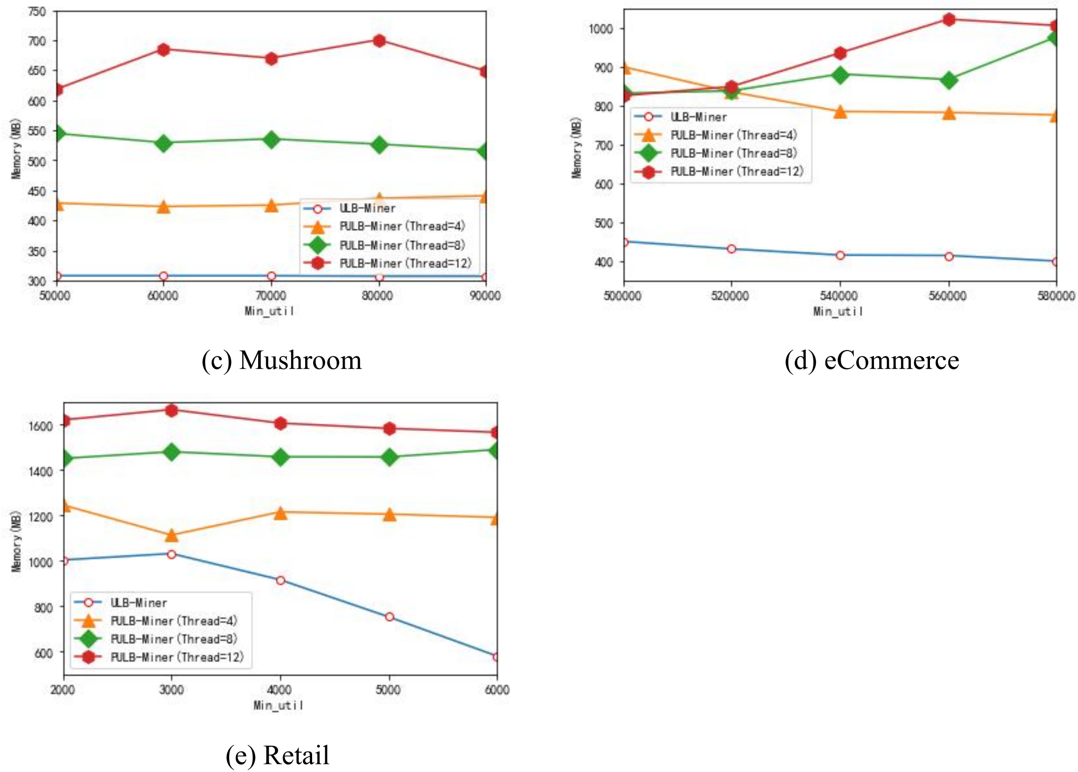
根据 ULB-Miner 算法和 PULB-Miner 算法在 4 线程、8 线程和 12 线程下的不同数据集上的运行内存绘制折线图，如图 4.3 所示。



(a)Pumsb



(b) Connect



(c) Mushroom

(d) eCommerce

(e) Retail

图 4.3 ULB-Miner 算法和 PULB-Miner 算法的运行内存

从图 4.3 可以很直观地看出，PULB-Miner 算法运行内存都明显高于 ULB-Miner 算法。进一步计算 PULB-Miner 算法相对于 ULB-Miner 算法的运行内存的变化幅度，将同一数据集上的不同最小效用阈值 min_util 的变化幅度取平均值进行表示，绘制为表 4.2。

表 4.2 运行内存的变化幅度

变化幅度 / 数据集 / 线程	Pumsb	Connect	eCommerce	Retail	Mushroom
4	1050.80%	1124.34%	40.17%	92.89%	45.90%
8	1093.90%	1298.26%	72.62%	108.71%	79.33%
12	1116.74%	1429.88%	116.15%	120.47%	95.48%

（3）总结

使用多线程进行并行计算，本身就是为了充分利用计算机资源。将串行算法

进行并行计算，多线程同时进行数据挖掘，也意味着运行内存的增大。在计算机运行内存能够承载的情况下，并行计算主要考虑的是运行时间的缩短。由此得出结论，并行算法 PULB-Miner 是可行且有效的。

4.3 并行 PULBwTMS-Miner 算法研究

本节将 ULBwTMS-Miner 算法结合 Thread Pool 框架进行并行计算，设计实现了 ULBwTMS-Miner 算法的并行算法 PULBwTMS-Miner。

4.3.1 Thread Pool 框架下 PULBwTMS-Miner 算法设计与实现

PULBwTMS-Miner 算法的主要过程展示为 Algorithm 4.2。其中，输入 D 为事务数据库，min_util 为最小效用阈值，输出为高效用项集。

Algorithm 4.2 PULBwTMS-Miner algorithm

Input:
D:a transaction database;
Thread:a number of threads;
min_util:a minimum utility threshold;
Output:
The set of high utility itemsets ;
1 Scan D to calculate the TWU of 1-item;
2 Let I* be the list of 1-item such that TWU ≥ min_util;
3 Let ≻ be the total order of TWU ascending values on I*;
4 Scan D again to rebuild the database and build the utility-list of each item i∈I*;
5 Sort transactions in D according to $\succ_T$;
6 delete empty transactions;
7 **foreach** i in I* **do**
8 **if** (i.SumIutils+i.SumRutils≥ min_util) **then**
9 Scan D to creat projected transaction i-D and use projected transaction merging;
10 Assign the projected transaction to each free thread and build the initial utility-list buffer UTLBuf, and SULs and build the EUCS according i-D;
11 **end if**
12 call the Search procedure;
13 **end**

