

融合多分组归并的券商数据 Shuffle 和数据倾斜算法

曹亚坤, 唐小勇

长沙理工大学计算机与通信工程学院, 湖南 长沙 410114

摘要

在证券行业, 用户数据处理和分析是核心技术, 对业务决策和风险控制具有重要的影响。然而, 证券公司庞大的用户数据规模和复杂的数据关系导致大数据计算面临 Shuffle 操作和数据倾斜问题。现有的 Shuffle 和数据倾斜优化方法或依赖于硬件升级, 或存在领域局限性, 难以针对性解决该问题。为此, 基于证券行业用户数据的特点, 提出了一种基于用户关系的多分组归并算法 (multi group merging algorithm, MGMA)。该算法通过有效分组和优化处理策略, 显著提升计算效率, 并降低计算资源消耗。实验表明, 相较于无优化对照组, MGMA 算法的数据倾斜率为 20%, 内存占用为 72%, 计算用时为 61%, 且上述 3 项指标均优于其他 4 种对比优化方法。

关键词

Shuffle 操作; 数据倾斜; 预处理; 券商数据

中图分类号: TP391; F49

文献标志码: A

doi: 10.11959/j.issn.2096-0271.2025074

Multi group merging algorithm for solving data Shuffle and data skew of securities companies

CAO Yakun, TANG Xiaoyong

School of Computer and Communication Engineering, Changsha University of Science & Technology, Changsha 410114, China

Abstract

In the securities industry, the processing and analysis of user data are critical technologies that significantly impact business decision-making and risk control. However, the vast scale and complexity of user data securities companies led to significant Shuffle operations and data skew issues in big data computations. Existing optimization methods either relied on hardware upgrades or were limited by domain-specific constraints, failing to address the problem effectively. To resolve this, a multi-group merging algorithm (MGMA) based on user relationships was proposed, which improved computational efficiency and reduces resource consumption through effective grouping and optimization strategies. Experimental results showed that, compared to the no optimized (NO) control group, MGMA algorithm achieved a 20% data skew rate, 72% memory usage, and 61% computation time. All three indicators surpass those of the other four comparison optimization methods.

Key words

Shuffle operations, data skew, preprocessing, data of securities companies

0 引言

随着科技与金融行业的深度融合，大数据平台在证券行业的典型应用^[1]落地日益增加，证券公司越来越依赖大数据管理和分析，以支持决策^[2-3]、优化服务和提升效率。随着时间的推移，用户的交易记录、行为习惯等数据不断累积，数据规模越来越大，这对数据处理提出了更高的要求。对于证券行业来说，如何准确、高效地处理大量的用户数据，以提供更好的服务、更精准的决策支持，已经成为一项重要的任务。

大数据技术已经广泛应用于金融风险控制^[4-5]、客户画像^[6]等领域，然而，大数据环境下数据的处理和管理也面临一系列挑战。在实践中，证券行业的数据量十分庞大，且数据每天的增量惊人，为方便这些数据的管理，往往采用天分区存储模式。另外，用户之间的关联关系复杂，如同属于某个投资顾问、同属于某个机构、购买同样的金融产品、推荐人关系，等等。庞大的数据量和复杂的用户关系使得数据处理难度大大增加，在分布式计算框架下，此类场景易引发频繁的 Shuffle^[7-8]操作及数据倾斜现象的发生，使系统整体效率大大降低。

1 相关研究

目前，大数据 Shuffle 优化算法主要包括：S-Shuffle 算法^[9]、NV-Shuffle（基

于非易失内存的 Shuffle 优化）算法^[10]、EC-Shuffle（动态擦除编码优化）算法^[11]、基于溢出历史的自适应调度算法^[12]、SARSO（Spark-based apriori with reduced Shuffle optimization）算法^[13]、基于文件预合并和分区重组的混合 Shuffle 模型^[14]、Shuffle Fetch 负载均衡策略^[15]等。针对数据倾斜问题的优化算法有基于水库抽样的动态分区平衡方法^[16]、RTLBPA（reduce task load balancing partitioning algorithm）^[17]、基于数据倾斜模型的混合划分优化方案^[18]等。另外，相关计算优化算法还包括：Wang 和 Zhou^[19]提出的基于最低成本的 RDD（resilient distributed dataset）缓存优化策略、Reena 等^[20]提出的基于敏感性参数的数据分区方法、SSPO-DQN（Spark shuffle performance optimization using deep Q-network）算法^[21]等。

Shuffle 优化算法，如 S-Shuffle 算法、NV-Shuffle 算法、EC-Shuffle 算法、基于溢出历史的自适应调度算法和 SARSO 算法，主要通过减少数据冗余、优化资源分配和提升硬件利用率来提高性能。例如，S-Shuffle 算法通过优化数据传输和处理流程，减少了 CPU 的消耗和数据冗余，而 NV-Shuffle 算法则利用非易失内存技术显著提升了计算性能，尤其是在 RAMDisk 上的表现。EC-Shuffle 通过擦除编码技术减少了网络流量和存储冗余，同时提高了系统的容错能力。基于溢出历史的自适应调度算法和 SARSO 算法则通过动态调整资源分配和减少 RDD 操作的 Shuffle 开销，有效提升了 Shuffle 的性能和可扩展性。然而，这些 Shuffle 优化算法也存在一些缺

点：NV-Shuffle算法依赖于特定的硬件支持，这在硬件不兼容的情况下难以实施；S-Shuffle算法在选择性较高时，其性能可能不如传统的G-Shuffle算法；基于溢出历史的自适应调度算法的实验验证主要在较小规模的集群上进行，需要进一步的实际工程验证和规模扩展。

针对数据倾斜问题的优化算法，如基于Key分布的动态重分区策略、基于水库抽样的动态分区平衡方法和RTLBP算法，都通过优化数据分布和负载平衡来提高处理效率。这些算法能够有效减少数据倾斜带来的性能瓶颈，提高Spark并行计算的效率。例如，基于水库抽样的动态分区平衡方法在优化应用程序完成时间和提高并行计算性能方面表现出色，而RTLBP算法在云环境下显著降低了作业的开销。尽管如此，数据倾斜优化算法也有其局限性，例如，基于水库抽样的动态分区平衡方法在数据倾斜度较高的情况下效果有限，尤其在倾斜度大于0.4时，运行时间仍然较大。

这些Shuffle优化算法和数据倾斜优化算法在提升性能和减少资源消耗方面各有优势，但也存在硬件依赖、实验局限和适用范围有限等缺点。特别是在券商大数据领域，现有算法并不能很好地解决所遇到的问题。因此，在实际应用中需要根据券商数据中台所存储数据的特点来选择最合适的优化方式。

2 券商数据中台架构下的数据处理挑战与需求

券商数据中台是一个高度集成和协同的系统，其多层次、多组件的设计，实现了数据的高效采集、处理、存储和应用，

为券商的业务发展提供了强大的数据支撑。同时，数据运维体系和数据安全体系的建立和完善，确保了数据中台的稳定、高效、安全运行，为决策支撑、风险控制^[22]、精准营销^[23]等券商业务发展提供坚实的支撑。券商数据中台的架构如图1所示。

2.1 券商数据分析

对于证券公司而言，用户数据的处理与分析是其核心活动之一。证券业务范围广泛，涉及个人客户、机构客户、基金管理和财富管理等多个领域，因此形成了庞大的用户数据集合。随着金融科技的迅猛发展和数字化转型的加速，证券公司已积累了大量用户数据，包括交易记录、资金流动和客户行为等，这些数据以天为单位持续增长。

