分布式的增量式张量 Tucker 分解方法

杨克宇[1)] 高云君[1),2)] 陈 璐[1)] 葛丛丛[1)] 沈怡峰[1)]

[1)](浙江大学计算机科学与技术学院 杭州 310027)

[2)](阿里巴巴-浙江大学前沿技术联合研究中心 杭州 311121)

摘 要 随着社交网络、电商系统、移动终端设备的快速发展,海量且高维的数据正以前所未有的速度不断地增长和积累.高维数据可以自然地表示为张量.张量的 Tucker 分解方法是一种常用且经典的高维数据分析机器学习方法,被广泛地应用于推荐系统、图像压缩、计算机视觉等多个领域.然而,传统的张量分解方法大多只能处理静态的数据,并不适用于动态增长的数据.当处理不断增长的数据时,传统方法大多只能低效地重新开始计算,以完成张量分解.针对增量式数据对传统张量分解方法带来的挑战,本文提出了一种分布式的增量式张量 Tucker 分解方法 DITTD,首次解决了海量高维且动态增长数据上高效的分布式张量 Tucker 分解问题.该方法首先根据增量数据相对原始数据的位置关系对其进行分类处理.为了实现分布式节点的负载均衡,本文指出张量的最优划分是 NP-难问题,并使用启发式方法以实现尽可能均匀的张量划分.为了避免张量 Tucker 分解的中间结果爆炸问题,本文提出了一种新颖的增量式张量 Tucker 分解计算方法.该方法减少了中间结果的计算和网络传输通信量,以提升分布式的增量式张量 Tucker 分解效率.最后,本文在真实与合成数据集上进行了大量的实验.实验结果验证了本文方法的运行效率比基准方法提升了至少 1 个数量级,并具有良好的可扩展性.

关键词 张量;Tucker 分解;分布式;增量式;Spark

中图法分类号 TP18 **DOI 号** 10.11897/SP.J.1016.2021.01696

Distributed Incremental Tensor Tucker Decomposition

YANG Ke-Yu[1)] GAO Yun-Jun[1),2)] CHEN Lu[1)] GE Cong-Cong[1)] SHEN Yi-Feng[1)]

[1)](*College of Computer Science, Zhejiang University, Hangzhou* 310027)

[2)](*Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Hangzhou* 311121)

Abstract With the rapid development of social networks, e-commerce systems, and mobile terminal devices, massive and high-dimensional data is overwhelmingly increasing. A natural representation of high dimensional data is called tensor. Tensor Tucker decomposition is a fundamental machine learning method for multi-dimensional data analysis. Tensor Tucker decomposition aims at discovering the latent representations for the given tensor. It is widely used in many applications, such as recommendation systems, image compression, computer vision, to name but a few. Nevertheless, most traditional tensor decomposition methods could only handle static data, and are not suitable for dynamic data. Because those traditional methods could only re-compute the whole tensor decomposition from scratch whenever data grows. Besides, several incremental tensor Tucker decomposition methods focus on one-mode incremental tensor. However, the tensor in real life could be developed in multiple modes. To our knowledge, there is only one method suitable for multi-mode incremental tensor Tucker decomposition. Nevertheless, all the existing incremental tensor Tucker decomposition methods are designed for the standalone machine,

收稿日期:2020-11-12;在线发布日期:2021-04-04.本课题得到国家重点研发计划项目(2018YFB1004003)、国家自然科学基金(62025206,61972338)资助.**杨克宇**,博士,主要研究方向为 DB 驱动的 AI、数据库.E-mail:kyyang@zju.edu.cn.**高云君**(通信作者),博士,教授,国家杰出青年科学基金入选者,中国计算机学会(CCF)高级会员,主要研究领域为数据库、大数据管理与分析、DB 与 AI 融合.E-mail:gaoyj@zju.edu.cn.**陈 璐**,博士,研究员,主要研究领域为数据库、大数据管理与分析.**葛丛丛**,博士研究生,主要研究方向为数据整合、数据质量.**沈怡峰**,硕士,主要研究方向为分布式数据处理.

and thus, they cannot handle large-scale dynamic incremental data efficiently. Considering the continuous expansion nature of data, it requires an efficient distributed incremental tensor Tucker decomposition method. Thus, this paper proposes a Distributed Incremental Tensor Tucker Decomposition method (DITTD for short), which is the first attempt to tackle this problem. Towards this, there are two main challenges. The first one is to achieve the load balancing among the workers in the distributed environment. DITTD firstly divides the incremental tensor based on its positional relationship with the previous one. Then, DITTD tries to generate the optimal partitioning result for the incremental tensor, such that the number of non-zero elements in each tensor partition is equal. However, the optimal tensor partitioning problem is proved to be NP-hard. Thus, DITTD utilizes two heuristic tensor partitioning methods to partition the incremental tensor as well as possible. One is the Greedy tensor Partitioning algorithm (GP for short), which greedily assigns partitioning boundaries and makes the number of non-zero elements in each tensor partition to be close to the optimal sum target. The other is the Max-min Matching tensor Partitioning algorithm (M2P), which iteratively assigns the tensor slice with the maximum number of non-zero elements into the tensor partition with the minimum number of non-zero elements. After the tensor partitioning, DITTD meets the second challenge, which is to compute the incremental tensor Tucker decomposition efficiently. DITTD provides a novel incremental Tucker decomposition computation method to avoid the explosion of intermediate data of Tucker decomposition. This method provides an equivalent conversion for the update rule of factor matrices and designs a row-wise computation strategy for the distributed Tucker decomposition. Based on the above-mentioned techniques, DITTD can reduce the computation of intermediate data and the network communication among the workers, and thus, DITTD improves the efficiency of the distributed Tucker decomposition for incremental tensor. Last but not the least, comprehensive experiments are conducted on both real and synthetic data sets. The experimental results show that our proposed method DITTD is at least 10× faster than the baseline method and scale well.

Keywords　tensor; Tucker decomposition; distributed; incremental; Spark

1　引　言

现实世界的诸多应用场景不断涌现出海量、高维的数据. 张量(Tensor)是高维数据的自然表达形式. 张量分解(Tensor Decomposition)是一种重要的高维数据分析机器学习方法,其旨在挖掘高维数据的隐含表示(主成分),已得到了学术界和工业界的广泛关注[1-2]. 张量分解的应用包括但不限于推荐系统[3-4]、图像压缩[5-6]以及计算机视觉[7-8]. 下面以推荐系统为例说明张量分解的应用.

推荐系统应用:推荐系统通过分析用户过往的行为数据及其相似用户的行为来得到用户的偏好,并以此为其提供精准的推荐. 在此,本文以用户评价场景为例,介绍张量分解方法在推荐系统中的作用. 在用户评价场景中,一个特定用户 u 对一个特定商品 i 在时刻 t 的评分数据 r 可以被表示为一个四元组,即 (u,i,t,r). 一个三维的张量(即"用户×商品×时间"三维评分数据张量)可以用来表示用户过往商品评价数据的四元组. 进一步地,可以将用户未显式做出评价的商品看作该张量的缺失元素. 而后,张量分解方法可以被用于对用户过往评价数据进行分解、得到隐含的数据表示信息,进而补全前述的缺失数据,即完成对用户商品评价的预测. 基于上述过程,可以完成基于张量分解的用户商品推荐.

当前,现实生活的众多应用无时无刻不在产生大量的增量式数据. 根据报告统计①,截止 2020 年,全球每天平均新增超过 2.5×10^{18} 字节的数据. 例如,在电商领域,阿里巴巴集团仅淘宝网一天即产生

① https://www.domo.com/solution/data-never-sleeps-6

超过 700 GB 的数据①;在社交网络领域,新浪微博日均文字及图片发布量均超过亿级②.上述动态增长的海量数据使得利用数据库领域的数据处理与优化技术(如分布式并行技术等)加速机器学习方法成为一种趋势[9-12].其中,分布式增量学习旨在基于已学习到的知识,利用分布式架构增量式地从海量新增数据中快速学习,以得到新的知识.就张量分解方法而言,从头重新全量式地计算增量式张量的分解结果显然是低效的[13-14].传统的张量分解方法[15-18]大多数关注静态数据场景,即它们仅适用于全量式的张量分解,而不能高效地支持动态增长张量数据的分解处理.

为了应对数据动态增长的挑战,目前,一些工作研究了增量式的张量分解方法[19-20].然而,它们一开始只讨论了张量仅在一个维度(张量的模)上增长的问题.实际上,现实数据可能会在任意模上增长.图 1 给出了两种张量增长模式示例,从中可以看出两种张量增长模式的不同:左侧表示张量仅在单个模上增长的情况,可以看到张量随着时间只在竖直方向上增长;而右侧表示张量在多模上增长的情况,可以看到张量随着时间在竖直、水平、前后方向上都有增长.显然,张量多模增长的模式在现实应用场景中更为普遍的存在.譬如,在前述的推荐系统应用例子中,新用户、新商品以及新评分数据均可能随时间演变而加入至系统.

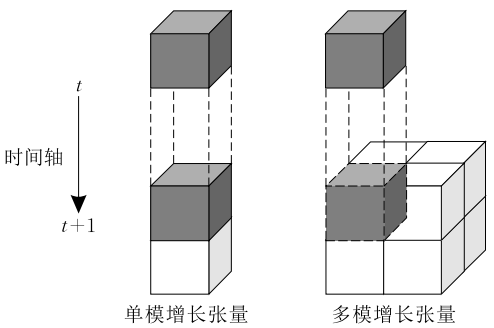


图 1 两种张量增长模式示例

因此,研究学者们逐渐开始考虑在所有维度上都存在数据增长的增量式张量分解问题[14,21].目前,面向多模增长的增量式张量分解研究正处于起步阶段,它们初步解决了单机环境下多模增长的张量分解问题.然而,因其可扩展性有限,不能很好地处理海量新增数据的分析需求.当前,诸多现实应用中涌现出不断增长的海量数据驱使着分布式的增量式张量 Tucker 分解方法研究.

本文提出了一种高效的分布式的增量式张量 Tucker 分解方法 DITTD,首次借助分布式框架处理海量新增数据的张量分解计算分析需求.为了高效地实现增量式张量的分布式 Tucker 分解方法,主要需要解决以下两大挑战:

(1)如何保证分布式框架下各个工作节点的负载均衡?工作节点的负载均衡是设计高效分布式算法的关键.为了避免工作节点的负载倾斜严重而导致性能瓶颈问题,需要精心地分配各个工作节点的工作负载,保证其负载均衡.通过分析张量分解的计算过程可以发现:张量分解的计算量与张量中非零元素的数量成正相关关系.因此,通过均匀地划分张量非零元素,使得张量划分结果的各个部分中包含的张量非零元素的数量保持一致,即可实现负载均衡.然而,实现上述张量的最优划分是一个 NP-难问题.为此,本文探讨了启发式的张量划分算法,利用贪心和最大-最小匹配策略,尽可能地实现张量非零元素的均匀划分.

(2)如何设计高效的分布式张量 Tucker 分解计算算法?在完成了增量式张量的划分之后,各个工作节点需要分布式地进行张量 Tucker 分解计算.在张量 Tucker 分解计算过程中,容易遇到中间结果爆炸的问题[14,18].尤其在本文讨论的分布式环境下,控制中间结果的大小有利于减少计算量以及节点间的网络通信量,是设计高效的 Tucker 分解计算算法的关键.本文通过分析增量式张量 Tucker 分解计算流程,发现了前人工作中的计算瓶颈问题,并对其进行优化,从而减少了中间结果的大小.进一步地,结合分布式框架设计了新颖的算法,以支持高效的增量式张量 Tucker 分解计算.