PULBwTMS-Miner 算法的主过程执行以下步骤：

Step 1:第一次扫描数据库来计算所有 1-项集的事务加权效用 TWU 值,创建

集合 I*存储 TWU≥min_util 的 1-项集，并按照 TWU 值升序排序（总顺序）。

Step 2:第二次扫描数据库，根据总顺序重构数据库，并对重构数据库中的事务排序。同时构建 1-项集的 Utility-List。

Step 3：使用剪枝策略 2 对 1-项集进行二次剪枝，对满足条件的项{e、c、b、a}分别构建投影数据库，在过程中对相同事务进行合并。

Step 4：空闲的线程从任务队列中获取投影数据库，并根据投影数据库构建初始 Utility-List buffer 和 EUCS，调用 Search 过程挖掘高效用项集, Search 过程见 Algorithm 3.11。

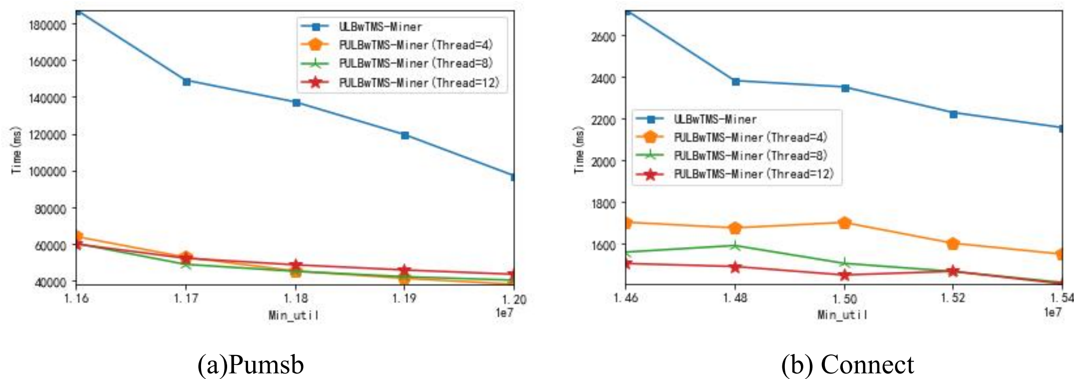
Step 5：直到每个投影数据库都挖掘完成，将各线程挖掘的结果合并输出，算法结束。

4. 3. 2 实验结果与分析

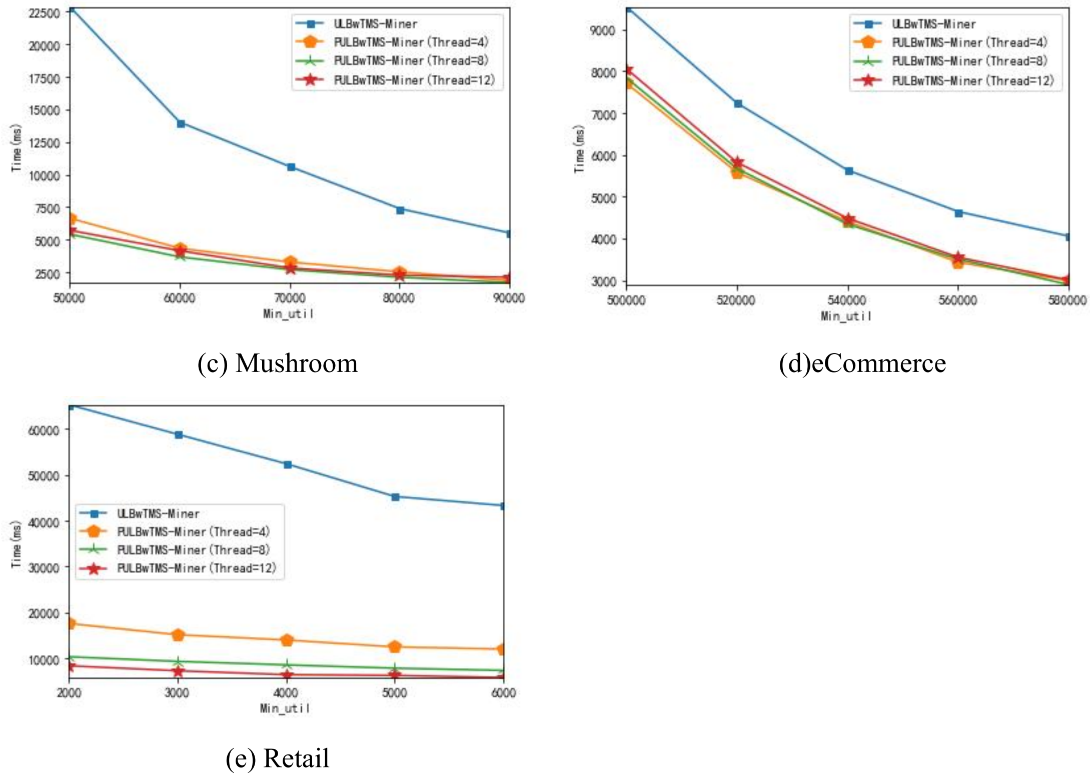
将 PULBwTMS-Miner 算法在表 2.5 中提到的五个数据集上分别在 4 线程、8 线程和 12 线程下进行 5 次实验，记录实验结果并取平均值，分别与 ULBwTMS-Miner 算法和 PULB-Miner 算法进行比较，评估算法在运行时间和运行内存方面的性能。