(1) 券商数据特点

券商数据的特点主要体现在以下4个方面。首先，数据规模庞大。证券交易活动频繁和用户类型多样，由此产生了大量

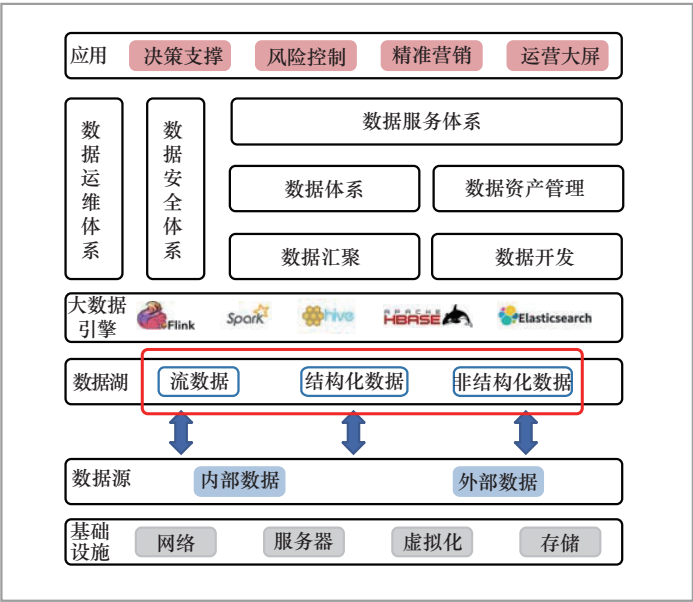


图1 券商数据中台架构

交易、资金和客户数据，且这些数据逐日增长，因此，高效处理和分析这些数据成为证券公司的迫切需求。其次，数据关系复杂。用户间存在多种关联，如客户经理归属、投资组合共同购买等，导致大数据计算中频繁的 Shuffle 操作。随着数据量增加，计算效率下降，带来数据处理挑战。再次，数据来源多样化。券商客户群体包括个人投资者、机构投资者和高净值客户，数据类型和来源各异，增加了处理和分析的复杂性。最后，数据价值巨大。通过数据处理和分析，证券行业能实现个性化服务、提升决策质量和优化投资组合，增强竞争力和盈利能力。

在券商大数据平台中，为方便数据存储和处理，大多数数据会以天分区或者月分区的形式进行存放。用户交易数据存储目录结构如下。

User_Transactions/
20180101
20180102
...

这样的数据目录结构方便对数据进行归档和备份，在进行数据迭代运算时也有较大的优势，但在一些其他的计算场景下，

这样的数据存储结构却起到了相反效果。以计算券商投资顾问下所有用户账户波动率为例，计算需要用到用户的每日交易数据和用户的每日净值数据，计算中先关联投资顾问与用户的关系，将所有用户数据分组到每个投资顾问下，然后再进行计算，其数据流如图2所示。

(2) 计算中的 Shuffle

在大数据处理中，Shuffle 是数据重新分配的过程，通常发生在需要跨节点协作的操作中，如 groupByKey、reduceByKey、join 等。Shuffle 操作会消耗大量的输入/输出 (input/output, I/O)、网络、CPU 等资源，是影响大数据处理性能的关键因素。在证券业务统计中，如统计每个客户经理名下的客户资产情况，这时需对整个客户数据进行 groupByKey 操作，而用户数据的存储方式却是以天分区的形式进行存储的，此时计算就会产生大量 Shuffle 操作。

(3) 计算中的数据倾斜

数据倾斜是指在分布式计算过程中，数据分布不均匀，导致某些计算节点的计算负载过重，而其他节点空闲，从而影响整体的计算效率。以计算证券公司各营业部的每日总体资产变化情况为例，由于每个营业部的用户数据量是不一样的，一些大营业部的用户数量可能是小营业部用户数的几十倍甚至上百倍，当几个大营业部业务的数据 Shuffle 到同一任务节点上时，数据倾斜就产生了。

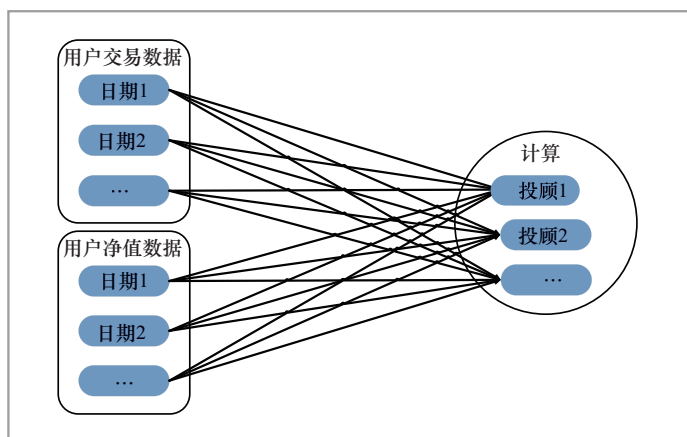


图2 投顾用户账户波动率计算数据流

2.2 传统方法

基于券商数据的特点，过往的经验是在进行大数据计算之前，先进行一次数据预分组操作。传统预分组方法参考了大数据计算中的宽依赖 (wide dependency)

和窄依赖 (narrow dependency) 的概念, 而宽依赖和窄依赖是分布式计算中的关键概念。宽依赖指的是一个操作需要来自多个父节点的数据, 这通常会导致数据的 Re-Shuffle 和大量的网络传输, 从而增加计算成本。相反, 窄依赖指的是一个操作只需要来自单个父节点的数据, 这样可以实现本地化处理, 减少数据传输和计算延迟。

传统的预分组方法在每次计算特定关系指标之前, 需对数据进行一次基于该关系的分组, 以达到计算窄依赖的效果。在处理前文中提到的投资顾问相关的计算时, 算法首先会基于用户与投资顾问的关系进行分组, 将同一分组下的用户数据写入同一文件块, 然后进行指标计算。传统方法数据流如图3所示。

然而, 当存在多个用户关系相关的计算时, 传统方法需要重复进行基于单一关系的分组操作, 这不仅增加了编码工作量, 而且每次分组都会再次消耗一定时间。同时, 由于单一关系下的数据量并不均衡, 传统预分组算法并不能解决数据倾斜的问题。

3 基于用户关系的多分组归并算法

基于对传统预分组方法的优劣分析, 本文提出了一种基于用户关系的多分组归并算法 (multi group merging algorithm, MGMA)。MGMA 能够在单个分组中同时合并多种用户关系, 同时还能保证每个分组下的数据量相对均衡。这种设计不仅利用了窄依赖的优势, 减少了数据传输和计算延迟, 还通过均衡数据分布有效地解决了数据倾斜问题。MGMA 数据流如图4所示。

MGMA 在提高计算效率的同时, 也简化了编码过程, 为处理复杂关系型数据提供了更高效和灵活的解决方案。MGMA 由关系分解、分组初始化、基于关系的分组合并、散组合并4部分算法组成。其中, 关系分解可分解不同类型的关系, 基于关系的分组合并可重复对不同类型的关系进行合并, 算法执行过程如图5所示。

3.1 用户关系分解

在券商数据中台的业务场景中, 用户关系 (如客户经理归属、投资组合关联等)

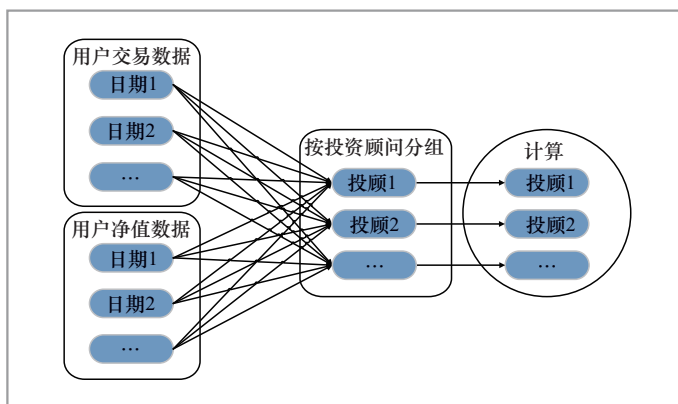


图3 传统方法数据流

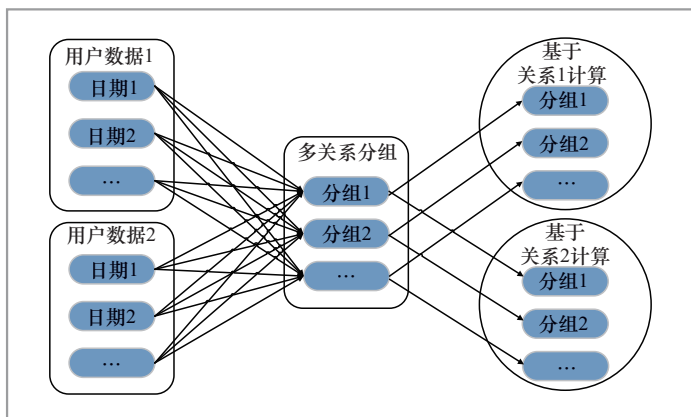


图4 MGMA数据流

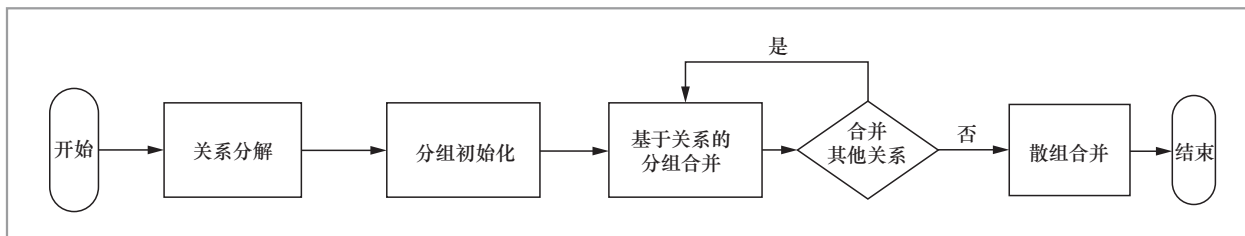


图5 算法执行过程

具有多样性和动态性。传统方法需针对单一关系独立抽取数据，导致重复计算和存储冗余。MGMA 提出多关系统一映射机制，通过关系分解将异构用户关系整合为统一的数据结构，为后续分组合并奠定基础。