本文工作的主要贡献可以总结为以下四点:

(1)提出了一种分布式的增量式张量 Tucker 分解方法 DITTD.该方法首次借助分布式框架高效地解决了海量新增数据的张量 Tucker 分解问题.

(2)分析了分布式的增量式张量 Tucker 分解工作负载,使用基于贪心和最大-最小分配策略的启发式张量划分算法,保证了各个节点的负载均衡.

(3)探讨了增量式张量 Tucker 分解的计算瓶颈、优化中间结果大小,并基于此设计了分布式的增量式张量 Tucker 分解计算方法.

(4)在真实与合成数据集上进行了大量的实验评估,验证了 DITTD 相比基准方法有至少 1 个数量级的效率提升,且具有良好的可扩展性.

① http://dzsws.mofcom.gov.cn/anli/detal_21.html
② https://data.weibo.com/report/reportDetail?id=433

接下来，本文第 2 节阐述张量分解的相关工作；第 3 节介绍增量式张量 Tucker 分解相关的基本概念；第 4 节详细阐述分布式的增量式张量 Tucker 分解方法 DITTD；第 5 节通过详尽的实验评估分析 DITTD 的性能；第 6 节总结全文.

2　相关工作

本节介绍张量分解的相关工作. 第 2.1 节回顾全量式张量分解的相关工作，包括单机、并行以及分布式环境下的张量分解方法；第 2.2 节介绍增量式张量分解的相关工作，主要分为单模增长和多模增长的张量分解两类研究工作.

2.1　全量式张量分解

张量分解作为重要的高维数据分析方法，主要有两种常用的算法：CP(CANDECOMP/PARAFAC)分解和 Tucker 分解. 张量 CP 分解最初由 Hitchcock[22]、Carroll 和 Chang[23] 以及 Harshman[24] 分别独立提出. CP 分解的基本思想是将给定的张量分解为多个因子矩阵的外积形式. 张量 Tucker 分解最初由 Tucker[25] 提出，而后 Kroonenberg 和 De Leeuw[26] 以及 De Lathauwer 等人[27] 对其做了进一步改进. Tucker 分解与 CP 分解不同，它将给定的张量分解为一个核心张量(Core Tensor)与多个因子矩阵的乘积. 实际上，CP 分解可以被看作 Tucker 分解在核心张量为超对角结构时的一个特例. 在应用场景中，CP 分解比 Tucker 分解更适用于具备低秩多重线性结构性质的张量分解，而 Tucker 分解适用于一般(非低秩)的张量分解[2]. 现有的张量分解算法按优化计算方式来区分，可以被分为以下三大类：(1) 交替最小二乘法(Alternating Least Square, ALS)[28-29]；(2) 随机梯度下降(Stochastic Gradient Descent, SGD)[30-31]；(3) 坐标下降法(Coordinate Descent, CDD)[32-33].

近年来，现实应用中产生的海量数据驱使着张量分解研究从单机环境向并行、分布式环境发展，以支持海量数据的高效计算需求. SPLATT[34-35] 为稀疏张量 CP 分解提供了高效的并行计算工具. 另外，P-TUCKER[18]、TuckerMPI[36] 以及 S-HOT[37] 为 Tucker 分解设计了高效的并行计算算法. 在分布式框架下，GigaTensor[38] 提供一种大规模张量分解算法，SCouT[39] 探索了伴随辅助信息的大规模张量分解问题. 上述两种张量分解方法被整合进了一个张量数据挖掘算法库 BIGtensor[40]. 此外，CartHP[41] 提出了一种基于超图的张量划分方法以提升分布式张量分解效率. DisTenC[17] 研究了基于 Spark 的分布式张量补全算法.

值得指出的是：上述所有的工作都是面向全量式的张量分解进行设计，并不能高效支持增量式的张量分解计算.

2.2　增量式张量分解

日益增长的海量数据驱使着增量式张量分解研究的发展. 最初，增量式张量分解的工作只解决了单模增长的张量分解问题. 在增量式张量 CP 分解方面，Nion 和 Sidiropoulos[42] 提出了两种自适应的 PARAFAC 算法，包括基于协同对角化的 PARAFAC-SDT 和基于加权最小二乘法的 PARAFAC-RLST. Phan 和 Cichocki[43] 探讨了一种可以被用于增量式张量分解的分块张量计算思想. Zhou 等人[20] 给出了增量式张量 CP 分解方法 OnlineCP. 在增量式张量的 Tucker 分解问题上，Sun 等人[19] 开发了增量式张量分析框架(ITA)以支持流式张量分析. Yu 等人[44] 探索了一种在线低秩张量学习算法(ALTO)以解决流式张量 Tucker 分解问题.

最近，多模增长的增量式张量分解问题开始被研究学者所关注. Song 等人[21] 给出了多模增长的张量分解问题形式化定义，并针对此问题提出了一种增量式张量 CP 分解方法 MAST. 基于此，Nimishakavi 等人[45] 探索了附带额外信息的多模增长张量 CP 分解问题. Yang 等人[46] 进一步提出了分布式的多模增长张量 CP 分解方法 DisMASTD. 另外，Xiao 等人[14] 设计了 eOTD 算法，首次研究了多模增长张量 Tucker 分解问题，以在保证结果准确度的同时，提升了单机上的计算效率.

综上所述，当前增量式张量 Tucker 分解方法大多只适用于单模增长模式，而唯一解决多模增长张量 Tucker 分解问题的 eOTD 算法只适用于单机环境. 然而，许多的现实应用会动态产生海量数据，亟待探索分布式的增量式张量 Tucker 分解方法，以支持海量新增数据的张量分解计算需求. 为此，本文提出了一种高效的分布式的增量式张量 Tucker 分解方法 DITTD.

3　基本概念

本节介绍张量分解的相关基本概念. 首先，第 3.1 节介绍张量及其相关的基本操作；而后，第 3.2 节介绍多模增长的张量 Tucker 分解.

3.1 张量及其相关基本操作

本小节介绍张量的概念以及与本文相关的张量基本操作,更多张量操作的细节可参见综述[1-2].

定义 1. 张量(Tensor).一个张量是一个多维数组,记为$\mathcal{X}$.

一般地,张量的维度数量被称为阶(Oder),也被称为模(Mode).例如,向量(Vector)是一阶张量,矩阵(Matrix)是二阶张量.在本文中,向量(1 阶张量)使用加粗的小写字母表示(如 $\boldsymbol{a}$)、矩阵(2 阶张量)使用加粗的大写字母表示(如 $\boldsymbol{A}$)、高阶张量(阶数大于等于 3)使用欧拉手写字母表示(如$\mathcal{X}$).向量 $\boldsymbol{a}$ 的第 i 个元素表示为 $\boldsymbol{a}(i)$,矩阵 $\boldsymbol{A}$ 的(i,j)位置上的元素表示为 $\boldsymbol{A}(i,j)$,三阶张量$\mathcal{X}$ 的(i,j,k)位置上的元素表示为$\mathcal{X}(i,j,k)$.本文使用小写字母作为元素位置标记时,取值范围为 1 到其对应的大写字母,譬如 $i=1,2,\cdots,I$.

定义 2. 张量矩阵化(Tensor Matricization).给定阶张量$\mathcal{X}\in\mathbb{R}^{I_1\times I_2\times\cdots\times I_N}$,$I_1,I_2,\cdots,I_N\in N^*$表示各模大小.张量$\mathcal{X}$ 在第 n 模上的矩阵化记为 $\boldsymbol{X}_{(n)}$.特别地,张量$(i_1,i_2,\cdots,i_N)$位置上的元素$\mathcal{X}(i_1,i_2,\cdots,i_N)$被映射至相应矩阵的$(i_n,j)$位置,即 $\boldsymbol{X}_{(n)}(i_n,j)$.

其中,$j=1+\sum_{k=1,k\neq n}^{N}(i_k-1)J_k$,$J_k=\prod_{m=1,m\neq n}^{k-1}I_m$.

定义 3. 张量矩阵乘法(Tensor Matrix Multiplication).给定 N 阶张量$\mathcal{X}\in\mathbb{R}^{I_1\times I_2\times\cdots\times I_N}$和矩阵$\boldsymbol{U}\in\mathbb{R}^{J\times I_n}$,$J=\prod_{m=1,m\neq n}^{N}I_m$,可以定义张量$\mathcal{X}$ 在第 n 模上的张量矩阵乘法,记为$\mathcal{Y}=\mathcal{X}\times_n\boldsymbol{U}$,$\mathcal{Y}\in\mathbb{R}^{I_1\times\cdots\times I_{n-1}\times J\times I_{n+1}\times\cdots\times I_N}$.特别地,

$$\mathcal{Y}(i_1,\cdots,i_{n-1},j,i_{n+1},\cdots,i_N)=\sum_{i_n=1}^{I_n}\mathcal{X}(i_1,i_2,\cdots,i_N)\boldsymbol{U}(j,i_n).$$

张量-矩阵乘法常被表示为矩阵化形式,即

$$\mathcal{Y}=\mathcal{X}\times_n\boldsymbol{U}\Leftrightarrow\boldsymbol{Y}_{(n)}=\boldsymbol{U}\boldsymbol{X}_{(n)}.$$

定义 4. 矩阵克罗内克积(Matrix Kronecker Product).给定两个矩阵$\boldsymbol{A}\in\mathbb{R}^{I\times J}$和$\boldsymbol{B}\in\mathbb{R}^{K\times L}$,二者的克罗内克积记为 $\boldsymbol{A}\otimes\boldsymbol{B}$,$\boldsymbol{A}\otimes\boldsymbol{B}$ 是一个$(IK)\times(JL)$的矩阵,定义如下:

$$\boldsymbol{A}\otimes\boldsymbol{B}=\begin{bmatrix}\boldsymbol{A}(1,1)\boldsymbol{B} & \boldsymbol{A}(1,2)\boldsymbol{B} & \cdots & \boldsymbol{A}(1,J)\boldsymbol{B}\\ \boldsymbol{A}(2,1)\boldsymbol{B} & \boldsymbol{A}(2,2)\boldsymbol{B} & \cdots & \boldsymbol{A}(2,J)\boldsymbol{B}\\ \vdots & \vdots & \ddots & \vdots\\ \boldsymbol{A}(I,1)\boldsymbol{B} & \boldsymbol{A}(I,2)\boldsymbol{B} & \cdots & \boldsymbol{A}(I,J)\boldsymbol{B}\end{bmatrix}.$$

3.2 多模增长的张量 Tucker 分解算法

本小节先给出 Tucker 分解的定义,而后阐述单机环境下多模增长的张量 Tucker 分解算法 eOTD[14]的流程.

定义 5. Tucker 分解(Tucker Decomposition).给定 N 阶张量$\mathcal{X}\in\mathbb{R}^{I_1\times I_2\times\cdots\times I_N}$,Tucker 分解将$\mathcal{X}$ 分解为 1 个核心张量(Core Tensor)和 N 个因子矩阵乘积形式:

$$\mathcal{X}\approx\mathcal{G}\times_1\boldsymbol{A}_{(1)}\times_2\boldsymbol{A}_{(2)}\cdots\times_N\boldsymbol{A}_{(N)}=\mathcal{G}\times\{\boldsymbol{A}_{(n)}\}.$$

其中,核心张量$\mathcal{G}\in\mathbb{R}^{R_1\times R_2\times\cdots\times R_N}$,因子矩阵 $\boldsymbol{A}_{(n)}\in\mathbb{R}^{I_n\times R_n}$,$n=1,2,\cdots,N$.