（1）运行时间的性能评估

根据 ULBwTMS-Miner 算法和 PULBwTMS-Miner 算法在 4 线程、8 线程和 12 线程下的运行时间绘制折线图, 如图 4.4 所示。



(a)Pumsb (b) Connect



(c) Mushroom (d)eCommerce

(e) Retail

图 4.4 ULBwTMS-Miner 算法和 PULBwTMS-Miner 算法的运行时间

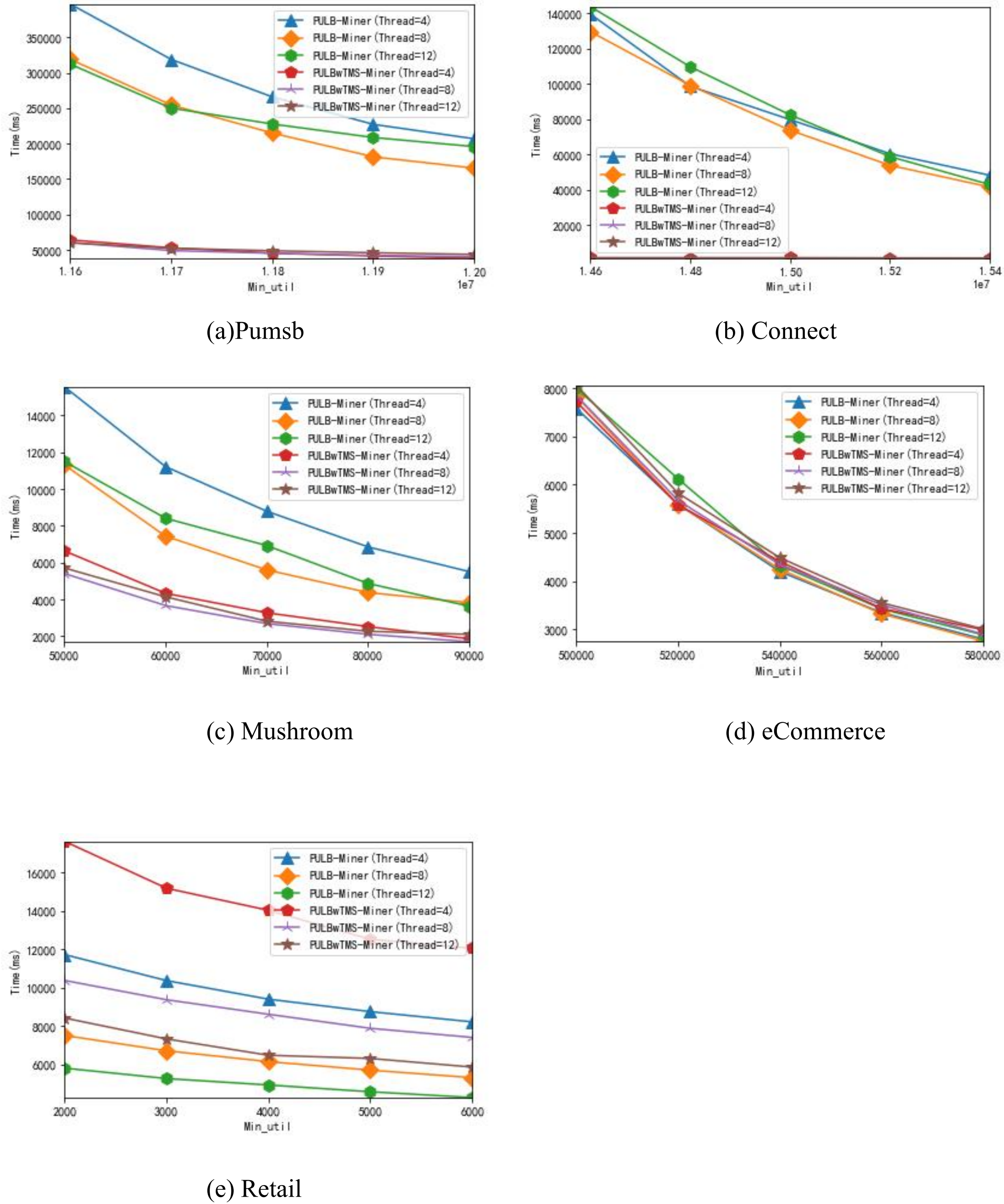
从图 4.4 可以很直观地看出，在五个数据集的不同最小效用阈值 min-util 上，PULBwTMS-Miner 算法在 4 线程、8 线程和 12 线程上的运行时间都较 ULBwTMS-Miner 算法有明显程度的缩减。

进一步计算 PULBwTMS-Miner 算法相对于 ULBwTMS-Miner 算法的运行时间的变化幅度，将同一数据集上的不同最小效用阈值 min_util 的变化幅度取平均值进行表示。绘制为表 4.3。

表 4.3 运行时间的变化幅度

变化幅度 / 数据集 / 线程	Pumsb	Connect	eCommerce	Retail	Mushroom
4	-64.68%	-30.14%	-23.26%	-72.98%	-68.31%
8	-65.02%	-36.03%	-23.11%	-83.44%	-73.14%
12	-62.83%	-37.80%	-21.01%	-86.95%	-70.06%

根据 PULB-Miner 算法和 PULBwTMS-Miner 算法在 4 线程、8 线程和 12 线程下的运行时间绘制折线图, 如图 4.5 所示。



(a)Pumsb (b) Connect

(c) Mushroom (d) eCommerce

(e) Retail

图 4.5 PULB-Miner 算法和 PULBwTMS-Miner 算法的运行时间

从图 4.5 也可以很直观地看出，PULBwTMS-Miner 算法在 Pumsb、Connect 和 mushroom 数据集上的运行时间要明显低于 PULB-Miner 算法，但是在 Retail 数据集上运行时间明显提升，在 eCommerce 数据集上表现不明显。

进一步计算 PULBwTMS-Miner 算法相对于 PULB-Miner 算法在相同线程上的运行时间的变化幅度，将同一数据集上的不同最小效用阈值 min_util 的变化幅度取平均值进行表示。绘制为表 4.4。