从用户各类关系数据中可以抽取出用户-关系的集合，然后根据关系对应用户生成 $\text{RelationID} \rightarrow \{U1、U2、U3\}$ ，用户关系分解示意如图6所示。

因为不同类型的关系可能存在ID重复的情况，所以每一类关系只能进行单独抽取。

设 P 为用户-关系对的集合，其中每个元素 (u, r) 表示一个用户 u 与一个关系 r 的配

对，即 $P = \{(u, r) | u \in U, r \in R\}$ ，其中 U 是用户集合， R 是关系集合。定义一个从 R 到 U 的幂集 2^U 的映射（relation to users, $R2Us$ ），即 $R2Us: R \rightarrow 2^U$ 。

输入：用户-关系对列表 $\text{List}(\text{UserID}, \text{RelationID})$ ，其中 RelationID 表示关系类型（如客户经理ID、投资组合ID等）。

输出：关系-用户字典 $\text{Dict}\{\text{RelationID}: \text{Set}[\text{UserID}]\}$ ，键为 RelationID ，值为关联的 UserID 集合。

目标：将分散的用户-关系对聚合为“关系→用户集合”的映射，同时隔离不同关系类型以避免ID冲突（例如，同一数值ID可能对应不同的关系类型）。

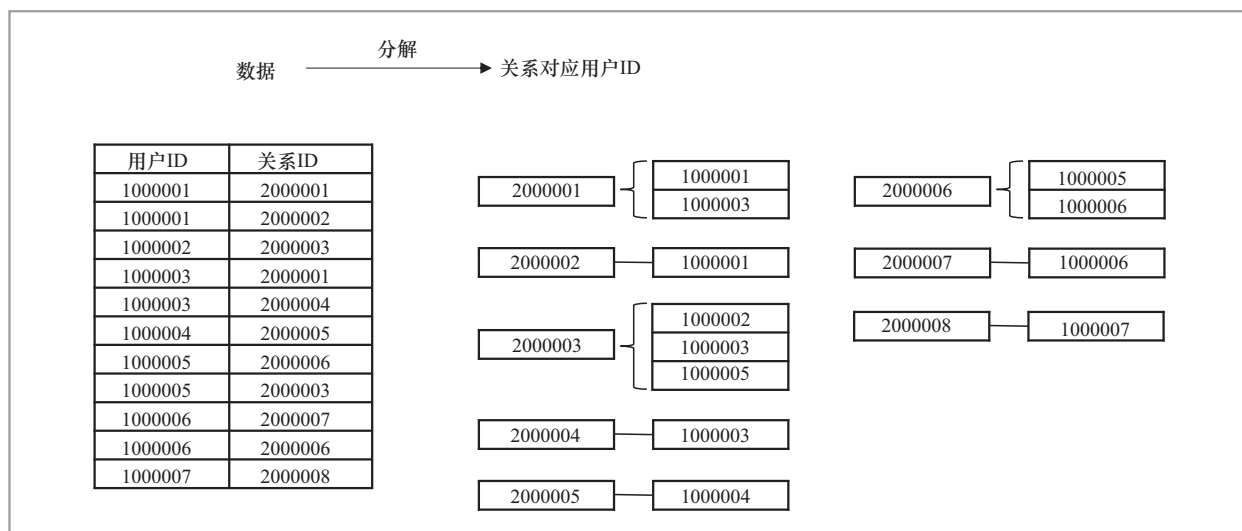


图6 用户关系分解示意图

算法的步骤可以用以下数学式表示。

(1) 初始化一个空字典 R2Us:

$$\text{R2Us} = \emptyset \quad (1)$$

(2) 对于每个 $(u, r) \in P$, 进行以下操作:

$$\forall (u, r) \in P: \begin{cases} r \notin \text{dom}(\text{R2Us}), \text{R2Us}[r] = \emptyset \\ \text{R2Us}[r] = \text{R2Us}[r] \cup \{u\} \end{cases} \quad (2)$$

(3) 返回 R2Us:

$$\text{return R2Us} \quad (3)$$

综上, 可以将算法的整个过程用以下数学式表示:

$$\text{R2Us} = \bigcup_{(u, r) \in P} \begin{cases} r \mapsto (\text{R2Us}[r] \cup \{u\}), r \in \text{dom}(\text{R2Us}) \\ r \mapsto \{u\}, r \notin \text{dom}(\text{R2Us}) \end{cases} \quad (4)$$

用户关系分解算法如算法 1。

算法 1 用户关系分解

输入: 用户-关系对列表 List (UserID, RelationID)

输出: 关系-用户字典 Dict {RelationID: Set[UserID]}

1. R2Us $\leftarrow \{\}$ // 初始化空字典

2. 对每个 (UserID, RelationID) $\in$ 输入列表执行:

2.1 若 RelationID $\notin$ R2Us

2.1.1 R2Us [RelationID] $\leftarrow \emptyset$

// 创建新键值, 隔离不同关系类型

2.2 添加用户: R2Us [RelationID].

add(UserID)

3 返回 R2Us

3.2 用户分组初始化

在分布式计算场景下, 初始分组粒度直接影响后续数据合并的灵活性与计算负载均衡性。本步骤通过“用户粒度原子化”

设计, 将每个用户独立划分至唯一初始分组, 为多维度分组合并提供细粒度控制基础, 避免分组粒度粗糙导致的二次倾斜风险。

分组初始化是将每个用户进行单独分组, 保证数据的最小粒度, 同时也进行分解操作, 将用户 ID 和分组 ID 分解为 [UserID, GroupID] 和 GroupID $\rightarrow \{U1, U2, \dots\}$ (初始化时一个 GroupID 只会对应一个 UserID), 这样可以方便后续进行分组合并。

设 U 为用户 ID 的集合, 将其表示为 $U = \{u_1, u_2, \dots, u_n\}$ 。然后定义两个输出集合: 用户-分组对的集合 (user group pair, UGPs), 从分组 ID 到用户集合的映射 (group to users, G2Us)。将初始的分组 ID 定义为 $G_0 = 100001$ 。

输入: 用户 ID 列表 List[UserID]。

输出: 用户-分组对列表 UGPs: 记录用户与初始分组的映射关系 (UserID, GroupID)。

分组-用户字典 G2Us: 存储分组到用户的逆向映射 GroupID $\rightarrow \{UserID\}$ 。

目标: 构建双向可追溯的分组结构, 支撑后续动态分组合并操作。

算法的步骤可以用以下数学式表示:

(1) 初始化一个空列表 UGPs 和一个空字典 G2Us

$$\text{UGPs} = \emptyset \quad (5)$$

$$\text{G2Us} = \emptyset \quad (6)$$

(2) 初始化一个分组 ID G

$$G = G_0 = 100001 \quad (7)$$

(3) 对于每个用户 ID $u \in U$, 进行以下操作

$$\forall u \in U: \begin{cases} \text{UGPs} \leftarrow \text{UGPs} \cup \{(u, G)\} \\ G \notin \text{dom}(G2Us), G2Us[G] = \emptyset \\ G2Us[G] = G2Us[G] \cup \{u\} \\ G = G + 1 \end{cases} \quad (8)$$

(4) 返回 UGPs 和 G2Us

return UGPs, G2Us (9)

综上，可以用一下数学式表示整个过程：

$$\text{UGPs} = \bigcup_{u \in U} \{(u, G_u)\} \quad (10)$$

其中， G_u 是每个用户 u 对应的唯一分组 ID。

G2Us =

$$\bigcup_{u \in U} \begin{cases} G_u \mapsto (G2Us[G_u] \cup \{u\}), G_u \in \text{dom}(G2Us) \\ G_u \mapsto \{u\}, G_u \notin \text{dom}(G2Us) \end{cases} \quad (11)$$