一般地,可以将核心张量$\mathcal{G}$ 看作是原始张量$\mathcal{X}$ 的压缩($R_n<I_n$),因子矩阵 $\boldsymbol{A}^{(n)}$ 为酉矩阵(Unitary Matrix).张量的 Tucker 分解还可以使用张量矩阵化形式表示,即 $\boldsymbol{X}_{(n)}=\boldsymbol{A}_{(n)}\boldsymbol{G}_{(n)}(\boldsymbol{A}_{(N)}\otimes\cdots\boldsymbol{A}_{(n+1)}\otimes\boldsymbol{A}_{(n-1)}\cdots\otimes\boldsymbol{A}_{(1)})^{\mathrm{T}}$.例如,三阶张量$\mathcal{X}\in\mathbb{R}^{I\times J\times K}$的 Tucker 分解可以表示为$\mathcal{X}\approx\mathcal{G}\times_1\boldsymbol{A}\times_2\boldsymbol{B}\times_3\boldsymbol{C}$,其矩阵化形式为

$$\boldsymbol{X}_{(1)}\approx\boldsymbol{A}\boldsymbol{G}_{(1)}(\boldsymbol{C}\otimes\boldsymbol{B})^{\mathrm{T}},$$
$$\boldsymbol{X}_{(2)}\approx\boldsymbol{B}\boldsymbol{G}_{(2)}(\boldsymbol{C}\otimes\boldsymbol{A})^{\mathrm{T}},$$
$$\boldsymbol{X}_{(3)}\approx\boldsymbol{C}\boldsymbol{G}_{(3)}(\boldsymbol{B}\otimes\boldsymbol{A})^{\mathrm{T}}.$$

其中,$\mathcal{G}\in\mathbb{R}^{P\times Q\times R}$,因子矩阵 $\boldsymbol{A}\in\mathbb{R}^{I\times P}$,$\boldsymbol{B}\in\mathbb{R}^{J\times Q}$,$\boldsymbol{C}\in\mathbb{R}^{K\times R}$.

如前所述,在许多现实应用中,张量数据随时间在所有模上增长.下面给出多模增长的 Tucker 分解的定义.

定义 6. 多模增长的 Tucker 分解.假定有一个张量数据流$\{\mathcal{X}^{(t)}\in\mathbb{R}^{I_1^{(t)}\times I_2^{(t)}\times\cdots\times I_N^{(t)}}\}$及其对应 t 时刻张量的 Tucker 分解结果$\mathcal{X}^{(t)}=\mathcal{G}^{(t)}\times\{\boldsymbol{A}_{(n)}^{(t)}\}$,其中核心张量 $\mathcal{G}^{(t)}\in\mathbb{R}^{R_1\times R_2\times\cdots\times R_N}$,因子矩阵 $\boldsymbol{A}_{(n)}^{(t)}\in\mathbb{R}^{I_n\times R_n}$,$n=1,2,\cdots,N$.在 $t+1$ 时刻的张量为$\mathcal{X}^{(t+1)}\in\mathbb{R}^{I_1^{(t+1)}\times I_2^{(t+1)}\times\cdots\times I_N^{(t+1)}}$,$\mathcal{X}^{(t+1)}$ 可以被看作$\{\mathcal{X}_{i_1i_2\cdots i_N}^{(t+1)}\}_{(i_1i_2\cdots i_N)\in\Theta}$,其中 Θ 是 N 项二元组$\Theta\triangleq\{0,1\}^N$,$\mathcal{X}_{0\cdots0}^{(t+1)}=\mathcal{X}^{(t)}$.给定 $t+1$ 时刻的增长张量$\{\mathcal{X}_{i_1i_2\cdots i_N}^{(t+1)}\}_{(i_1i_2\cdots i_N)\in\Theta\backslash(0\cdots0)}$以及 t 时刻的 Tucker 张量分解结果$\mathcal{G}^{(t)}$和$\{\boldsymbol{A}_{(n)}^{(t)}\}$,多模增长的 Tucker 分解给出新增长张量 $\mathcal{X}^{(t+1)}$ 的 Tucker 分解结果 $\mathcal{G}^{(t+1)}$和$\{\boldsymbol{A}_{(n)}^{(t+1)}\}$.

图 2 给出了三阶多模增长张量 Tucker 分解的示例.Xiao 等人[14]提出了 eOTD 算法以解决多模增长的 Tucker 分解问题,但仅适用于单机环境.接下来简单介绍 eOTD 算法,详细算法过程请参见文献[14].eOTD 算法主要包括两个步骤,(1)更新因子矩阵;(2)更新核心张量.

(1)更新因子矩阵.首先,根据增长张量

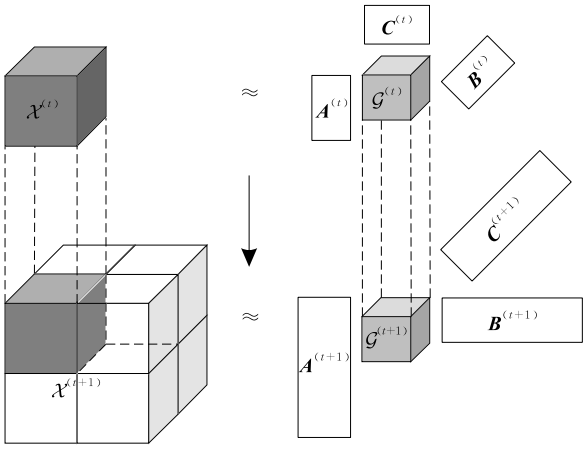


图 2　多模增长的张量 Tucker 分解示例

$\{\mathcal{X}^{(t+1)}_{i_1 i_2 \cdots i_N}\}_{(i_1 i_2 \cdots i_N)\in\Theta\setminus(0\cdots0)}$ 下标 $(i_1 i_2 \cdots i_N)$ 中含 1 的个数不同将其归为 N 类 $\{\mathbb{C}_m\}_{m=1}^{N}$. 然后，分别依次按 $\mathbb{C}_m$ 下标含 1 的个数顺序对相应的因子矩阵进行更新. 特别地，对于子张量类别 $\mathbb{C}_m$ 中的子张量 $\mathcal{X}^{(t+1)}_{i_1 i_2 \cdots i_N}$，若 $i_n=1$，则对应因子矩阵进行如下更新：

$$\boldsymbol{A}'^{\text{new}}_{(n)} \leftarrow \alpha \boldsymbol{A}'^{\text{old}}_{(n)} + (1-\alpha)(\mathcal{X}^{(t+1)}_{i_1 \cdots i_N})_{(n)} (\mathcal{G}_{-i_n})^{\dagger}_{(n)} \quad (1)$$

其中，系数 $\alpha\in(0,1)$ 是遗忘因子，表示多大程度上保留前一步得到的信息：$\mathcal{G}_{-i_n} \triangleq \mathcal{G}_{i_1 \cdots i_{n-1} i_{n+1} \cdots i_N} = \mathcal{G} \times_1 \boldsymbol{A}_{(1)} \cdots \times_{n-1} \boldsymbol{A}_{(n-1)} \times_{n+1} \boldsymbol{A}_{(n+1)} \cdots \times_N \boldsymbol{A}_{(N)}$. 若 $i_k=0$，则 $\boldsymbol{A}_{(k)} = \boldsymbol{A}^{(t)}_{(k)} \in \mathbb{R}^{I_k^{(t)} \times R_k}$，否则 $\boldsymbol{A}_{(k)} = \boldsymbol{A}'_{(k)} \in \mathbb{R}^{(I_k^{(t+1)} - I_k^{(t)}) \times R_k}$；符号 † 表示矩阵的伪逆（pseudo-inverse）. 部分更新的因子矩阵 $\{\boldsymbol{A}'_{(i_n)}\}$ 与上一时刻的因子矩阵 $\{\boldsymbol{A}^{(t)}_{(i_n)}\}$ 拼接得到矩阵 $\{(\boldsymbol{V}^{(t+1)}_{(n)})^{\mathrm{T}} \triangleq [(\boldsymbol{A}^{(t)}_{(i_n)})^{\mathrm{T}} (\boldsymbol{A}'_{(i_n)})^{\mathrm{T}}]\}$，$\boldsymbol{V}^{(t+1)}_{(n)} \in \mathbb{R}^{I_n^{(t+1)} \times R_n}$. 接着，在矩阵 $(\boldsymbol{V}^{(t+1)}_{(n)})^{\mathrm{T}}$ 上经过改进的格拉姆-施密特正交化过程（Modified Gram-Schmidt process，以下简称为 MGS）处理后得到 $t+1$ 时刻的因子矩阵 $\{\boldsymbol{A}^{(t+1)}_{(n)}\}$，$\boldsymbol{A}^{(t+1)}_{(n)} \in \mathbb{R}^{I_n^{(t+1)} \times R_n}$.

(2) 更新核心张量. 对于上述步骤得到的因子矩阵 $\{\boldsymbol{A}^{(t+1)}_{(n)}\}$，将其重新拆分为两个矩阵 $\{\boldsymbol{A}^{(t+1)}_{(n),0}\}$ 和 $\{\boldsymbol{A}^{(t+1)}_{(n),1}\}$，即 $(\boldsymbol{A}^{(t+1)}_{(n)})^{\mathrm{T}} = [(\boldsymbol{A}^{(t+1)}_{(n),0})^{\mathrm{T}} (\boldsymbol{A}^{(t+1)}_{(n),1})^{\mathrm{T}}]$，$\boldsymbol{A}^{(t+1)}_{(n),0} \in \mathbb{R}^{I_n^{(t+1)} \times R_n}$，$\boldsymbol{A}^{(t+1)}_{(n),0} \in \mathbb{R}^{I_n^{(t)} \times R_n}$ 以及 $\boldsymbol{A}^{(t+1)}_{(n),1} \in \mathbb{R}^{(I_n^{(t+1)} - I_n^{(t)}) \times R_n}$. 根据酉矩阵性质，核心张量可以按如下规则进行更新：

$$\mathcal{G}^{(t+1)} \leftarrow \mathcal{G}^{(t)} \times \{(\boldsymbol{A}^{(t+1)}_{(n),0})^{\mathrm{T}} \boldsymbol{A}^{(t)}_{(n)}\} + \sum_{(i_1 \cdots i_N)\in\Theta\setminus(0\cdots0)} \mathcal{X}^{(t+1)}_{i_1 \cdots i_N} \times \{(\boldsymbol{A}^{(t+1)}_{(n),i_n})^{\mathrm{T}}\} \quad (2)$$

其中，$\Theta \triangleq \{0,1\}^N$.

4　技术细节

本节详细阐述本文所提出的分布式的增量式张量 Tucker 分解方法 DITTD. DITTD 主要分为两个步骤：(1) 增量式张量划分；(2) 分布式 Tucker 分解计算. 为了叙述直观且方便，首先在第 4.1 节和第 4.2 节中，以三阶张量为例，分别给出上述 DITTD 两个步骤的技术细节；然后，在第 4.3 节中，阐述 DITTD 扩展至任意 N 阶增量式张量的 Tucker 分解方法.