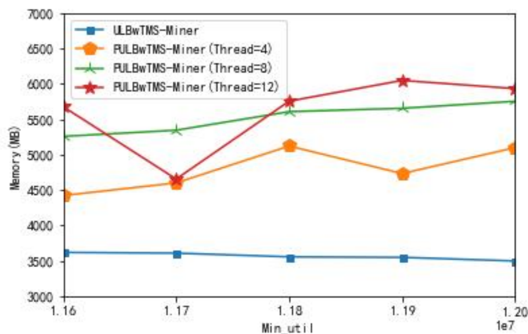
表 4.4 运行时间的变化幅度

变化幅度 \ 数据集 \ 线程	Pumsb	Connect	eCommerce	Retail	Mushroom
4	-82.72%	-97.82%	3.22%	47.12%	-62.23%
8	-78.63%	-97.80%	2.90%	39.10%	-52.50%
12	-78.85%	-98.01%	1.50%	38.11%	-50.41%

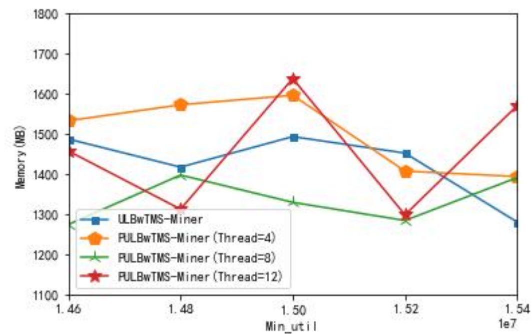
在第三章的实验中已经得出结论：ULBwTMS-Miner 算法不适用于稀疏数据集，在稠密数据集上表现良好。结合表 4.4 中的数据，在 eCommerce 数据集上的运行时间变化幅度为正数，代表 PULBwTMS-Miner 算法的运行时间较 PULBMiner 算法相比有所增加。所以，第三章的实验结论同样适用于并行算法 PULBMiner 和 PULBwTMS-Miner 的对比实验。

（2）运行内存的性能评估

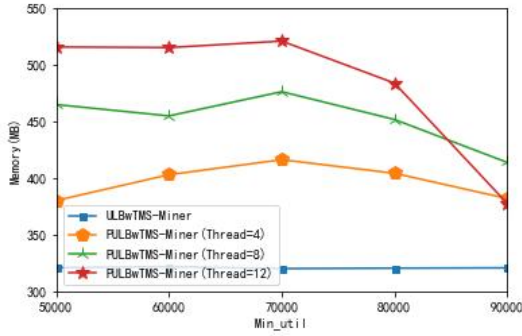
根据 ULBwTMS-Miner 算法和 PULBwTMS-Miner 算法在 4 线程、8 线程和 12 线程下的运行内存绘制折线图，如图 4.6 所示。