其中， $G_u = G_0 + (u - u_l)$ 为每个用户 u 对应的唯一分组 ID，从 G_0 开始，每次递增 1。

初始化分组算法如算法 2。

算法 2 用户分组初始化

输入：用户 ID 列表 List[UserID]

输出：用户-分组列表 UGPs，分组-用户字典 G2Us

```

1  UGPs ← []
// 初始化用户-分组对列表
2  G2Us ← {}
// 初始化分组-用户字典
3  GroupID ← 100001
// 起始分组 ID (可配置)
4  对每个 UserID ∈ 输入列表执行
    4.1 添加元组到 UGPs: UGPs.append
        ((UserID, GroupID))
    4.2 若 GroupID ∉ G2Us:
        4.2.1 UGPs [GroupID] ← {UserID}
// 新建分组项
    4.3 否则

```

4.3.1 UGPs [GroupID].add(UserID)

// 理论上此分支不会执行 (初始分组唯一)

4.4 GroupID ← GroupID + 1

// 严格递增，保证分组唯一性

5 返回 UGPs 和 G2Us

3.3 基于关系的分组合并

在多关系场景下，分组合并需实现动态负载均衡与关系关联性最大化的双重目标。本步骤通过“大吞小”合并策略（用户数多的分组吸收少的分组），逐步将同一关系下的用户聚合至同一计算单元，减少跨节点 Shuffle 操作，同时避免传统预分组方法因固定划分导致的数据倾斜复发问题。

完成用户关系分解和用户分组初始化之后，如果某关系对应的用户数大于 1，则通过该用户关系下的所有用户找到对应的分组并进行去重。如果去重后的分组数量仍大于 1，则比较这些分组下的用户数量，用户少的向用户多的分组进行合并，合并完后更新预分组。基于关系分组的单次执行过程如图 7 所示。

随着关系的遍历，分组将会逐步合并，最终留下较大的分组。分组合并的过程如图 8 所示。

设 U 为用户 ID 的集合， G 为分组 ID 的集合， R 为关系 ID 的集合。设 R2Us 为关系到用户的映射，即 $R2Us: R \rightarrow 2^U$ ；UGPs 为用户-分组对集合，即 $UGPs \subseteq U \times G$ ；G2Us 为分组到用户的映射，即 $G2Us: G \rightarrow 2^U$ 。

输入：

R2Us: 关系-用户字典 (来自第 3.1 节)

UGPs: 用户-分组列表 (来自第 3.2 节)

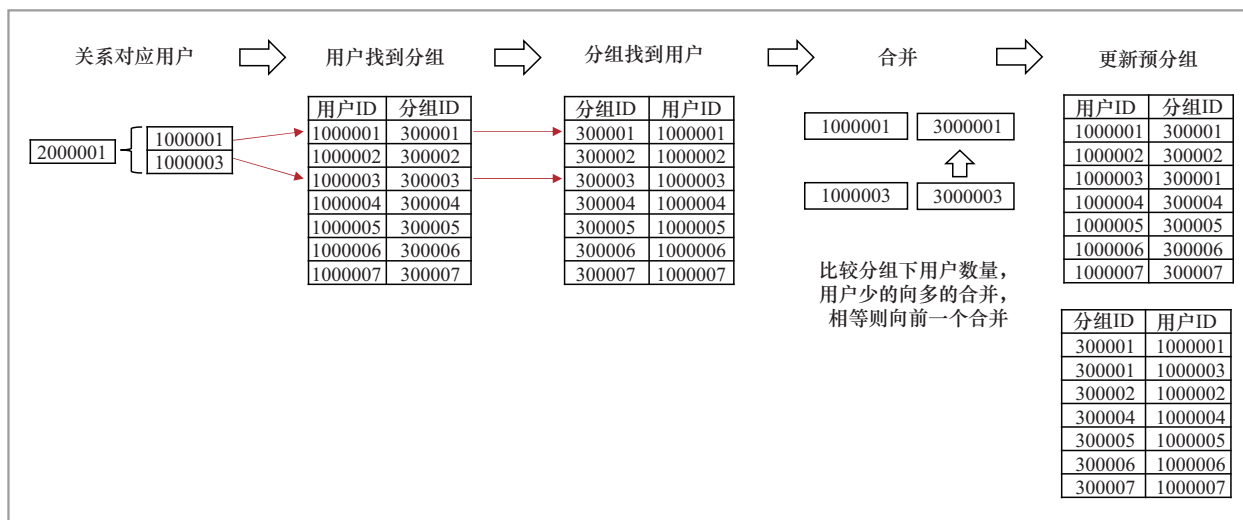


图7 基于关系分组的单次执行过程

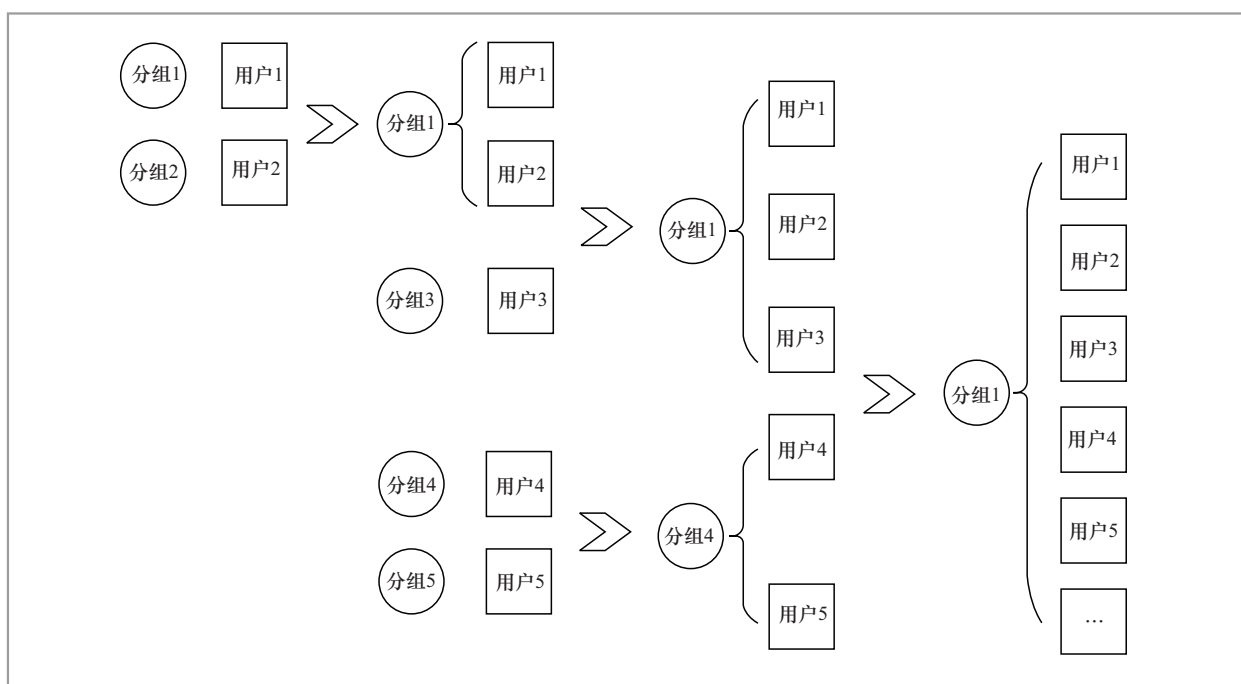


图8 分组合并过程

G2Us: 分组-用户字典 (来自第3.2节)

输出: 更新后的用户-分组列表与分组-用户字典

目标: 通过层级合并, 将同一关系下的用户收敛至最少分组, 优化窄依赖计算

效率。

步骤1: 遍历关系并处理用户数量大于1的关系

对于每个 $(r, U_r) \in R2Us$, 如果 $|U_r| > 1$:

(1) 初始化一个空集合 Unique

GroupIDs (UniGIDs):

$$\text{UniGIDs} = \emptyset \quad (12)$$

(2) 对于每个 $u \in U_r$, 找到对应的分组 ID g 并添加到 UniGIDs:

$$\forall u \in U_r, (u, g) \in \text{UGPs} \Rightarrow \text{UniGIDs} = \text{UniGIDs} \cup \{g\} \quad (13)$$