具体来说，分布式的三阶增量式张量 Tucker 分解，假定 t 时刻的张量为 $\mathcal{X}^{(t)} \in \mathbb{R}^{I\times J\times K}$，$t+1$ 时刻的张量为 $\mathcal{X}^{(t+1)} \in \mathbb{R}^{(I+d_1)\times(J+d_2)\times(K+d_3)}$. 记 $\mathcal{X}^{(t+1)}_{000} = \mathcal{X}^{(t)}$，$\mathcal{X}^{(t+1)}$ 相对 $\mathcal{X}^{(t)}$ 的增长部分 $\{\mathcal{X}^{(t+1)}_{ijk}\}_{(ijk)\in\Theta\setminus(000)}$，$\Theta \triangleq \{0,1\}^3$ 为三项二元组. 给定 t 时刻张量的 Tucker 分解结果核心张量 $\mathcal{G}^{(t)} \in \mathbb{R}^{P\times Q\times R}$ 和因子矩阵 $\boldsymbol{A}^{(t)} \in \mathbb{R}^{I\times P}$，$\boldsymbol{B}^{(t)} \in \mathbb{R}^{J\times Q}$，$\boldsymbol{C}^{(t)} \in \mathbb{R}^{J\times R}$，DITTD 需要基于 t 时刻的 Tucker 分解结果完成分布式的增量式张量 Tucker 分解得到 $t+1$ 时刻张量 $\mathcal{X}^{(t+1)}$ 的 Tucker 分解结果，包括核心张量 $\mathcal{G}^{(t+1)} \in \mathbb{R}^{P\times Q\times R}$ 以及因子矩阵 $\boldsymbol{A}^{(t+1)} \in \mathbb{R}^{(I+d_1)\times P}$、$\boldsymbol{B}^{(t+1)} \in \mathbb{R}^{(J+d_2)\times Q}$ 以及 $\boldsymbol{C}^{(t+1)} \in \mathbb{R}^{(K+d_3)\times R}$.

4.1　增量式张量划分

新增长张量的数据量可能十分庞大，需要将其进行划分，而后分配至分布式系统的各个节点进行处理. 对增长的张量进行合适的划分，使得各个分布式工作节点的负载均衡是张量划分步骤中需要考虑的首要目标.

根据因子矩阵的更新式 (1) 和核心张量的更新式 (2)，可以分析得到以下两点：(1) 增量式张量 Tucker 分解过程中，应以增长张量的层为最小单位进行划分计算；(2) 增量式张量 Tucker 分解计算复杂度与增长张量中所含非零元素的数量成线性关系（第 4.2 节将对更新计算过程进行详细阐述）. 因此，一个直观的想法是保证各个划分中张量非零元素的数量相等. 然而，实现这个目标的算法是 NP-难的. 下面给出具体的讨论.

定理 1.　张量的最优划分问题是 NP-难问题.

证明.　此问题可以归约到一个 NP-完全问题. 划分问题（Partition problem）. 划分问题的描述如下：给定一个具有 n 个正整数的集合 $S=\{s_i \mid i=1,2,\cdots,n, s_i\in N^*\}$，判断是否能将 S 划分为两个子集 S_1 和 S_2，使得集合 S_1 中的数据和等于集合 S_2 中的数据和.

接下来，考虑张量划分中最简单的情况，假定只在一个模上对张量进行划分，且只需将其划分成为两个部分. 此时，可以将张量当前模上各层的非零元

素数量的集合看作 S，张量的最优划分问题等价于判断是否能将 S 划分为两个子集 S_1 和 S_2，以使得集合 S_1 和 S_2 中的数据和相等，即上述划分问题. 因此，张量的最优划分问题是 NP-难问题. 证毕.

考虑到张量的最优划分问题是 NP-难问题，本文使用启发式的张量划分算法，以尽可能地保证各个划分中张量非零元素的数量相等. 整个增量式划分过程可以分为两步：(1)增量式子张量分类；(2)子张量划分.

(1)增量式子张量分类. 首先，$t+1$ 时刻的增长张量 $\{\mathcal{X}_{ijk}^{(t+1)}\}_{(ijk)\in\Theta\backslash(000)}$ 根据其增长下标 (ijk) 中含 1 的个数不同被归为三类：

$$\mathbb{C}_1=\{\mathcal{X}_{100}^{(t+1)},\mathcal{X}_{010}^{(t+1)},\mathcal{X}_{001}^{(t+1)}\},$$

$$\mathbb{C}_2=\{\mathcal{X}_{011}^{(t+1)},\mathcal{X}_{101}^{(t+1)},\mathcal{X}_{110}^{(t+1)}\},$$

$$\mathbb{C}_3=\{\mathcal{X}_{111}^{(t+1)}\}.$$

其中，$\mathbb{C}_1$、$\mathbb{C}_2$、$\mathbb{C}_3$ 中子张量的增长下标 (ijk) 分别包含 1 个、2 个、3 个下标为 1 的模.

接下来，分别依次对三类子张量进行划分，而后进行分布式 Tucker 分解计算.

(2)子张量划分. 在完成子张量的分类后，DITTD 对各个子张量进行划分. 由因子矩阵更新式(1)可得，当子张量模的下标为 1 时，需要在此模上按层进行计算更新因子矩阵. 例如，对于子张量 $\mathcal{X}_{100}^{(t+1)}$，其需要在第 1 个模上按层进行计算更新因子矩阵 $\boldsymbol{A}'$. 因此，DITTD 应按增长下标对子张量在各对应模上进行划分.

一个容易想到的张量划分算法是基于贪心策略的张量划分算法(Greedy tensor Partitioning algorithm，以下简称为 GP). 具体来说，各个张量划分以张量划分中所含非零元素理论最优值(即若假定张量非零元素数量为 NNZ，划分数量为 p；则理论最优值为非零元素平均数 NNZ/p)为目标，依次将张量层分配至当前划分，直到当前划分非零元素数量达到理论最优值，完成当前划分的张量分配，继续下一个划分. 以此类推，GP 完成所有张量划分的分配.

然而，现实应用中的张量所含非零元素可能是不均匀分布的，即各个张量层所含的张量非零元素数量相差很大. 在这样情况下，GP 可能得到不均匀的张量划分.

本文进一步给出一种启发式张量划分算法. 该算法基于最大-最小分配策略，称为最大-最小匹配张量划分算法(Max-min Matching tensor Partitioning algorithm，以下简称为 M2P). M2P 的基本思想是：迭代地将当前张量中含非零元素数量最大的层划分至当前含非零元素数量最小的划分中，以尽可能地实现张量的均匀划分. M2P 算法的伪代码如算法 1 所示.

算法 1. 最大-最小匹配张量划分 M2P 算法.

输入：待划分子张量 $\mathcal{X}_{sub}$，待划分的模 n，划分数量 p_n

输出：张量划分结果 $\{P_p\}_{p=1}^{p_n}$

1. 统计子张量 $\mathcal{X}_{sub}$ 模 n 上各层非零元素 $\{NNZ_i\}_{i=1}^{d_n}$
2. $NHeap \leftarrow buildMaxHeap(\{NNZ_i\}_{i=1}^{d_n})$
3. 初始化划分 $\{P_p\}_{p=1}^{p_n}$ 为空
4. $PHeap \leftarrow buildMinHeap(\{P_p\}_{p=1}^{p_n})$
5. WHILE $NHeap \neq \varnothing$
6. $Slice = NHeap.pop()$ //非零元素数量最大层
7. $Part = PHeap.pop()$ //非零元素数量最小划分
8. $Part.add(Slice)$
9. $PHeap.push(Part)$
10. END WHILE
11. $\{P_p\}_{p=1}^{p_n} \leftarrow PHeap.getResult()$
12. 输出 $\{P_p\}_{p=1}^{p_n}$

给定待划分子张量 $\mathcal{X}_{sub}$、待划分的模 n 和划分数量 p_n 三个输入，M2P 算法输出张量划分结果 $\{P_p\}_{p=1}^{p_n}$. M2P 首先在给定的模 n 上统计 $\mathcal{X}_{sub}$ 各层非零元素的数量 $\{NNZ_i\}_{i=1}^{d_n}$(行 1). 其中，d_n 为 $\mathcal{X}_{sub}$ 在模 n 上的层数(模大小). 接着，根据各层非零元素的数量大小为各层建立最大堆 $NHeap$(行 2). 然后，初始化 p_n 个划分 $\{P_p\}_{p=1}^{p_n}$ 为空(行 3)，并按各划分含非零元素的数量大小(初始为零)为其建立最小堆 $PHeap$(行 4). 接下来，依次根据各层非零元素数量从大到小的顺序从 $NHeap$ 中取出张量层 $Slice$ 进行分配至当前含张量非零元素数量最小的划分 $Part$ 中，并更新划分结果至 $PHeap$(行 5～10). 最后，M2P 得到张量划分结果 $\{P_p\}_{p=1}^{p_n}$ 并将其输出(行 11～12).

图 3 给出了张量划分算法的示例. 给定一个待划分子张量 $\mathcal{X}_{sub}\in\mathbb{R}^{3\times5\times2}$，如下：

$$\boldsymbol{X}_1=\begin{bmatrix}1&6&0&0&0\\2&0&8&0&0\\0&0&9&0&0\end{bmatrix},\quad \boldsymbol{X}_2=\begin{bmatrix}3&0&1&3&0\\4&7&0&4&6\\5&0&2&5&0\end{bmatrix}.$$

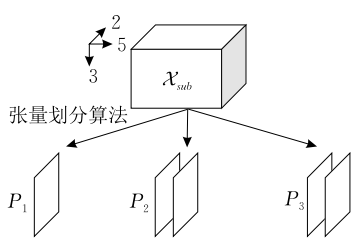


图 3 子张量划分示例

现在需要在子张量 $\mathcal{X}_{sub}$ 第 2 模($n=2$)上得到 3 个划分($p_n=3$). M2P 算法首先在第 2 模上统计可得各层非零元素的数量依次为 5、2、4、3、1,经最大-最小分配策略后,可以得到 3 个划分 P_1、P_2、P_3. 其中,P_1 包含 $\mathcal{X}_{sub}$ 在第 2 模上的第 1 层,即 $P_1=\{\mathcal{X}_{sub}(:,1,:)\}$;$P_2$ 包含 $\mathcal{X}_{sub}$ 在第 2 模上的第 3 层和第 5 层,即 $P_2=\{\mathcal{X}_{sub}(:,3,:),\mathcal{X}_{sub}(:,5,:)\}$;另外,$P_3$ 包含 $\mathcal{X}_{sub}$ 在第 2 模上的第 2 层和第 4 层,即 $P_3=\{\mathcal{X}_{sub}(:,2,:),\mathcal{X}_{sub}(:,4,:)\}$. 三个划分分别都包含 5 个张量非零元素.

作为对比,在上述子张量 $\mathcal{X}_{sub}$ 例子中,GP 算法首先计算得到张量划分非零点数量理论最优值为 5,而后进行贪心划分分别得到的 3 个张量划分 P_1、P_2、P_3. 其中,$P_1=\{\mathcal{X}_{sub}(:,1,:)\}$、$P_2=\{\mathcal{X}_{sub}(:,2,:),\mathcal{X}_{sub}(:,3,:)\}$、$P_3=\{\mathcal{X}_{sub}(:,4,:),\mathcal{X}_{sub}(:,5,:)\}$. 三个划分分别包含 5、6、4 个张量非零元素. 可以看到 M2P 算法得到的张量划分结果相比 GP 算法得到的结果更为均匀,有利于分布式工作节点的负载均衡.

在完成各个子张量的划分之后,DITTD 对增量式张量进行分布式的 Tucker 分解计算,下面第 4.2 节对其进行详细阐述.

4.2　分布式 Tucker 分解计算

在分布式 Tucker 分解计算过程中,如何设计高效的计算算法是一大挑战. 在此过程中,最重要的是避免中间结果爆炸. 在本文所讨论的分布式环境下,张量 Tucker 分解所面临的数据规模本身比单机环境更加庞大,因而避免中间结果爆炸显得尤为重要. 设计高效的 Tucker 分解计算算法,使之尽可能少地产生中间结果,不仅有利于减少计算工作量,而且有利于降低分布式节点之间的网络传输量. 基于上述考虑,本文设计了分布式的增量式张量 Tucker 分解计算方法. 该方法整体分为两个步骤:(1)更新因子矩阵;(2)更新核心张量. 下面对此两个步骤的流程进行详细阐述.