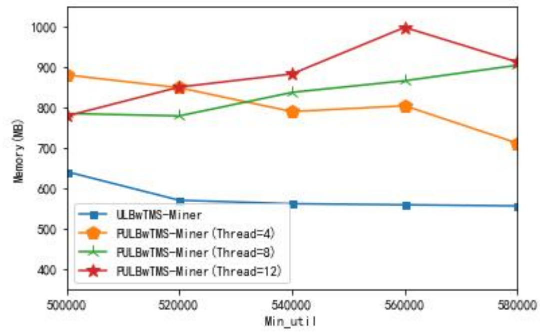
(a)Pumsb



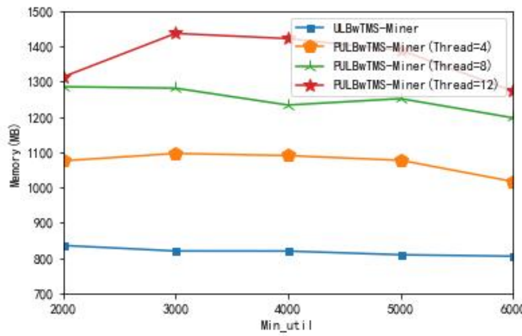
(b) Connect



(c) Mushroom



(d)eCommerce



(e) Retail

图 4.6 ULBwTMS-Miner 算法和 PULBwTMS-Miner 算法的运行内存

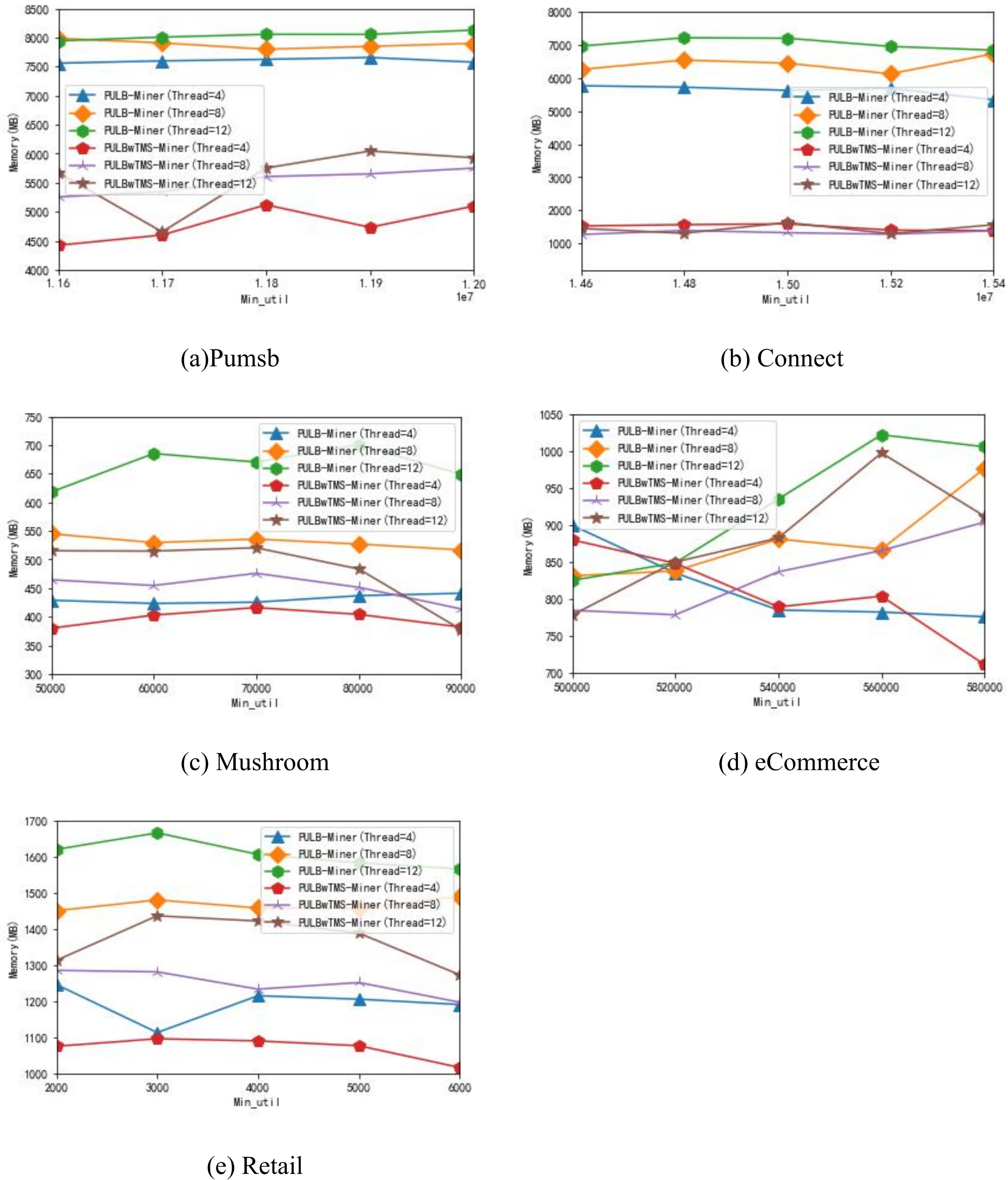
从图 4.6 可以很直观地看出，在 Pumsb、Connect、Mushroom 和 Retail 这四个数据集上，PULBwTMS-Miner 算法运行时占用的内存都明显高于 ULBwTMS-Miner 算法；在 Connect 数据集上，两个算法的占用内存比较接近。

进一步计算 PULBwTMS-Miner 算法相对于 ULBwTMS-Miner 算法的运行内存的变化幅度，将同一数据集上的不同最小效用阈值 min_util 的变化幅度取平均值进行表示，绘制为表 4.5。

表 4.5 运行内存的变化幅度

变化幅度 \ 数据集 / 线程	Pumsb	Connect	eCommerce	Retail	Mushroom
4	34.55%	5.32%	39.67%	30.92%	23.74%
8	54.95%	-5.01%	45.08%	52.75%	40.92%
12	57.52%	2.47%	54.02%	67.04%	50.35%

根据 PULB-Miner 算法和 PULBwTMS-Miner 算法在 4 线程、8 线程和 12 线程下的运行内存绘制折线图, 如图 4.7 所示。



(a)Pumsb (b) Connect

(c) Mushroom (d) eCommerce

(e) Retail

图 4.7 PULB-Miner 算法和 PULBwTMS-Miner 算法的运行内存

从图 4.7 可以很直观地看出，在相同的线程参数上，PULBwTMS-Miner 算法运行时占用的内存都低于 PULB-Miner 算法的运行内存。

进一步计算 PULBwTMS-Miner 算法相对于 PULB-Miner 算法在相同的线程

参数上的运行内存的变化幅度，将同一数据集上的不同最小效用阈值 min_util 的变化幅度取平均值进行表示。绘制为表 4.6。

表 4.6 运行内存的变化幅度

变化幅度 数据集 / 线程	Pumsb	Connect	eCommerce	Retail	Mushroom
4	-36.96%	-73.33%	-1.14%	-10.16%	-7.87%
8	-29.99%	-79.18%	-5.07%	-14.79%	-14.88%
12	-30.21%	-79.28%	-4.58%	-15.03%	-27.34%

（3）总结

通过对 ULBwTMS-Miner 算法和 PULBwTMS-Miner 算法的实验结果分析、PULB-Miner 算法和 PULBwTMS-Miner 算法的实验结果分析，可以得出结论：并行算法 PULBwTMS-Miner 是有效且可行的。

4.4 本章小结

本章首先对并行算法设计原则进行介绍，着重介绍了分解技术和并行算法模型的相关知识。然后对 ULB-Miner 算法在 Thread Pool 框架和 Fork/Join 框架上进行可行性分析。最后根据并行算法设计相关理论，在单机多核处理器环境下，在 Thread Pool 框架下，设计实现了 PULB-Miner 算法和 PULBwTMS-Miner 算法，并通过实验证明了 PULB-Miner 算法和 PULBwTMS-Miner 算法的可行性和有效性。

5 总结与展望

5.1 工作总结

在高效用项集挖掘领域中，使用 utility-list 数据结构的一类算法在挖掘过程中通过对两个 k-项集的 utility-list 进行 Tid 识别来构造(k+1)−项集的 utility-list。Tid 的数量取决于 utility-list 的大小，决定算法的挖掘效率。减小 utility-list 的大小也就是减少了算法在挖掘过程中的计算量，提高算法挖掘效率。本文主要工作总结如下：