如果 $|\text{UniGIDs}| > 1$, 则继续执行合并操作。

步骤2: 合并操作

(1) 初始化一个字典 GroupSizes (GSs) 来存储每个分组的大小:

$$\forall g \in \text{UniGIDs} \setminus \{g_{\max}\}: \begin{cases} \forall u \in \text{G2Us}[g]: \begin{cases} (u, g) \rightarrow (u, g_{\max}) \in \text{UGPs} \\ \text{G2Us}[g_{\max}] = \text{G2Us}[g_{\max}] \cup \{u\} \end{cases} \\ g \neq g_{\max}, \text{G2Us} = \text{G2Us} - \text{G2Us}[g] \end{cases} \quad (17)$$

步骤3: 返回更新后的集合 UGPs 与 G2Us

$$\text{return UGPs, G2Us} \quad (18)$$

基于关系分组的算法代码如算法3。

算法3 基于关系的分组合并

输入: 关系-用户字典 R2Us, 用户-分组列表 UGPs, 分组-用户字典 G2Us

输出: 更新后的 UGPs、G2Us

1 对每个 (RelationID, Users) $\in$ R2Us.items() 执行

1.1 若 $|\text{Users}| \leq 1$, 跳过当前关系

1.2 否则

1.2.1 初始化唯一分组集合 UniGIDs $\leftarrow \emptyset$

1.2.2 对每个 UserID $\in$ Users

1.2.2.1 查询当前分组: GroupID $\leftarrow$ UGPs 中 UserID 对应的分组

1.2.2.2 将 GroupID 加入 UniGIDs

$$\text{GSs} = \emptyset \quad (14)$$

(2) 对于每个 $g \in \text{UniGIDs}$, 存储分组 g 的用户数量到 GSs:

$$\forall g \in \text{UniGIDs}, \text{G2Us}[g] \Rightarrow \text{GSs}[g] = |\text{G2Us}[g]| \quad (15)$$

(3) 找到具有最大用户数量的分组 ID $g_{\max}$:

$$g_{\max} = \arg \max_{g \in \text{UniGIDs}} \text{GSs}[g] \quad (16)$$

(4) 对于每个 $g \in \text{UniGIDs} \setminus \{g_{\max}\}$, 将其用户合并到 $g_{\max}$, 并更新 G2Us 和 UGPs:

1.3 若 $|\text{UniGIDs}| > 1$

1.3.1 初始化分组规模字典 GSs $\leftarrow \{\}$

1.3.2 对每个 GroupID $\in$ UniGIDs

1.3.2.1 用户数 size $\leftarrow |\text{G2Us}[\text{GroupID}]|$

1.3.2.2 GSs [GroupID] $\leftarrow$ size

1.3.3 确定主分组: $g_{\max} \leftarrow \arg \max(\text{GSs})$

1.3.4 对每个 GroupID $\in$ UniGIDs $\setminus \{g_{\max}\}$

1.3.4.1 迁移用户 for UserID $\in$ G2Us[GroupID]:

UGPs 中更新 UserID 的分组为 $g_{\max}$

// 正向映射更新

G2Us[$g_{\max}$].add(UserID) // 逆向映射更新

1.3.4.2 删除旧分组: delete G2Us [GroupID]

2 返回 UGPs、G2Us

3.4 散组合并

在完成基于关系的分组合并后，可能仍存在长尾分组分布问题（少量大分组伴随大量零散小组）。本步骤通过动态平衡归约机制，将分组数量收敛至预设值，同时满足每个分组的用户数接近理论均衡值，解决传统方法因固定阈值导致的过合并或欠合并问题。

预设最终分组值为 N ，总用户数为 M ，首先将初步预分好的组按照组内用户数从大到小进行排序，取前 N 个作为基础分组，然后按照排序顺序遍历剩下的分组，从第 $N+1$ 个分组开始合并，即第 $N+1$ 分组与第 N 分组合并， $N+2$ 分组与 $N-1$ 分组合并， $N+3$ 分组与 $N-2$ 分组合并。当完成一轮之后进行翻转， $2N+1$ 分组与第 1 分组合并， $2N+2$ 与第 2 分组合并。第二轮合并完成之后再次翻转，以此类推，直到所有分组都遍历合并完成。如果某组内用户数 $\geq M/N$ ，则不再向该组进行合并，而是跳过向下一个分组进行合并。

输入：

User Group Pairs：用户-分组列表（来自第 3.3 节）

Group To Users：分组-用户字典（来自第 3.3 节）

N ：目标分组数量

M ：总用户数

输出：均衡化后的用户-分组列表与分组-用户字典

目标：在控制分组数量的同时，最大化分组间负载均衡性。

设当前分组集合为 $G=\{g_1, g_2, \dots, g_k\}$ ，按用户数降序排列后取前 N 个基础分组作为 $G_{\text{base}}=\{g_1, g_2, \dots, g_N\}$ ，剩余分组为 $G_{\text{residual}}=\{g_{N+1}, \dots, g_k\}$ 。合并规则可形式化定义如下。

$$\forall g_i \in G_{\text{residual}}, \text{TargetIndex} = \begin{cases} i \bmod N & \text{if Flip} = 1 \\ N - (i \bmod N) & \text{otherwise} \end{cases} \quad (19)$$

其中，翻转因子 $\text{Flip} \in \{-1, 1\}$ 控制合并方向，避免局部过载。

散组合并算法代码如算法 4。

算法 4 散组均衡合并

输入：用户-分组列表 UGPs，分组-用户字典 G2Us，目标分组数 N ，总用户数 M

输出：更新后的 UGPs 和 G2Us

```

1. 按分组用户数降序排序 G2Us → SortedGroups
2.      BaseGroups ← SortedGroups[: N]
   // 保留前 N 个为基础分组
3. CurrentIndex ← N
   // 从第 N+1 个分组开始处理
4. Flip ← 1
   // 初始合并方向为正向
5. 当 CurrentIndex < |SortedGroups| 时循环
   5.1 TargetIndex ← CurrentIndex % N
   5.2 若 Flip == -1
       TargetIndex ← N - TargetIndex - 1 // 反向索引（从右向左）
   5.3 目标分组用户数 size ← |BaseGroups[TargetIndex]|
   5.4 若 size ≥ M/N:
       5.4.1 CurrentIndex ← CurrentIndex + 1
       5.4.2 跳过后续步骤，继续循环
   5.5 迁移用户
       for UserID ∈ SortedGroups[CurrentIndex]:
           UGPs 中更新 UserID 的分组为 BaseGroups[TargetIndex]
```

BaseGroups[TargetIndex].add
(UserID)

5.6 删除原分组：del SortedGroups
[CurrentIndex]

5.7 $CurrentIndex \leftarrow CurrentIndex + 1$

5.8 若 $CurrentIndex \% N = 0$
Flip $\leftarrow -Flip$ // 每处理完 N 个分组后
翻转方向

6 返回 UGPs, BaseGroups

3.5 算法复杂度分析

(1) 用户关系分解

用户关系分解的空间复杂度取决于存储用户与关系映射的数据结构，即字典。在这个字典中，每个关系 RelationID 对应一个集合，该集合存储了所有与该关系关联的用户 UserID。对于每个关系，如果该关系是第一次出现，则需要创建一个新的集合。集合的大小取决于与该关系关联的用户数量。假设有 n 个 {UserID, RelationID} 对，且每个关系平均关联 m 个用户，则字典中的总元素数量为 n 。每个集合平均大小为 m ，因此，总空间复杂度为 $O(n \cdot m)$ 。在理想情况下，每个关系只关联一个用户，此时空间复杂度为 $O(n)$ 。因为每个操作都是常数时间，则用户关系分解的时间复杂度为 $T(n)$ 。

(2) 用户分组初始化

在用户分组初始化中，假设有 n 个用户，则 UGPs 列表的大小为 n ，每个组对应的集合平均大小也为 n （因为每个用户都会被分配到一个组）。因此，字典的总空间复杂度为 $O(n)$ 。此时操作同样为常数时间，即时间复杂度为 $T(n)$ 。

(3) 基于关系的分组合并

基于关系的分组合并的空间复杂度主

要取决于 5 个数据结构：R2Us、UGPs、G2Us、UniGIDs 和 GSs。

存储和更新数据：对于每个关系，如果关联的用户数量大于 1，则需要存储这些用户的组 ID 到 UniGIDs 中，并计算每个组的规模。

假设有 m 个关系，每个关系平均关联 k 个用户，且每个用户属于一个组。对于每个关系，UniGIDs 的大小平均为 k ，GSs 的大小也为 k 。因此，额外空间复杂度为 $O(k)$ ，平均的空间复杂度为 $O(n + mk)$ 。