4.2.1　更新因子矩阵

首先,利用第一类子张量 $\mathbb{C}_1=\{\mathcal{X}_{100}^{(t+1)},\mathcal{X}_{010}^{(t+1)},\mathcal{X}_{001}^{(t+1)}\}$ 更新因子矩阵. 根据因子矩阵更新式(1),可以得到三阶张量对应的因子矩阵基于第一类子张量 $\mathbb{C}_1$ 的更新公式:

$$\boldsymbol{A}'\leftarrow(\mathcal{X}_{100}^{(t+1)})_{(1)}(\mathcal{G}_{.00})_{(1)}^{\dagger},$$

$$\boldsymbol{B}'\leftarrow(\mathcal{X}_{010}^{(t+1)})_{(2)}(\mathcal{G}_{0.0})_{(2)}^{\dagger},$$

$$\boldsymbol{C}'\leftarrow(\mathcal{X}_{001}^{(t+1)})_{(3)}(\mathcal{G}_{00.})_{(3)}^{\dagger}.$$

其中,$\mathcal{G}_{.00}=\mathcal{G}^{(t)}\times_2\boldsymbol{B}^{(t)}\times_3\boldsymbol{C}^{(t)}$,$\mathcal{G}_{0.0}=\mathcal{G}^{(t)}\times_1\boldsymbol{A}^{(t)}\times_3\boldsymbol{C}^{(t)}$,$\mathcal{G}_{00.}=\mathcal{G}^{(t)}\times_1\boldsymbol{A}^{(t)}\times_2\boldsymbol{B}^{(t)}$. 由于在此之前未更新过因子矩阵,故遗忘因子 α 取 0.

下面以因子矩阵 $\boldsymbol{A}'$ 的更新计算为例进行讨论. 首先,考察核心张量与因子矩阵乘积 $\mathcal{G}_{.00}=\mathcal{G}^{(t)}\times_2\boldsymbol{B}^{(t)}\times_3\boldsymbol{C}^{(t)}\in\mathbb{R}^{P\times J\times K}$,其对应矩阵化为 $(\mathcal{G}_{.00})_{(1)}\in\mathbb{R}^{P\times(JK)}$. 其中,$J$ 和 K 都是原张量的模大小,因而二者的乘积 JK 是巨大的. 根据第 5.2.1 节实验数据显示,JK 大小可达 10^{13} 量级. 现有的机器都难以生成如此巨大的中间矩阵对象,更难以支持此矩阵伪逆的计算.

上述的中间结果爆炸现象,驱使着本文设计一种优化方法,以避免计算如此巨大的矩阵运算操作. 本文设计了一种计算方法,将上述计算进行等价转换,并拆分巨大的中间矩阵计算,以避免中间结果爆炸问题. 特别地,对于因子矩阵 $\boldsymbol{A}$,进行如下计算转换:

$$\boldsymbol{A}'\leftarrow(\mathcal{X}_{100}^{(t+1)})_{(1)}(\mathcal{G}_{.00})_{(1)}^{\dagger} \tag{3}$$

$$\Leftrightarrow\boldsymbol{A}'\leftarrow(\mathcal{X}_{100}^{(t+1)})_{(1)}(\boldsymbol{C}^{(t)}\otimes\boldsymbol{B}^{(t)})(\boldsymbol{G}_{(1)}^{(t)})^{\dagger} \tag{4}$$

在这个转换过程中,依次利用了矩阵伪逆、张量-矩阵乘法、矩阵克罗内克积以及因子酉矩阵的性质来将更新式(3)转换为更新式(4),详细证明可以参见附录 1.

本文对更新式(3)到更新式(4)的转换主要有以下两点好处:(1)避免了核心张量与因子矩阵乘积的中间结果爆炸;(2)为因子矩阵与增长张量结合进行分布式计算提供了理论基础. 接下来,本文详细阐述因子矩阵 $\boldsymbol{A}'$ 根据更新式(4)的分布式计算过程.

在前一步骤增量式张量划分中,DITTD 已将子张量 $\mathcal{X}_{100}^{(t+1)}$ 尽可能均匀地划分至分布式节点. 根据更新式(4),其余涉及到计算元素为因子矩阵 $\boldsymbol{B}^{(t)}\in\mathbb{R}^{J\times Q}$、$\boldsymbol{C}^{(t)}\in\mathbb{R}^{K\times R}$ 以及核心张量矩阵化 $\boldsymbol{G}_{(1)}^{(t)}\in\mathbb{R}^{P\times(QR)}$. 以上计算元素均为小矩阵,可以按行广播至各个分布式节点进行计算. 接下来,阐述 DITTD 的分布式 Tucker 分解计算步骤,总体分为以下三步:

(1)首先,计算核心张量矩阵化伪逆 $(\boldsymbol{G}_{(1)}^{(t)})^{\dagger}$. 这一步骤因为避免了大矩阵 $(\mathcal{G}_{.00})_{(1)}\in\mathbb{R}^{P\times(JK)}$ 的伪逆计算,将计算复杂度从 $O(\max\{P,(JK)\}^3)$ 降低为 $O(\max\{P,(QR)\}^3)$(一般地,在 Tucker 分解中待分解张量的模大小 I,J,K 远大于核心张量的模大小 P,Q,R).

(2)其次,计算$(\mathcal{X}_{100}^{(t+1)})_{(1)}(\boldsymbol{C}^{(t)}\otimes\boldsymbol{B}^{(t)})$.该项是矩阵化张量乘矩阵克罗内克积(Marticized Tensor Times Kronecker Product,以下简称为 MTTKP),记为$\hat{\boldsymbol{A}}$,$\hat{\boldsymbol{A}}=(\mathcal{X}_{100}^{(t+1)})_{(1)}(\boldsymbol{C}^{(t)}\otimes\boldsymbol{B}^{(t)})$.若显式地计算矩阵克罗内克积$(\boldsymbol{C}^{(t)}\otimes\boldsymbol{B}^{(t)})\in\mathbb{R}^{(KJ)\times(RQ)}$,而后与矩阵化张量$(\mathcal{X}_{100}^{(t+1)})_{(1)}$做矩阵乘法,可能重新遇到前述的中间结果爆炸问题.为此,本文给出了一种 MTTKP 行式计算的策略,以避免产生巨大的中间结果.$\hat{\boldsymbol{A}}$的行式计算公式如下:

$$\hat{\boldsymbol{A}}(i,:)=\sum_{\mathcal{X}_{100}^{(t+1)}(i,:,:)\neq 0}\mathcal{X}_{100}^{(t+1)}(i,j,k)(\boldsymbol{C}^{(t)}(k,:)\otimes\boldsymbol{B}^{(t)}(j,:)) \tag{5}$$

图 4 给出了按照式(5)对 MTTKP 进行行式计算的示例.在各个张量划分中,根据式(5),按所含张量非零元素(如图 4 中黑点所示)的下标 j、k 对应分配因子矩阵$\boldsymbol{B}^{(t)}$、$\boldsymbol{C}^{(t)}$的对应行(如图 4 矩阵 $\boldsymbol{B}$、$\boldsymbol{C}$ 中黑色粗线所示),即可分布式地完成当前划分对应矩阵$\hat{\boldsymbol{A}}$行(如图 4 矩阵$\hat{\boldsymbol{A}}$中黑色粗线所示)的计算.然后,汇总各个划分分别计算好的矩阵$\hat{\boldsymbol{A}}$行,得到完整的 MTTKP 结果矩阵$\hat{\boldsymbol{A}}$.值得一提的是:MTTKP 的分布式行式计算过程决定了在第 4.1 节描述的增量式张量划分过程中,需要依照张量的层次且以各划分包含的非零元素数量均匀为目标进行张量划分.

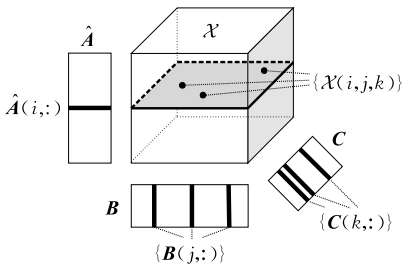


图 4 MTTKP 行式计算示例

(3)最后,将 MTTKP 结果矩阵与核心张量矩阵化伪逆$(\boldsymbol{G}_{(1)}^{(t)})^{\dagger}$相乘得到更新部分的因子矩阵$\boldsymbol{A}'$.此时,$\boldsymbol{A}'$并不满足酉矩阵性质,进一步对其进行改进的格拉姆-施密特正交化过程(Modified Gram-Schmidt process,以下简称为 MGS)处理,以得到列正交的酉矩阵$\boldsymbol{A}'$,供下一步骤更新使用.

类似地,DITTD 基于第一类子张量$\mathbb{C}_1$其余两个子张量$\mathcal{X}_{010}^{(t+1)}$和$\mathcal{X}_{001}^{(t+1)}$对因子矩阵 $\boldsymbol{B}$、$\boldsymbol{C}$ 进行更新,其更新公式如下:

$$\boldsymbol{B}'\leftarrow(\mathcal{X}_{010}^{(t+1)})_{(1)}(\boldsymbol{C}^{(t)}\otimes\boldsymbol{A}^{(t)})(\boldsymbol{G}_{(2)}^{(t)})^{\dagger},$$

$$\boldsymbol{C}'\leftarrow(\mathcal{X}_{001}^{(t+1)})_{(1)}(\boldsymbol{B}^{(t)}\otimes\boldsymbol{A}^{(t)})(\boldsymbol{G}_{(3)}^{(t)})^{\dagger}.$$

在完成基于第一类子张量$\mathbb{C}_1$的因子矩阵更新之后,DITTD 继续基于第二类子张量$\mathbb{C}_2=\{\mathcal{X}_{011}^{(t+1)},\mathcal{X}_{101}^{(t+1)},\mathcal{X}_{110}^{(t+1)}\}$对各个因子矩阵进行更新.每个第二类子张量与两个因子矩阵的更新同时有关.例如,子张量$\mathcal{X}_{011}^{(t+1)}$会被用于因子矩阵$\boldsymbol{B}'$、$\boldsymbol{C}'$的更新.基于第二类子张量$\mathbb{C}_2$的因子矩阵更新公式分别如下:

$$\boldsymbol{B}'^{\text{new}}\leftarrow\alpha\boldsymbol{B}'^{\text{old}}+(1-\alpha)(\mathcal{X}_{011}^{(t+1)})_{(2)}(\boldsymbol{C}'\otimes\boldsymbol{A}^{(t)})(\boldsymbol{G}_{(2)}^{(t)})^{\dagger},$$

$$\boldsymbol{C}'^{\text{new}}\leftarrow\alpha\boldsymbol{C}'^{\text{old}}+(1-\alpha)(\mathcal{X}_{011}^{(t+1)})_{(3)}(\boldsymbol{B}'\otimes\boldsymbol{A}^{(t)})(\boldsymbol{G}_{(3)}^{(t)})^{\dagger},$$

$$\boldsymbol{A}'^{\text{new}}\leftarrow\alpha\boldsymbol{A}'^{\text{old}}+(1-\alpha)(\mathcal{X}_{101}^{(t+1)})_{(1)}(\boldsymbol{C}'\otimes\boldsymbol{B}^{(t)})(\boldsymbol{G}_{(1)}^{(t)})^{\dagger},$$

$$\boldsymbol{C}'^{\text{new}}\leftarrow\alpha\boldsymbol{C}'^{\text{old}}+(1-\alpha)(\mathcal{X}_{101}^{(t+1)})_{(3)}(\boldsymbol{B}^{(t)}\otimes\boldsymbol{A}')(\boldsymbol{G}_{(3)}^{(t)})^{\dagger},$$

$$\boldsymbol{A}'^{\text{new}}\leftarrow\alpha\boldsymbol{A}'^{\text{old}}+(1-\alpha)(\mathcal{X}_{110}^{(t+1)})_{(1)}(\boldsymbol{C}^{(t)}\otimes\boldsymbol{B}')(\boldsymbol{G}_{(1)}^{(t)})^{\dagger},$$

$$\boldsymbol{B}'^{\text{new}}\leftarrow\alpha\boldsymbol{B}'^{\text{old}}+(1-\alpha)(\mathcal{X}_{110}^{(t+1)})_{(2)}(\boldsymbol{C}^{(t)}\otimes\boldsymbol{A}')(\boldsymbol{G}_{(2)}^{(t)})^{\dagger}.$$