（1）通过查阅大量中英文文献，归纳总结了频繁项集和高效用项集的国内外研究现状，并对关联规则、频繁项集和高效用项集的相关定义和性质做出详细介绍。

（2）重点介绍适用于单机并行的 Thread Pool 框架和 Fork/Join 框架，并对适用于多处理机环境并行的 Hadoop 框架和 Spark 框架进行简要说明。

（3）基于对 utility-list 的研究，分别提出了 HUI-Miner 算法和 ULB-Miner 算法的改进算法 HUIwTMS-Miner 和 ULBwTMS-Miner。首先，结合数据库示例详细阐述了 HUI-Miner 算法和 ULB-Miner 算法的实现过程；其次，介绍在改进算法中加入的投影数据库技术和事务合并策略；然后，将改进策略融入到算法中，对改进算法的实现过程进行描述；最后，在 Retail、Pumsb、Connect、Mushroom、eCommerce 数据集的不同最小效用阈值上进行实验，结合数据集的特征，从算法的运行时间和运行内存两个方面进行分析，并验证了 HUIwTMS-Miner 算法和 ULBwTMS-Miner 算法的可行性和有效性。

（4）设计并实现了并行算法 PULB-Miner 和 PULBwTMS-Miner。首先，围绕分解技术和并行算法模型，对并行算法的设计原则进行了简单介绍；其次，分析了 ULB-Miner 算法的并行可行性；然后，利用 Thread Pool 框架实现了 PULB-Miner 算法和 PULBwTMS-Miner 算法；最后，分别在 Retail、Pumsb、Connect、Mushroom、eCommerce 数据集的不同最小效用阈值、不同线程上进行实验，结合数据集的特征，对算法的性能进行分析，实验验证了 PULB-Miner 算法和 PULBwTMS-Miner 算法的可行性和有效性。

5.2 工作展望

本文针对采用 utility-list 数据结构的高效用项集挖掘算法进行改进，虽然提出的改进算法在时间性能上有提升，但是运行内存也有增加。目前数据挖掘的数据量越来越大，在保证提升算法时间性能的情况下，也要注意内存瓶颈的问题。

根据本文研究存在的问题，对未来的研究工作进行如下展望：

（1）本文使用的事务合并策略要对数据库中的不同事务进行排序，相同事务进行合并，但是只在原始数据库上使用事务合并策略并不能明显的提升算法性能。在串行计算时，通过投影数据库技术，将大的数据库进一步划分为多个小的数据库，存储划分的投影数据库在一定程度上增加内存消耗。希望在未来能够提出新的数据结构来存储投影数据库，在减少算法运行时间的同时减少内存消耗；

（2）本文选用两个采用 utility-list 数据结构、具有代表性的高效用项集挖掘算法，通过实验对运行时间和运行内存进行性能评估，验证了改进算法能够取得较好的效果。未来希望将改进策略继续应用到其他具有代表性的算法上，并在更多的数据集上进行实验验证。

（3）本文的并行算法是基于 Thread Pool 框架设计、实现的，仅使用一台计算机完成实验。随着并行计算平台的不断出现，使用计算机集群进行分布式计算越来越成熟，希望未来将 ULBwTMS-Miner 算法应用到并行计算集群上，分析算法在单机和多机并行计算的性能。

参考文献

[1] Hong T P, Kuo C S, Chi S C. Mining association rules from quantitative data[J]. Intelligent data analysis, 1999, 3(5): 363-376.

[2] Zhang Y, Chen T, Jiang Y, et al. Possibilistic c-means clustering based on the nearest-neighbour isolation similarity[J]. Journal of Intelligent & Fuzzy Systems, 2023 (Preprint): 1-12.

[3] Laohakiat S, Sa-Ing V. An incremental density-based clustering framework using fuzzy local clustering[J]. Information Sciences, 2021, 547: 404-426.

[4] Saraswat P. Supervised machine learning algorithm: a review of classification techniques[J]. Integrated Emerging Methods of Artificial Intelligence & Cloud Computing, 2021: 477-482.

[5] Cai S, Zhang L, Zuo W, et al. A probabilistic collaborative representation based approach for pattern classification[C]. Proceedings of the IEEE conference on computer vision and pattern recognition. 2016: 2950-2959.

[6] Li Y, Zhang S, Guo L, et al. NetNMSP: Nonoverlapping maximal sequential pattern mining[J]. Applied Intelligence, 2022: 1-24.

[7] 张春砚,韩萌,孙蕊,等.高效用模式挖掘关键技术综述[J].计算机应用研究,2021,38(02):330-340.

[8] 穆晓芳,邓红霞,郭虎升,等.基于快速高效用项集挖掘的大规模消息流预测算法研究与应用[J].计算机应用与软件,2019,36(11):243-249.

[9] Agrawal R, Srikant R. Fast algorithms for mining association rules[C]. Proc. 20th int. conf. very large data bases, VLDB. 1994, 1215: 487-499.

[10] Mahdi M A, Hosny K M, Elhenawy I. FR-Tree: A novel rare association rule for big data problem[J]. Expert Systems with Applications, 2022, 187: 115898.

[11] Chan R, Yang Q, Shen Y D. Mining high utility itemsets[C]. Third IEEE international conference on data mining. IEEE Computer Society, 2003: 19-19.

[12] Arunkumar M S, Suresh P, Gunavathi C. High utility infrequent itemset mining using a customized ant colony algorithm[J]. International Journal of Parallel Programming, 2020, 48: 833-849.

[13] Pei J, Han J, Lu H, et al. H-mine: Hyper-structure mining of frequent patterns in large databases[C]. proceedings 2001 IEEE international conference on data mining. IEEE, 2001: 441-448.