遍历关系和用户：对于每个关系，需要检查关联的用户数量，这需要 $O(k)$ 的时间复杂度。

处理每个关系的用户：对于每个用户，需要查找其对应的组 ID，并将组 ID 添加到 UniGIDs 中。

计算组大小：对于每个在 UniGIDs 中的组 ID，需要计算其组大小，这需要时间复杂度 $T(n)$ 。

合并组：对于除最大组的每个组，需要更新 UGPs 和从 G2Us 中移除组，这需要时间复杂度 $T(n)$ 。假设有 m 个关系，每个关系平均关联 k 个用户，则总的时间复杂度为 $T(mk + mn)$ 。

(4) 散组合并

散组合并算法中 UGPs、G2Us 为输入参数，其他的临时变量并不会占用。

初始化额外数据结构：BaseGroups 的大小为 N ，这是固定的。

存储和更新数据：算法在运行过程中会更新 UGPs 和 G2Us，但这些更新不会增加额外的空间需求。

因此，额外空间复杂度为 $O(n)$ 。总的空间复杂度为 $O(m + n)$ ，其中 m 是用户数量， n 是最终的组数量。

排序：首先需要对 G2Us 中的组按用户数量进行降序排序，这需要 $O(n \cdot \log n)$ 的

时间复杂度。

遍历和合并组：需要遍历剩余的组，并将其合并到 BaseGroups 中。对于每个组，需要更新 UGPs 和从 G2Us 中移除组，这需要 $O(m)$ 的时间复杂度。

假设有 m 个用户，最坏的情况下每个用户属于一个组，则总的时间复杂度为 $O(m \cdot \log m + m) = O(m \cdot \log m)$ 。

综上，MGMA 的空间复杂度为 $O(n)$ ，时间复杂度为 $O(n \cdot \log n)$ 。

4 实验

本研究的实验过程包括实验平台的选择和搭建、算法的设计与实现、实验结果的比较与评估等。

4.1 实验平台准备

实验平台参考实际生产环境，平台系统使用 Centos7.2，大数据平台采用原生 Hadoop2.7+Spark3.2。

Hadoop 是一个用于大规模数据集的分布式存储和处理的开源平台，它以分布式文件系统 HDFS、分布式计算框架 MapReduce、作业调度和集群资源管理框架 YARN 为核心，解决了海量数据存储、资源任务调度和海量数据计算的问题。

Spark 和 MapReduce 都是分布式计算框架，但由于 Spark 基于内存计算，具有丰富的算子、高容错性和机器学习库等优

势，许多项目选择部署到 Spark 上。

实验用环境具体配置见表1。

4.2 MGMA 算法的平台集成与应用

在本研究中，MGMA 主要用于在数据加载阶段对数据进行优化处理。虽然 MGMA 并未对 Hadoop 和 Spark 的核心架构进行直接修改，但它充分利用了这些平台的功能和特性来提升计算效率。具体来说，MGMA 通过以下方式与 Hadoop 和 Spark 平台进行集成。

与 Hadoop 平台的集成：MGMA 在 Hadoop 平台上的实现主要依赖于 HDFS（Hadoop 分布式文件系统）的数据存储和读取功能。在数据加载阶段，MGMA 对存储在 HDFS 上的数据进行多分组融合归并处理，生成优化后的中间数据。

与 Spark 平台的集成：在 Spark 平台上，MGMA 利用了 Spark 的内存计算特性，进一步提升了数据处理的效率；合理利用 Spark 算子（如 select、filter、agg、convert、collect 等）对用户关系数据进行提取组合。

4.3 实验数据准备

本文所用实验数据集为券商脱敏数据，已经去除用户身份信息，并对用户 ID 和交易行为数据进行了脱敏处理，用户总数约 100 万，提取了用户 N 年前，半年区间跨度的用户交易数据和用户资产每日变化数

表1 集群配置表

节点类型	数量/台	CPU	内存/GB	硬盘/GB	任务模式	系统环境
Master	1	8核16线程,2.6 GHz	16	200	YARN	Centos7.2, JDK1.8
Slave	9	8核16线程,2.6 GHz	16	200		

据。同时提取了用户账户关系、用户家庭关系、用户推荐人关系、用户理财产品购买情况、投资顾问用户关系、用户投资组合关系、用户营业部关系总共7组关系。数据量总计1.8亿条左右，数据集总大小为14.62 GB。

实验所用计算指标都是实际生产环境中真实使用的，包括对用户的基础指标计算（如夏普比率、波动率、詹森 Alpha、收益率、最大回撤、止盈、止损等），营业部的计算指标有实际交易用户数、用户交易量、用户风控能力、用户胜率等，计算的不同关系对应的各类指标总计100多个，还有不同统计维度的归因分析。

实验对照组分别是，无优化（no optimization, NO）、MGMA、S-Shuffle算法、基于水库抽样的动态平衡分区法（DBPM）、传统预分组方法（National）。

实验主要统计评估以下3个指标：计算用时、计算峰值内存占用、Executor最大任务执行时间差。

（1）计算用时

统计计算所用时长，直观比较各类算法带来的计算效率提升。用时包括从计算任务启动到计算结果写回至HDFS的完整时间。

任务启动时间：从提交计算任务到任

务开始执行的时间。

数据加载时间：从HDFS读取数据并进行预处理的时间。

计算执行时间：算法在集群中执行计算任务的时间。

结果写回时间：将计算结果写回HDFS的时间。

统计完整计算任务的用时，能够更全面地评估各算法在实际生产环境中的性能表现。

（2）计算峰值内存占用

完成同样计算使用的内存越少，说明计算资源使用率越高，整个计算集群的稳定性也更高。

（3）Executor最大任务执行时间差

用于判断计算时是否出现了数据倾斜现象。

4.4 用户基础指标计算实验

本实验中只计算了用户的基础指标，没有其他用户关系的指标计算，计算的维度为单个用户，各算法的计算用时如图9所示。

从图9可以看出，MGMA在处理不涉及复杂关系的计算任务时，与传统预分组方法在计算时间上表现相近。这是因为在当前情况下，MGMA相当于只进行了分组初始化和散组合并，与传统预分组方法并无实质区别。同时，S-Shuffle算法在处理参与计算字段较少的数据集时展现出显著优势，而DBPM算法在单用户指标计算场景下并未体现出其设计优势。

4.5 用户指标加营业部指标计算实验

在前一个实验的基础上增加了营业部指标的计算，以评估用户指标加营业部指标计算用时，实验结果如图10所示。

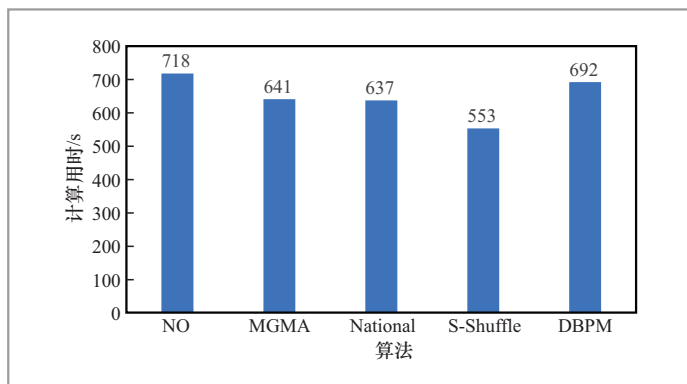


图9 用户指标计算用时

从图10来看, MGMA已经明显优于传统预分组方法。虽然在进行基于关系的分组合并之后, 分组结果与传统预分组方法一致, 但之后的散组合并对数据进行了均衡的再次合并, 而传统分组方法数据本身的特点(各营业部人数并不均衡)导致了分组数据的不均衡, 最终使得两个预分组方法产生了较大差距。S-Shuffle算法由于参与本次计算的字段还是不多, 因此仍然有不错的优化效果。DBPM算法在当前情况也取得了不错的优化效果。

在分析完本次分组结果之后, 研究人员发现了一个问题, 由于每个用户都会对应一个营业部, 而营业部数量实际上并不会很多, 当前抽取的关系数据中, 营业部总数不到100个。在有其他关系参与MGMA的计算时, 有可能将大多数用户分在少数的几个分组中, 甚至于同一个分组中。这样的效果不是本次实验想要的, 因而在后续实验中, 屏蔽掉了用户营业部关系。