依次类推,接下来可以基于第三子张量$\mathbb{C}_3=\{\mathcal{X}_{111}^{(t+1)}\}$对因子矩阵进行更新.基于第三类子张量$\mathbb{C}_3$的因子矩阵更新公式分别如下:

$$\boldsymbol{A}'^{\text{new}}\leftarrow\alpha\boldsymbol{A}'^{\text{old}}+(1-\alpha)(\mathcal{X}_{111}^{(t+1)})_{(1)}(\boldsymbol{C}'\otimes\boldsymbol{B}')(\boldsymbol{G}_{(1)}^{(t)})^{\dagger},$$

$$\boldsymbol{B}'^{\text{new}}\leftarrow\alpha\boldsymbol{B}'^{\text{old}}+(1-\alpha)(\mathcal{X}_{111}^{(t+1)})_{(2)}(\boldsymbol{C}'\otimes\boldsymbol{A}')(\boldsymbol{G}_{(2)}^{(t)})^{\dagger},$$

$$\boldsymbol{C}'^{\text{new}}\leftarrow\alpha\boldsymbol{A}'^{\text{old}}+(1-\alpha)(\mathcal{X}_{111}^{(t+1)})_{(3)}(\boldsymbol{B}'\otimes\boldsymbol{A}')(\boldsymbol{G}_{(3)}^{(t)})^{\dagger}.$$

基于第二类和第三类子张量的因子矩阵更新过程与前述分布式 Tucker 分解计算步骤类似,故此不再赘述.

在完成因子矩阵的增量更新后,将更新部分的因子矩阵与上一时刻的因子矩阵拼接得到当前时刻的因子矩阵,即:

$$(\boldsymbol{A}^{(t+1)})^{\mathrm{T}}=[(\boldsymbol{A}^{(t)})^{\mathrm{T}}(\boldsymbol{A}')^{\mathrm{T}}],$$

$$(\boldsymbol{B}^{(t+1)})^{\mathrm{T}}=[(\boldsymbol{B}^{(t)})^{\mathrm{T}}(\boldsymbol{B}')^{\mathrm{T}}],$$

$$(\boldsymbol{C}^{(t+1)})^{\mathrm{T}}=[(\boldsymbol{C}^{(t)})^{\mathrm{T}}(\boldsymbol{C}')^{\mathrm{T}}].$$

拼接后的矩阵$(\boldsymbol{A}^{(t+1)})$、$(\boldsymbol{B}^{(t+1)})$、$(\boldsymbol{C}^{(t+1)})$不满足酉矩阵性质.因此,进一步对其进行 MGS 处理,以得到列正交的因子酉矩阵.至此完成因子矩阵的更新步骤.

在完成因子矩阵的更新后,DITTD 接下来对核心张量进行更新.下面,第 4.2.2 节对核心张量更新过程进行详细阐述.

4.2.2 更新核心张量

根据核心张量更新式(2),可以得到三阶的增量式张量 Tucker 分解对应的核心张量更新公式,如下所示:

$$\mathcal{G}^{(t+1)}\leftarrow\mathcal{G}^{(t)}\times\{(\boldsymbol{A}_{(n),0}^{(t+1)})^{\mathrm{T}}\boldsymbol{A}_{(n)}^{(t)}\}+\sum_{(i_1i_2i_3)\in\Theta\setminus(000)}\mathcal{X}_{i_1i_2i_3}^{(t+1)}\times\{(\boldsymbol{A}_{(n),i_n}^{(t+1)})^{\mathrm{T}}\} \tag{6}$$

其中,$n=1,2$ 或 3,$\Theta\triangleq\{0,1\}^3$.更新后得到的因子

矩阵$\{\boldsymbol{A}_{(n)}^{(t+1)}\}$被重新拆分为两个部分的独立矩阵$\{\boldsymbol{A}_{(n),0}^{(t+1)}\}$和$\{\boldsymbol{A}_{(n),1}^{(t+1)}\}$. 这里，$(\boldsymbol{A}_{(n)}^{(t+1)})^{\mathrm{T}}=[(\boldsymbol{A}_{(n),0}^{(t+1)})^{\mathrm{T}}(\boldsymbol{A}_{(n),1}^{(t+1)})^{\mathrm{T}}]$，$\boldsymbol{A}_{(n)}^{(t+1)}\in\mathbb{R}^{I_n^{(t+1)}\times R_n}$，$\boldsymbol{A}_{(n),0}^{(t+1)}\in\mathbb{R}^{I_n^{(t)}\times R_n}$以及$\boldsymbol{A}_{(n),1}^{(t+1)}\in\mathbb{R}^{d_n\times R_n}$，$I_n^{(t+1)}=I_n^{(t)}+d_n$.

可以看到更新式(6)的第一项(即加号前一项$\mathcal{G}^{(t)}\times\{(\boldsymbol{A}_{(n),0}^{(t+1)})^{\mathrm{T}}\boldsymbol{A}_{(n)}^{(t)}\}$)中各元素的维度分别对应为$\mathcal{G}^{(t)}\in\mathbb{R}^{R_1\times R_2\times R_3}$、$\boldsymbol{A}_{(n),0}^{(t+1)}\in\mathbb{R}^{I_n^{(t)}\times R_n}$以及$\boldsymbol{A}_{(n)}^{(t)}\in\mathbb{R}^{I_n^{(t)}\times R_n}$. 其中，$I_1^{(t)}=I$，$I_2^{(t)}=J$，$I_3^{(t)}=K$；$R_1=P$，$R_2=Q$，$R_3=R$. 从中可以看到，更新式(6)中第一项涉及的计算元素均为小张量和小矩阵，可以单机完成.

接下来，本文考察更新式(6)的第二项(即加号后一项$\sum_{(i_1i_2i_3)\in\Theta\setminus(000)}\mathcal{X}_{i_1i_2i_3}^{(t+1)}\times\{(\boldsymbol{A}_{(n),i_n}^{(t+1)})^{\mathrm{T}}\}$)，该项为展开的7个分项和. 本文接下来以下标$(i_1i_2i_3)=(001)$的分项为例，即$\mathcal{X}_{001}^{(t+1)}\times_1(\boldsymbol{A}_0^{(t+1)})^{\mathrm{T}}\times_2(\boldsymbol{B}_0^{(t+1)})^{\mathrm{T}}\times_3(\boldsymbol{C}_1^{(t+1)})^{\mathrm{T}}$，对其计算过程进行阐述. 实际上，此项可以看成是MTTKP与矩阵乘积，即：

$$\mathcal{X}_{001}^{(t+1)}\times_1(\boldsymbol{A}_0^{(t+1)})^{\mathrm{T}}\times_2(\boldsymbol{B}_0^{(t+1)})^{\mathrm{T}}\times_3(\boldsymbol{C}_1^{(t+1)})^{\mathrm{T}}=(\boldsymbol{A}_0^{(t+1)})^{\mathrm{T}}(\boldsymbol{X}_{001}^{(t+1)})_{(1)}((\boldsymbol{C}_1^{(t+1)})^{\mathrm{T}}\otimes(\boldsymbol{B}_0^{(t+1)})^{\mathrm{T}}) \tag{7}$$

因此，其计算的策略可以分为如下两步：(1)按照前述MTTKP分布式行式计算的方式得到MTTKP结果$(\boldsymbol{X}_{001}^{(t+1)})_{(1)}((\boldsymbol{C}_1^{(t+1)})^{\mathrm{T}}\otimes(\boldsymbol{B}_0^{(t+1)})^{\mathrm{T}})$；(2)将MTTKP结果与矩阵$(\boldsymbol{A}_0^{(t+1)})^{\mathrm{T}}$相乘得到式(7)的结果.

类似地，可以依次计算得到更新式(6)中第二项其余6个分项的结果，进而加和得到第二项的结果. 最后与更新式(6)中的第一项相加，以完成核心张量的更新. 至此，本文以三阶增量式张量为例，详细阐述了本文提出的分布式的增量式张量Tucker分解方法DITTD.

接下来，第4.2.3节阐述DITTD推广至任意N阶增量式张量的Tucker分解情况.

4.2.3　高阶张量的扩展

下面，本文重新聚焦至任意N阶增量式张量的Tucker分解，整个处理过程是前述三阶增量式张量Tucker分解的推广. 接下来，结合伪码对分布式的任意N阶增量式张量的Tucker分解方法DITTD进行阐述. N阶DITTD方法的伪代码如算法2所示.

算法 2.　N阶DITTD方法.

输入：t时刻张量Tucker分解结果$\mathcal{G}^{(t)}$、$\{\boldsymbol{A}_{(n)}^{(t)}\}$，$t+1$时刻增长张量$\{\mathcal{X}_{i_1i_2\cdots i_N}^{(t+1)}\}_{(i_1i_2\cdots i_N)\in\Theta\setminus(0\cdots0)}$，划分数量$p_n$

输出：$t+1$时刻张量Tucker分解结果$\mathcal{G}^{(t+1)}$、$\{\boldsymbol{A}_{(n)}^{(t+1)}\}$

1. $\{\mathbb{C}_m\}\leftarrow$划分$t+1$时刻增长张量　//分布式处理
2. FOR 子张量类$\mathbb{C}_m\in\{\mathbb{C}_m\}$
3. FOR 子张量$\mathcal{X}_{i_1i_2\cdots i_N}^{(t+1)}\in\mathbb{C}_m$
4. FOR 模$i_n\in(i_1i_2\cdots i_N)\wedge i_n=1$
5. 张量划分算法$(\mathcal{X}_{i_1i_2\cdots i_N}^{(t+1)},i_n,p_n)$
6. 根据式(8)更新因子矩阵$\boldsymbol{A}'_{(n)}$ //行5～6为分布式处理
7. 对因子矩阵$\boldsymbol{A}'_{(n)}$进行MGS处理
8. END FOR
9. END FOR
10. END FOR
11. FOR $n\in\{1,2,\cdots,N\}$
12. 根据式(9)拼接得到因子矩阵$(\boldsymbol{A}_{(n)}^{(t+1)})^{\mathrm{T}}$
13. 对因子矩阵$(\boldsymbol{A}_{(n)}^{(t+1)})^{\mathrm{T}}$进行MGS处理
14. END FOR
15. 根据式(10)更新核心张量$\mathcal{G}^{(t+1)}$　//分布式处理
16. 输出核心张量$\mathcal{G}^{(t+1)}$、因子矩阵$\{\boldsymbol{A}_{(n)}^{(t+1)}\}$

具体来说，面向前一时刻t张量$\mathcal{X}^{(t)}\in\mathbb{R}^{I_1^{(t)}\times I_2^{(t)}\times\cdots\times I_N^{(t)}}$和当前时刻$t+1$增长张量$\mathcal{X}^{(t+1)}\in\mathbb{R}^{I_1^{(t+1)}\times I_2^{(t+1)}\times\cdots\times I_N^{(t+1)}}$，$\mathcal{X}^{(t+1)}$可以看作$\{\mathcal{X}_{i_1i_2\cdots i_N}^{(t+1)}\}_{(i_1i_2\cdots i_N)\in\Theta}$，$\Theta$是$N$项二元组$\Theta\triangleq\{0,1\}^N$，$\mathcal{X}_{0\cdots0}^{(t+1)}=\mathcal{X}^{(t)}$. DITTD方法输入$t$时刻张量的Tucker分解结果$\mathcal{G}^{(t)}$、$\{\boldsymbol{A}_{(n)}^{(t)}\}$，$t+1$时刻增长张量$\{\mathcal{X}_{i_1i_2\cdots i_N}^{(t+1)}\}_{(i_1i_2\cdots i_N)\in\Theta\setminus(0\cdots0)}$以及划分数量$p_n$，DITTD输出$t+1$时刻张量的Tucker分解结果的核心张量$\mathcal{G}^{(t+1)}$以及因子矩阵$\{\boldsymbol{A}_{(n)}^{(t+1)}\}$.