[14] Uno T, Kiyomi M, Arimura H. LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets[C]. Fimi. 2004, 126.

[15] Hong T P, Lin K Y, Chien B C. Mining fuzzy multiple-level association rules from quantitative data[J]. Applied Intelligence, 2003, 18: 79-90.

[16] Yan D, Qin S, Bhattacharya D, et al. 20th International Workshop on Data Mining in Bioinformatics (BIOKDD 2021)[C]. Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining. 2021: 4175-4176.

[17] Lan K, Wang D, Fong S, et al. A survey of data mining and deep learning in bioinformatics[J]. Journal of medical systems, 2018, 42: 1-20.

[18] Chapela-Campa D, Mucientes M, Lama M. Mining frequent patterns in process models[J]. Information Sciences, 2019, 472: 235-257.

[19] Zhu Y, Liu M, Li L, et al. Online and Offline Blending Learning Exploration of Data Mining Course Based on Internet+[C]. 2022 10th International Conference on Information and Education Technology (ICIET). IEEE, 2022: 224-228.

[20] Riahi F, Schulte O. Model-based exception mining for object-relational data[J]. Data Mining and Knowledge Discovery, 2020, 34: 681-722.

[21] Agrawal R, Srikant R. Fast algorithms for mining association rules[C]. Proc. 20th int. conf. very large data bases, VLDB. 1994, 1215: 487-499.

[22] Han J, Pei J, Yin Y, et al. Mining frequent patterns without candidate generation: A frequent-pattern tree approach[J]. Data mining and knowledge discovery, 2004, 8: 53-87.

[23] Zaki M J. Scalable algorithms for association mining[J]. IEEE transactions on knowledge and data engineering, 2000, 12(3): 372-390.

[24] Savasere A, Omiecinski E, Navathe S. An E cient Algorithm for Mining Association Rules in Large Databases[C]//Proceedings of the 21st International Conference on Very Large Databases (VLDB). 1995: 432-444.

[25] Zaki M J, Gouda K. Fast vertical mining using diffsets[C]. Proceedings of the

ninth ACM SIGKDD international conference on Knowledge discovery and data mining. 2003: 326-335.

[26]Borgelt C. Keeping things simple: finding frequent item sets by recursive elimination[C]. Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations. 2005: 66-70.

[27]Deng Z H, Wang Z H. A new fast vertical method for mining frequent itemsets[J]. Int J Comput Intell Syst, 2010, 3: 733-744.

[28]Deng Z H, Wang Z H, Jiang J J. A new algorithm for fast mining frequent itemsets using N-lists[J]. Science China Information Sciences, 2012, 55: 2008-2030.

[29]Deng Z H, Lv S L. Fast mining frequent itemsets using Nodesets[J]. Expert Systems with Applications, 2014, 41(10): 4505-4512.

[30]Rymon R. Search through Systematic Set Enumeration[C]//KR. 1992: 539-550.

[31]Liu Y, Liao W, Choudhary A. A two-phase algorithm for fast discovery of high utility itemsets[C]. Advances in Knowledge Discovery and Data Mining: 9th Pacific-Asia Conference, PAKDD 2005, Hanoi, Vietnam, May 18-20, 2005. Proceedings 9. Springer Berlin Heidelberg, 2005: 689-695.

[32]Yao H, Hamilton H J, Butz C J. A foundational approach to mining itemset utilities from databases[C]. Proceedings of the 2004 SIAM International Conference on Data Mining. Society for Industrial and Applied Mathematics, 2004: 482-486.

[33]Ahmed C F, Tanbeer S K, Jeong B S, et al. Efficient tree structures for high utility pattern mining in incremental databases[J]. IEEE Transactions on Knowledge and Data Engineering, 2009, 21(12): 1708-1721.

[34]Tseng V S, Wu C W, Shie B E, et al. UP-Growth: an efficient algorithm for high utility itemset mining[C]. Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining. 2010: 253-262.

[35]Tseng V S, Shie B E, Wu C W, et al. Efficient algorithms for mining high utility itemsets from transactional databases[J]. IEEE transactions on knowledge and data engineering, 2012, 25(8): 1772-1786.

[36] Lin C W, Hong T P, Lu W H. An effective tree structure for mining high utility itemsets[J]. Expert Systems with Applications, 2011, 38(6): 7419-7424.

[37] Liu M, Qu J. Mining high utility itemsets without candidate generation[C]. Proceedings of the 21st ACM international conference on Information and knowledge management. 2012: 55-64.

[38] Liu J, Wang K, Fung B C M. Direct discovery of high utility itemsets without candidate generation[C]. 2012 IEEE 12th international conference on data mining. IEEE, 2012: 984-989.

[39] Fournier-Viger P, Wu C W, Zida S, et al. FHM: Faster high-utility itemset mining using estimated utility co-occurrence pruning[C]. Foundations of Intelligent Systems: 21st International Symposium, ISMIS 2014, Roskilde, Denmark, June 25-27, 2014. Proceedings 21. Springer International Publishing, 2014: 83-92.

[40] Zida S, Fournier-Viger P, Lin J C W, et al. EFIM: a highly efficient algorithm for high-utility itemset mining[C]. Mexican international conference on artificial intelligence. Cham: Springer International Publishing, 2015: 530-546.

[41] Dawar S, Goyal V, Bera D. A hybrid framework for mining high-utility itemsets in a sparse transaction database[J]. Applied intelligence, 2017, 47: 809-827.

[42] Duong Q H, Fournier-Viger P, Ramampiaro H, et al. Efficient high utility itemset mining using buffered utility-lists[J]. Applied Intelligence, 2018, 48: 1859-1877.