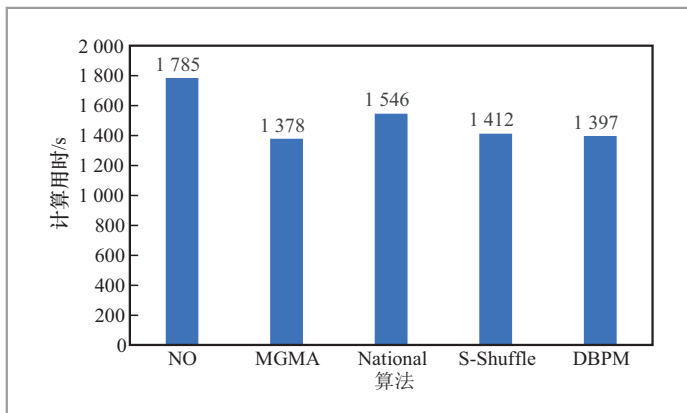


图10 用户指标加营业部指标计算用时

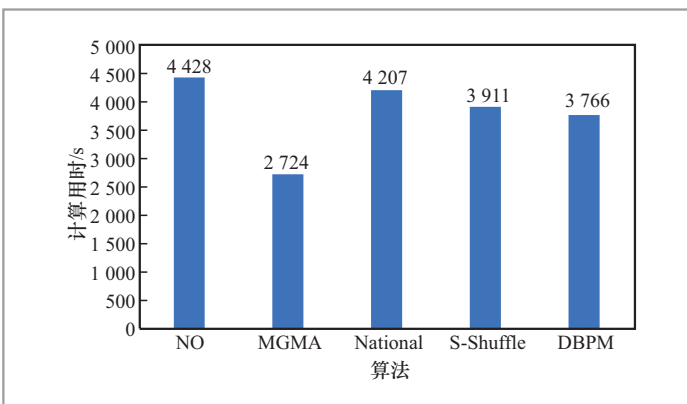


图11 多种关系指标计算用时

4.6 多种关系指标计算实验

多种关系指标计算实验计算了除营业部关系外其他6组关系对应的指标, 计算用时如图11所示。

从图11可以看出, MGMA明显优于其他对照组。由此可见, 参与计算的关系越多, MGMA的表现越好。

4.7 多关系指标计算统计分析实验

在多种关系指标计算实验的基础上, 让对照组都进行了5次计算, 并统计计算用时、任务执行当中峰值内存占用量、Executor最大任务执行时间差。

(1) 计算用时

统计了对照组所有计算任务的每次计

算用时总和, 以及5轮实验中每组的平均值。具体统计结果如图12所示。

从图12可以看到, NO平均用时最长为4555s, MGMA平均计算用时最少为2790s。其他方法也都有一定的优化效果, 传统预分组平均用时4198s, S-Shuffle算法平均用时为3948s, DBPM算法平均用时为3785s。

(2) 峰值内存占用

通过对任务执行时集群状态的日志监控, 以及对计算任务DAG的分析, 所有实验对照组的内存占用峰值都出现在计算复杂度最高的任务中, 即不同统计维度的归因分析。这也是最后一组计算任务。各对

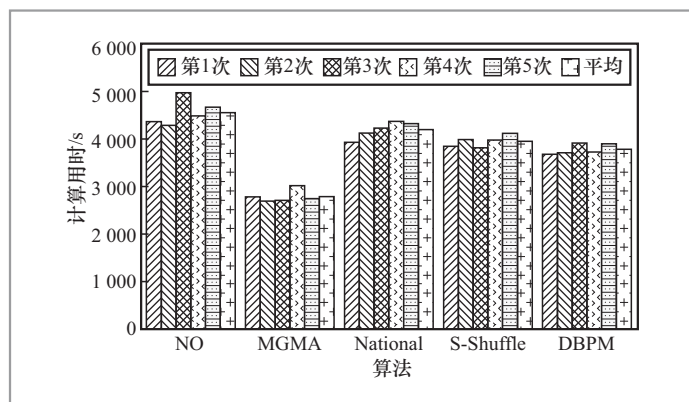


图12 计算用时

照组具体情况如图13所示。

从图13来看，NO峰值内存占用的均

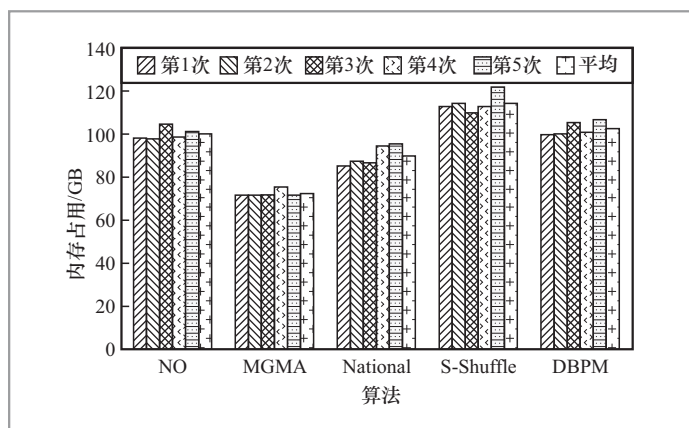


图13 峰值内存占用

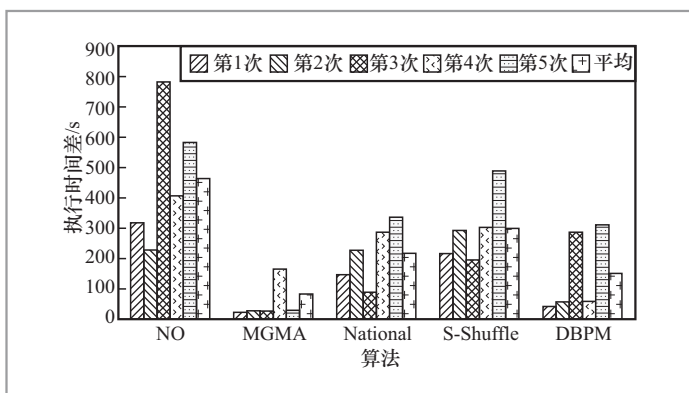


图14 执行节点最大任务时间差

值是100.1 GB，MGMA的峰值内存占用的均值最小，为72.41 GB，传统预分组方法峰值内存占用的均值为89.84 GB。值得注意的是，虽然S-Shuffle算法在参与计算字段较少的计算任务中可以有效降低Shuffle量，但是每次计算字段发生变化时都需要单独进行一次转RDD操作，这反而使S-Shuffle算法的峰值内存占用均值成为各对照组中最高的，达到了114.34 GB。相较于NO，DBPM算法并未展现出峰值内存优化的效果，其均值为102.58 GB。

(3) Executor最大任务执行时间差

Executor最大任务执行时间差即同一任务在多个Executors上执行，最早完成和最晚完成的执行节点的用时差。如果执行节点最大时间差超过120 s即可认定为此次计算发生了数据倾斜，实验统计结果如图14所示。

从图14看到MGMA仅在第4次计算中发生较为明显的数据倾斜现象，相较于其他对照组优势明显。DBPM算法同样取得了不错的优化效果，只在第3次和第5次计算中出现了较为明显的数据倾斜现象。传统预分组方案只在第2次中未出现明显数据倾斜现象，而S-Shuffle算法随着计算复杂度的提升，在5次计算中均发生了数据倾斜。

4.8 实验总结

以NO为基线，多关系指标计算统计分析实验的统计结果见表2。

S-Shuffle内存占用更高的原因：该算法在每次计算时需提取特定字段并执行RDD转换操作。例如，在统计除了营业部关系外的6组关系指标时，由于不同关系对应字段的差异性，S-Shuffle需额外执

行至少6次RDD转换，导致其平均内存占用（114%）显著高于其他算法。

NO发生数据倾斜率100%的合理性：实验每组执行5次计算，在复杂计算场景下数据倾斜概率已达100%，这与实际生产环境中“稍复杂任务即频繁发生倾斜”的观测结果一致。

在此背景下，本研究提出的基于关系的MGMA展现出显著优势。

（1）数据倾斜控制：通过动态平衡不同组间数据分布，将倾斜概率从NO组的100%降至20%，有效解决了传统算法（如National组80%倾斜率）的分布不均问题。

（2）内存占用优化：内存占用率（72%）较S-Shuffle（114%）降低37%，在降低Shuffle的同时避免了冗余RDD操作带来的资源浪费。

（3）计算效率提升：通过流水线优化和并行化处理，计算用时仅为NO的61%，在数据倾斜高发场景下仍保持稳定性能。

综上所述，与S-Shuffle（侧重减少Shuffle数据量而非解决倾斜）和DBPM（侧重中间数据分区）相比，MGMA在计算效率、计算稳定性方面表现更优。实验结果验证了该算法在大规模券商数据场景中的有效性，为未来类似场景下的数据处理和分析提供了有力的技术支持。