首先，面向N阶的增长张量，根据增量式子张量下标含1的个数，将其分为N类$\{\mathbb{C}_m\}_{m=1}^N$(行1). 接下来，依次按下标含1的个数对各类子张量进行划分(行2～5). 对于特定子张量类别$\mathbb{C}_m$(行2)的子张量$\mathcal{X}_{i_1i_2\cdots i_N}^{(t+1)}$(行3)中$i_n=1$的模(行4)，调用张量划分算法(GP或M2P)对子张量$\mathcal{X}_{i_1i_2\cdots i_N}^{(t+1)}$在模$i_n$上划分成$p_n$份划分(行5). 在完成子张量的划分后，对应以分布式的行式计算方式更新因子矩阵(行6). 其中，根据附录1，N阶张量对应的更新式(1)转化为

$$\boldsymbol{A}'^{\mathrm{new}}_{(n)}\leftarrow\alpha\boldsymbol{A}'^{\mathrm{old}}_{(n)}+(1-\alpha)(\mathcal{X}_{i_1\cdots i_N}^{(t+1)})_{(n)}(\mathcal{G}_{-i_n})^{\dagger}_{(n)}\Leftrightarrow$$
$$\boldsymbol{A}'^{\mathrm{new}}_{(n)}\leftarrow\alpha\boldsymbol{A}'^{\mathrm{old}}_{(n)}+(1-\alpha)(\mathcal{X}_{i_1\cdots i_N}^{(t+1)})_{(n)}(\boldsymbol{A}_{(N)}\otimes\cdots\boldsymbol{A}_{(n+1)}\otimes\boldsymbol{A}_{(n-1)}\cdots\otimes\boldsymbol{A}_{(1)})^{\dagger}(\boldsymbol{G}_{(n)}^{(t)})^{\dagger} \tag{8}$$

若$i_k=0$，则$\boldsymbol{A}_{(k)}=\boldsymbol{A}_{(k)}^{(t)}$；否则$\boldsymbol{A}_{(k)}=\boldsymbol{A}'_{(k)}$，$k=1,2,\cdots,n-1,n+1,\cdots,N$. 上一时刻的核心张量$\mathcal{G}$被记为$\mathcal{G}^{(t)}$，即附录1中的$(\boldsymbol{G}_{(n)})^{\dagger}$与式(8)中的$(\boldsymbol{G}_{(n)}^{(t)})^{\dagger}$对应相等.

完成当前步骤更新后，对其进行MGS处理(行7). 依次类推，直至所有的因子矩阵完成更新(行2～10).

然后，将增量部分因子矩阵$\{\boldsymbol{A}'_{(n)}\}$和因子矩阵$\{\boldsymbol{A}_{(n)}^{(t)}\}$拼接得到当前更新的因子矩阵$\{\boldsymbol{A}_{(n)}^{(t+1)}\}$(行12). 其中：

$$(\boldsymbol{A}_{(n)}^{(t+1)})^{\mathrm{T}}=[(\boldsymbol{A}_{(n)}^{(t)})^{\mathrm{T}}(\boldsymbol{A}'_{(n)})^{\mathrm{T}}] \tag{9}$$

拼接完成后,再次进行 MGS 处理(行 13).以此类推,得到所有符合酉矩阵性质的因子矩阵(行 11～14).至此,DITTD 方法完成了所有因子矩阵的更新计算.

在完成因子矩阵的更新后,DITTD 接着进行核心张量的更新(行 15),更新公式如下:

$$\mathcal{G}^{(t+1)}\leftarrow\mathcal{G}^{(t)}\times\{(\boldsymbol{A}_{(n),0}^{(t+1)})^{\mathrm{T}}\boldsymbol{A}_n^{(t)}\}+\sum_{(i_1\cdots i_N)\in\Theta\setminus(0\cdots0)}\mathcal{X}_{i_1\cdots i_N}^{(t+1)}\times\{(\boldsymbol{A}_{n,i_n}^{(t+1)})^{\mathrm{T}}\} \tag{10}$$

其中,$\Theta\triangleq\{0,1\}^N$,更新的因子矩阵$\{\boldsymbol{A}_{(n)}^{(t+1)}\}$被重新拆分为两个部分矩阵$\{\boldsymbol{A}_{(n),0}^{(t+1)}\}$和$\{\boldsymbol{A}_{(n),1}^{(t+1)}\}$,即$(\boldsymbol{A}_{(n)}^{(t+1)})^{\mathrm{T}}=[(\boldsymbol{A}_{(n),0}^{(t+1)})^{\mathrm{T}}(\boldsymbol{A}_{(n),1}^{(t+1)})^{\mathrm{T}}]$,$\boldsymbol{A}_{(n)}^{(t+1)}\in\mathbb{R}^{I_n^{(t+1)}\times R_n}$,$\boldsymbol{A}_{(n),0}^{(t+1)}\in\mathbb{R}^{I_n^{(t)}\times R_n}$以及$\boldsymbol{A}_{(n),1}^{(t+1)}\in\mathbb{R}^{(I_n^{(t+1)}-I_n^{(t)})\times R_n}$.

值得注意的是:前述所有涉及待分解张量的操作(包括张量划分、因子矩阵更新、核心张量更新),DITTD 方法均采取分布式处理的方式.特别地,因子矩阵更新式(8)和核心张量更新式(10)涉及 MTTKP 的计算都采用前述分布式的行式计算方式,以避免中间结果的爆炸.

至此,DITTD 完成了 $t+1$ 时刻张量 Tucker 计算,输出最终结果的核心张量 $\mathcal{G}^{(t+1)}$、因子矩阵$\{\boldsymbol{A}_{(n)}^{(t+1)}\}$(行 16).

5 实验分析

本节在真实与合成数据集上对本文所提出的 DITTD 方法进行实验测试,并与基准方法进行对比,以验证 DITTD 的效率和可扩展性能.第 5.1 节对本文的实验设置进行说明,第 5.2 节给出实验结果及其对应的分析.

5.1 实验设置

为了全面测试本文所提出方法的性能,本文使用 3 个真实数据集和 1 个合成数据集进行实验测试.3 个真实数据集分别是公开的 Amazon 商品评论数据集① Book 和 Clothing 以及 Netflix 公开的电影评价数据集② Netflix. Amazon 商品评论数据集 Book 和 Clothing 包括了用户-商品-评价时间-评分四元组张量数据,而 Netflix 数据集提供了评论者-电影-评价时间-评分四元组张量数据.除了真实数据集外,本文还按照均匀分布随机合成一个张量数据集 Synthetic.表 1 给出了实验测试中用到数据集的统计信息.其中 I、J、K 分别表示上述张量数据三个模的大小.

表 1 实验测试数据集统计信息

数据集	I	J	K	非零元素密度/%
Book	1.5E07	2.9E06	8.2E03	1.4E-08
Clothing	1.2E07	2.7E06	7.0E03	1.4E-08
Netflix	4.8E05	1.8E04	2.2E03	5.3E-04
Synthetic	1.0E03～2.0E03	1.0E03～2.0E03	1.0E03～2.0E03	1.0E-02～2.0E-01

在整个实验测试过程中,除了特殊说明的情况外,本文默认的数据集及参数设定如下:核心张量模大小 $P=Q=R$ 为 3,张量划分数量 p 为 11,遗忘因子 $\alpha=0.8$.本文将 3 个真实数据集进行预处理,以模拟真实数据随时间增长的情况.具体来说,首先对真实张量数据集根据时间模排序,而后分割得到增长张量数据.默认取前 50%时间的数据为前一时刻 t 的数据,50%～60%时间的数据为当前时刻 $t+1$ 相较前一时刻增长的数据.至于合成数据集 Synthetic,默认使用张量模大小 $I=J=K=1.0\times10^4$、非零元素密度为 1‰的合成张量数据.在此默认设定下,Synthetic 数据集中张量非零元素数量为 1.0×10^9.与真实据集类似,合成数据集 Synthetic 选择其中一个模排序,而后分割得到前 50%时间的数据为前一时刻 t 的数据,50%～60%时间的数据为当前时刻 $t+1$ 相较前一时刻增长的数据.在实验测试中,本文选取一个参数变化,以测试该参数对实验方法性能的影响,并设其余参数为默认值不变.

本文的所有实验程序使用 Scala 语言编写,相关实验代码公开发布于 Github③.实验测试环境为一套由 12 台 Dell 服务器组成的 Spark 分布式集群(默认使用 1 台作为 Master 主节点,其余 11 台作为 Slave 工作节点),服务器的配置为:英特尔至强 CPU 处理器 E5-2650 v4,128 GB 内存,1T 硬盘;Spark 版本为 2.4.4,Hadoop 版本为 2.7.3,Scala 版本为 2.11.8,JDK 版本为 1.8.0.

5.2 实验结果与分析

接下来,首先在第 5.2.1 节中通过与基准方法的对比,验证本文所提出的 DITTD 方法的性能效率;而后,在第 5.2.2 节中对影响 DITTD 方法性能的多个参数设置进行测试,以验证本文所提出的 DITTD 方法的可扩展性.

5.2.1 性能测试实验

如第 2 节张量分解的相关工作所述,当前未有

① https://nijianmo.github.io/amazon/index.html
② https://www.kaggle.com/netflix-inc/netflix-prize-data
③ https://github.com/Jessonky/DITTD

相关研究解决本文所讨论的分布式环境下多模增长张量的 Tucker 分解问题. 唯一解决多模增长的张量 Tucker 分解问题的 eOTD 算法[14]仅适用于单机环境. 因此，本文将 eOTD 算法拓展至 Spark 分布式框架中，实现了对比基准方法 DeOTD. DeOTD 同样基于 DITTD 所使用的启发式张量划分方法进行增长张量的划分，但 DeOTD 无法避免核心张量与因子矩阵乘积的中间结果爆炸问题.

本文考虑了 4 种不同的方法，分别标记为：DITTD-GP、DITTD-M2P、DeOTD-GP 以及 DeOTD-M2P，将本文所提出的 DITTD 与拓展的 DeOTD 进行对比，后缀 GP 和 M2P 分别表示贪心张量划分算法和最大-最小匹配张量划分算法. 图 5 给出了 4 种方法在 Synthetic 数据集上的实验结果，测试指标包括运行时间和中间结果通信量. 图 5(a)和图 5(b)给出了 Synthetic 数据集上张量模大小变化的实验结果. 其中，张量模大小 $I=J=K$ 从 1×10^3 增加至 20×10^3. 图 5(c)和图 5(d)给出了 Synthetic 数据集上张量非零元素密度变化的实验结果. 其中，非零元素密度从 0.1‰变化至 2‰.