[43] Duong Q H, Liao B, Fournier-Viger P, et al. An efficient algorithm for mining the top-k high utility itemsets, using novel threshold raising and pruning strategies[J]. Knowledge-Based Systems, 2016, 104: 106-122.

[44] Wu J M T, Lin J C W, Tamrakar A. High-utility itemset mining with effective pruning strategies[J]. ACM Transactions on Knowledge Discovery from Data (TKDD), 2019, 13(6): 1-22.

[45] Shen W, Zhang C, Fang W, et al. Efficient high-utility itemset mining based on a novel data structure[C]. 2021 IEEE International Smart Cities Conference (ISC2). IEEE, 2021: 1-6.

[46] Wu P, Niu X, Fournier-Viger P, et al. UBP-Miner: An efficient bit based high utility itemset mining algorithm[J]. Knowledge-Based Systems, 2022, 248:

108865.

[47]（美）Jiawei Han, Micheline Kambei, Jian Pei.数据挖掘：概念与技术（原书第三版）[M].范明，孟小峰.北京：机械工业出版社，2012.7：158-159

[48]贾红学. 板的特大增量步算法及并行计算[D]. 上海交通大学, 2015.

[49]Agrawal R, Shafer J C. Parallel mining of association rules[J]. IEEE Transactions on knowledge and Data Engineering, 1996, 8(6): 962-969.

[50]张乐，魏昕怡，徐苏，林两位. 面向大数据的数据库划分 FP-Growth 改进算法[J]. 南昌大学学报(理科版)，2022，46(05)：570-576.

[51]黄文海.Java 多线程编程实战指南[M].北京：电子工业出版社，2015.10：113-127

[52]Borthakur D. HDFS architecture guide[J]. Hadoop apache project, 2008, 53(1-13): 2.

[53]Zaharia M, Chowdhury M, Franklin M J, et al. Spark: Cluster computing with working sets[C]. 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10). 2010.

[54]Xin R S, Gonzalez J E, Franklin M J, et al. Graphx: A resilient distributed graph system on spark[C]. First international workshop on graph data management experiences and systems. 2013: 1-6.

[55]Meng X, Bradley J, Yavuz B, et al. Mllib: Machine learning in apache spark[J]. The journal of machine learning research, 2016, 17(1): 1235-1241.

[56]Alias S, Norwawi N M. pSPADE: Mining sequential pattern using personalized support threshold value[C]. 2008 International Symposium on Information Technology. IEEE, 2008, 2: 1-8.

[57]Nguyen T D D, Tung N T, Pham T, et al. Parallel approaches to extract multi-level high utility itemsets from hierarchical transaction databases[J]. Knowledge-Based Systems, 2023: 110733.

[58]（美）Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar.并行计算导论（原书第一版）[M].张武，毛国勇，程海英等.北京：机械工业出版社，2005.1：63-103.

攻读硕士学位期间发表的论文及科研情况

发表论文：

[1] 张晓蝶, 张迪, 李岩. 频繁项集挖掘算法研究综述[J].电脑采购, 2023(6):14-16.

[2] 秦伟德, 杨海军, 张晓蝶. 基于无监督聚类算法的多准则 ABC 分析应用[J].物流时代周刊, 2022(12):27-29. DOI:10.12323/j.issn.1673-0542.2022.12.004.

[3] 刘文杰, 秦伟德, 张晓蝶. 闭项集挖掘算法研究综述[J].大众标准化, 2022(8):151-153. DOI:10.3969/j.issn.1007-1350.2022.08.054.

致 谢

时光飞逝，落笔于此，就意味着研究生三年的求学生活即将结束。坐在实验室里，看着面前的每一张桌子，座位上的每一位同伴，回想这三年的点点滴滴，颇多感慨。

感谢我的学校，兰州财经大学，因为在这里给我提供了学习和科研的平台，不仅让我在学术上有所进步，更多的是培养了学术研究的思维，为终身学习打下基础。

感谢我的恩师杨海军教授。与老师第一次见面、参加的第一场报告会等等都犹如昨日。从确定研究方向、科研方法到确定毕业论文选题、实验和完成，老师始终给予我耐心的指导和支持。恩师开阔的视野，精益求精的工作作风，深深感染着我。每次与老师讨论学术问题，老师总是能一语中的，点出我的疑惑和误区...老师就像是我人生道路的明灯，在我迷茫时给我前进的方向和动力，在日常的生活中也是倍加关心。能够成为您的学生是我的荣幸，再次向您致以最真挚的感谢！

感谢信息工程与人工智能学院的每位教师，感谢研究生秘书。老师们严谨的学术作风深深影响着我，为我前行的道路照亮了一束耀眼的光。

感谢我的师姐刘文杰和我的师兄秦伟德，在学术上给了我很多宝贵的建议，同时谢谢我的师妹李岩，你的乐观态度深深感染着我。

感谢实验室的郭泓和韩运龙，我们共同探讨学业，在找工作时的信息共享，感谢实验室其他的伙伴，感谢我们一起留下的美好回忆。

感谢 2021 级管理科学与工程的全体同学陪我走过了这难忘的一段历程。

感谢我的室友，林林、璠璠和嫣嫣，谢谢你们在我研究生生涯中带给我的快乐，我会永远记得我们一起夜谈、一起逛街、一起互相鼓励。

谢谢我的父母，谢谢你们二十多年的培养与付出，我所取得的所有成就都离不开你们的教育，谢谢我的家人，让我从小可以成长的无忧无虑，谢谢你们教会我的所有，谢谢你们给我最坚实的依靠。

感谢答辩组的所有专家教授，在百忙之中抽出宝贵时间指导我的研究工作，提出宝贵意见和建议，鞭策我不断努力不断进步！

最后，感谢所有帮助过我和关心过我的人，感谢所有遇见的人。