5 结束语

本文通过对券商真实用户数据进行分析，针对其中天分区存储和复杂用户关系导致的计算时Shuffle和数据倾斜问题，提出了一种新的多分组归并算法（MGMA）。MGMA通过有效分组和优化

表2 多关系指标计算统计分析实验的结果统计

采用方法	数据倾斜概率	内存占用	计算用时
NO	100%	100%	100%
MGMA	20%	72%	61%
National	80%	89%	92%
S-Shuffle	100%	114%	86%
DBPM	40%	102%	83%

处理策略，降低了计算资源消耗，提高了计算效率。具体步骤包括：首先，用户关系分解，提取出计算相关的所有关系；然后，初始化分组，保证分组的最小粒度，从而保证后续数据分组的均衡化；接下来，多关系分组合并，通过算法实现在同一分组内包含多类关系的数据；最后，散组合并，进一步减少数据分散度的同时，让各组分内的数据量更加的均衡。理论分析和实证测试表明，多分组归并算法在数据倾斜控制、内存占用优化和计算效率提升方面都有显著的优势。

总之，本研究对金融券商领域的大数据处理问题进行了有效探索和优化。多分组归并算法的提出不仅为金融用户数据处理提供了新的方法论，还为相关数据密集型行业的数据处理问题提供了新的视角。当然，该算法仍然存在一些不足，例如，在多分组归并过程中，如果存在某一关系将大多数用户数据覆盖，可能会让大多数数据归并到同一分组下，从而导致分组均衡失效。未来的研究可进一步探索该问题的优化和改进，以应对不断增长的数据规模和复杂性，从而进一步提高证券服务的效率和质量。

参考文献:

- [1] 罗圣美, 戚晨, 王敏, 等. 大数据平台在金融行业的典型应用[J]. 大数据, 2018, 4(2): 107-114.
- LUO S M, QI C, WANG M, et al. Typical application in financial industry of big data platform[J]. Big Data Research, 2018, 4(2): 107-114.
- [2] 张维, 林毓, 康俊卿, 等. 计算实验金融工程: 大数据驱动的金融管理决策工具[J]. 管理世界, 2023, 39(5): 173-190.
- ZHANG W, LIN S, KANG J Q, et al. Agent-based computational financial engineering: a financial management decision-making tool driven by big data[J]. Journal of Management World, 2023, 39(5): 173-190.
- [3] 李金金, 雷凌. 基于大数据的智能投后管理机制研究[J]. 企业科技与发展, 2023(6): 122-125.
- LI J J, LEI L. Research on intelligent post-investment management mechanism based on big data[J]. Sci-Tech & Development of Enterprise, 2023(6): 122-125.
- [4] XIE H X, SHI Y F. A big data technique for Internet financial risk control[J]. Mobile Information Systems, 2022, 2022(1): 9549868.
- [5] CHEN L C, JIANG R. Internet financial risk model evaluation and control decision based on big data[J]. Wireless Communications and Mobile Computing, 2022, 2022(1): 8606624.
- [6] 兴业证券股份有限公司课题组. 大数据画像技术在证券公司的研究与应用[C]//中国证券业2018年论文集. 2019: 518-547.
- Industrial Securities Co., Ltd. Research Group. Research and application of big data profiling technology in securities companies[C]//Proceedings of China Securities Industry 2018. 2019: 518-547.
- [7] 侯伟凡, 樊玮, 张宇翔. 改进的Spark Shuffle内存分配算法[J]. 计算机应用, 2017, 37(12): 3401-3405, 3429.
- HOU W F, FAN W, ZHANG Y X. Improved Spark Shuffle memory allocation algorithm[J]. Journal of Computer Applications, 2017, 37(12): 3401-3405, 3429.
- [8] 吴信东, 嵇圣础. MapReduce与Spark用于大数据分析之比较[J]. 软件学报, 2018, 29(6): 1770-1791.
- WU X D, JI S W. Comparative study on MapReduce and Spark for big data analytics[J]. Journal of Software, 2018, 29(6): 1770-1791.
- [9] TANG D X, LIU T Y, LEE R B, et al. A case study of optimizing big data analytical stacks using structured data shuffling[C]//Proceedings of the 2016 IEEE International Congress on Big Data (BigData Congress). Piscataway: IEEE Press, 2016: 91-100.
- [10] 潘锋烽, 熊劲. NV-Shuffle: 基于非易失内存的Shuffle机制[J]. 计算机研究与发展, 2018, 55(2): 229-245.
- PAN F F, XIONG J. NV-Shuffle: Shuffle based on non-volatile memory[J]. Journal of Computer Research and Development, 2018, 55(2): 229-245.
- [11] YAO X, WANG C L, ZHANG M Z. EC-Shuffle: dynamic erasure coding optimization for efficient and reliable Shuffle in Spark[C]//Proceedings of the 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). Piscataway: IEEE Press, 2019: 41-51.

- [12] 陈英芝. Spark Shuffle 的内存调度算法分析及优化[D]. 杭州: 浙江大学, 2016.
- CHEN Y Z. Analysis and optimization of memory scheduling algorithm for Spark Shuffle[D]. Hangzhou: Zhejiang University, 2016.
- [13] RAJ S, RAMESH D, SETHI K K. A Spark-based Apriori algorithm with reduced shuffle overhead[J]. The Journal of Supercomputing, 2021, 77(1): 133-151.
- [14] GU R, HUANG X, DAI H P, et al. Efficient, scalable and robust data Shuffle service for distributed MapReduce computing on cloud[C]//Proceedings of the 2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys). Piscataway: IEEE Press, 2022: 337-346.
- [15] 熊安萍, 夏玉冲, 杨方方. 一种 Spark 集群下的 Shuffle 优化机制[J]. 计算机工程与应用, 2018, 54(4): 72-76.
- XIONG A P, XIA Y C, YANG F F. Shuffle optimization for Spark cluster[J]. Computer Engineering and Applications, 2018, 54(4): 72-76.
- [16] LI C L, CAI Q Q, LUO Y L. Data balancing-based intermediate data partitioning and check point-based cache recovery in Spark environment[J]. The Journal of Supercomputing, 2022, 78(3): 3561-3604.
- [17] LI H J, ZHU L S, WANG S C, et al. Cost-Aware scheduling and data skew alleviation for big data processing in heterogeneous cloud environment[J]. Journal of Grid Computing, 2023, 21(3): 33.
- [18] YAN Y F, WANG Z L, QIU X S, et al. A shuffle partition optimization scheme based on data skew model in Spark[J]. Journal of Beijing University of Posts and Telecommunications, 2020, 43(2): 116-121.
- [19] WANG Y Y, ZHOU T L. A lowest cost RDD caching strategy for spark[C]//Proceedings of the 2019 4th International Conference on Automatic Control and Mechatronic Engineering (ACME). Beijing: CSP, 2019: 30-36.
- [20] REENA M J, NARGUN A S. Preprocessing big data for efficient storage and research[J]. International Journal of Recent Technology and Engineering, 2019, 8(2S3): 11-16.
- [21] KANTAPALLI B, MARKAPUDI B R. SSPO-DQN Spark: shuffled student psychology optimization based deep Q network with spark architecture for big data classification[J]. Wireless Networks, 2023, 29(1): 369-385.
- [22] 周睿敏, 张文秀. 金融科技创新风险及控制探析: 基于大数据、人工智能、区块链的研究[J]. 中国管理信息化, 2017, 20(19): 33-36.
- ZHOU R M, ZHANG W X. Analysis on the risk and control of financial technology innovation: based on the research of big data, artificial intelligence and blockchain[J]. China Management Informationization, 2017, 20(19): 33-36.
- [23] 卓莉莉. M 券商基于客户特征的大数据精准营销策略研究[D]. 上海: 上海财经大学, 2020.
- ZHUO L L. Research on big data precision marketing strategy based on customer characteristics in M securities company[D]. Shanghai: Shanghai University of Finance and Economics, 2020.

作者简介



曹亚坤（1990-），男，长沙理工大学计算机与通信工程学院硕士生，主要研究方向为大数据、并行计算。



唐小勇（1973-），男，博士，长沙理工大学计算机与通信工程学院教授，主要研究方向为云计算、并行分布式计算、大数据。

收稿日期: 2024-10-29