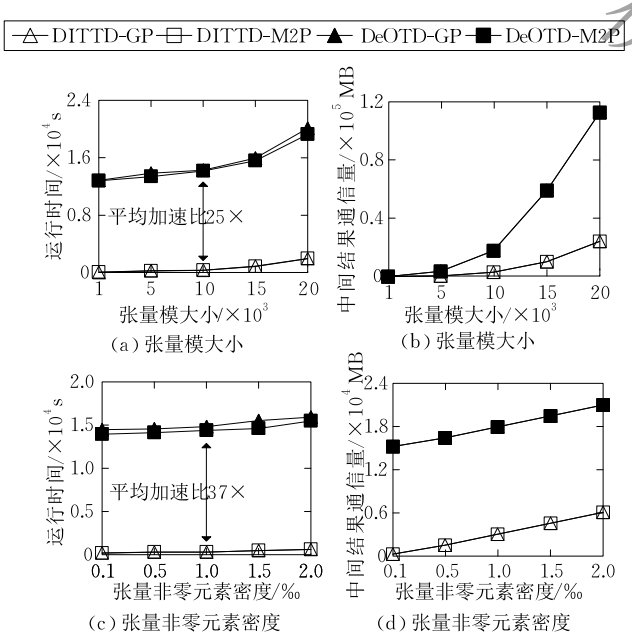


图 5　Synthetic 数据集上的对比结果

首先，从图 5(a)和图 5(b)中可以观察到 4 种方法的运行时间与中间结果通信量均随张量模大小的增大而快速增加；而从图 5(c)和图 5(d)中可以观察到 4 种方法的运行时间与中间结果通信量均随非零元素密度的增大而平稳增加. 这是因为随着张量模大小的增大，各个方法处理的数据规模成指数级增长；而张量非零元素密度的增大只线性地增加处理的张量非零元素数量，且不影响张量模大小.

其次，从图 5 中可以看到本文所提出的方法 DITTD 显著地优于基准方法 DeOTD. 具体地说，在张量模大小变化时，DITTD 比 DeOTD 平均加速了 25 倍；在张量非零元素密度变化时，DITTD 比 DeOTD 平均加速了 37 倍. 此外，DITTD 的中间结果通信量均小于 DeOTD. 其原因是 DITTD 基于转换优化后的 MTTKP 行式计算策略有效地降低了中间结果计算量和分布式工作节点间的通信量，从而提升了性能效率.

最后，从图 5 中可以观察到 M2P 划分算法的运行时间仅比 GP 划分算法稍好，且二者的中间结果通信量对应一致. 这是因为 Synthetic 数据集中非零元素服从均匀分布，两种划分算法均能实现较均匀的张量划分结果. 另外，两种张量划分算法对应 DITTD 与 DeOTD 的中间结果通信量分别相等. 其原因是两种张量划分算法仅仅影响各个分布式工作节点的负载，而不影响分布式集群整体的中间结果通信量.

接下来，本文进一步在 4 个数据集上，考察本文所提出的方法 DITTD 与基准方法 DeOTD 的性能对比结果. 图 6 给出了增长张量大小变化情况下的运行时间对比结果. 其中，前一时刻张量大小被固定为完整张量数据的 50%，而当前时刻张量大小从完整张量数据的 60%变化至 100%.

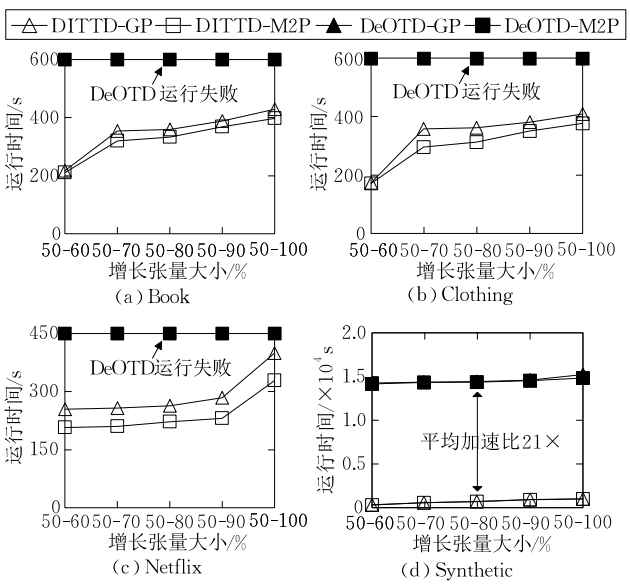


图 6　不同增长张量大小的对比结果

首先,从图 6 中可以看到 DeOTD 在 3 个真实数据集 Book、Clothing 和 Netflix 上均运行失败,仅仅在 Synthetic 数据集上运行成功;而本文所提出的方法 DITTD 在 4 个数据集上均能高效地运行,并在 Synthetic 数据集上比基准方法 DeOTD 平均加速了 21 倍. 其原因是 DITTD 使用转换优化后的 MTTKP 行式计算策略,有效地避免了中间结果爆炸问题. 如第 4.2.1 节所述,若更新因子矩阵时无法避免核心张量与因子矩阵乘积的中间结果,则需要生成且处理巨大的中间矩阵. 具体地说,对于 Book 和 Clothing 数据集,中间矩阵的最大维度大小达到 10^{13};对于 Netflix 数据集,中间矩阵的最大维度大小达到 10^{9};对于 Synthetic 数据集,中间矩阵的最大维度大小达到 10^{8}. 基准方法 DeOTD 无法避免核心张量与因子矩阵乘积的中间结果爆炸问题,需要生成且处理前述巨大的中间矩阵. 因此,DeOTD 只能在中间矩阵相对较小的 Synthetic 数据集上运行成功,而在其余 3 个中间矩阵更巨大的真实数据集上运行失败.

其次,从图 6 中可以观察到当前时刻张量的增长规模从完整张量数据的 10%(即 50%～60%)变化至 50%(即 50%～100%)时,DITTD 以及 DeOTD 的运行时间随之相应地增加. 这是因为随着张量增长规模的增大,DITTD 与 DeOTD 所需处理的数据量变大.

最后,从图 6 中可以看到 M2P 划分算法在 3 个真实数据集 Book、Clothing 和 Netflix 上的运行时间均小于 GP 划分算法. 这是因为 3 个真实数据集中的张量非零元素与 Synthetic 数据集中的张量非零元素不同,且不服从均匀分布. 在非零元素不均匀的情况下,M2P 算法使用最大-最小匹配策略能提供比 GP 更均匀的划分结果. 接下来的第 5.2.2 节进一步深入地讨论了这两种张量划分算法的特性.

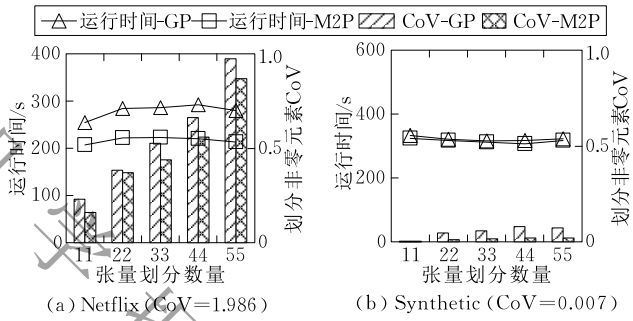
从上述的实验结果可以看出,本文所提出的方法 DITTD 的性能效率比基准方法 DeOTD 提升了 1 个数量级以上,并降低了分布式工作节点间的中间结果通信量.

5.2.2 可扩展性测试实验

接下来,本文对影响 DITTD 性能的多个参数进行测试,以验证其可扩展性. 具体包括张量划分算法以及遗忘因子 α、前一时刻张量大小、核心张量模大小、工作节点数量变化对 DITTD 的性能影响.

(1)张量划分算法性能. 本小节首先评估张量划分算法的性能. 从第 5.2.1 节的实验结果中可以看出两种张量划分算法的性能与数据集中非零元素分布是否均匀有关. 为此,本文引入了概率论和统计学中的"变异系数"概念来度量数据的离散程度. 变异系数,又称离散系数(Coefficient of Variation,以下简称为 CoV),是一个衡量数据离散程度的规一化量度,其定义为标准差与平均值之比.

图 7 给出了 DITTD 分别使用 GP 和 M2P 两种张量划分算法在张量划分数量变化情况下的实验结果. 其中,张量划分数量从 11 变化至 55,使用的数据集分别为 Netflix(非零元素 CoV 为 1.986)和 Synthetic(非零元素 CoV 为 0.007). 首先,从图 7 的折线趋势可以看出,DITTD 的运行时间基本不受张量划分数量的影响. 这说明 DITTD 的性能效率在不同张量划分数量下表现较稳定,在选择张量划分数量时选择分布式工作节点数量的整数倍即可.



(a) Netflix (CoV=1.986) (b) Synthetic (CoV=0.007)

图 7 张量划分算法性能

下面对比 GP 和 M2P 两种张量划分算法的效果. 图 7(b)显示在 Synthetic 数据集上两种张量划分算法的运行时间相近,且两种算法划分结果的 CoV 都较小. 该实验现象的原因是 Synthetic 数据集服从均匀分布,其中非零元素 CoV 较小,GP 和 M2P 均能实现较均匀的张量划分. 作为对比,图 7(a)显示在 Netflix 数据集上,M2P 算法划分结果的 CoV 小于 GP 算法结果,且 M2P 算法运行时间优于 GP 算法. 这是因为 M2P 算法使用了最大-最小匹配策略,比 GP 算法更能适应张量非零元素不均匀分布的情况(即非零元素 CoV 较大的情况,如 Netflix 数据集).

进一步地,本文合成了 5 组不同 CoV 大小的 Synthetic 数据集,以测试两种张量划分算法的性能特性. 这 5 组 Synthetic 数据集统一张量模大小 $I=J=K=1.0\times10^{4}$、非零元素密度为 1‰,非零元素 CoV 从 0 变化至 0.8.

图 8 给出了 GP 和 M2P 两种张量划分算法在对应不同 CoV 大小的 Synthetic 数据集上的实验结果(图 8 的图例与图 7 一致). 从中可以看出，M2P 算法划分结果的 CoV 小于 GP 算法划分结果的 CoV. 相应地，M2P 算法运行时间也略小于 GP 算法运行时间. 其原因是 M2P 算法使用的最大-最小匹配策略比 GP 算法使用的贪心策略更适用于不同离散程度(CoV)的张量数据，能提供更均匀的张量划分结果，以更好地保证分布式工作节点的负载均衡，从而提高算法性能.

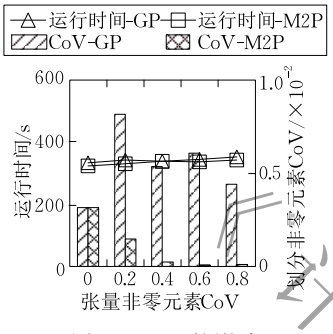


图 8　CoV 的影响

(2) 遗忘因子 α 的影响. 接下来，本文考察遗忘因子 α 对 DITTD 方法运行性能的影响. 图 9 给出了 DITTD 分别使用 GP 和 M2P 两种张量划分算法在 Synthetic 数据集上的运行时间结果. 其中，遗忘因子 α 从 0.1 变化至 0.9. 从中可以看到 DITTD 的运行时间随遗忘因子 α 增大而保持不变. 这是因为遗忘因子 α 只控制多大程度地保留前一步所得信息，而不对 DITTD 方法的计算量(运行时间)产生影响. 另外，从图 9 中还可以观察到 M2P 算法的性能略优于 GP 算法的性能，这与前述张量划分算法性能的实验分析结论一致，不再赘述.

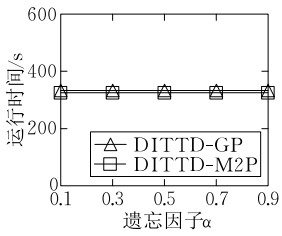


图 9　遗忘因子 α 的影响

(3) 前一时刻张量大小的影响. 下面，本文测试前一时刻张量大小对 DITTD 方法运行性能的影响. 图 10 给出了 DITTD 在 4 个数据集上的运行时间结果. 其中，当前时刻张量增长的比例固定为完整张量数据的 10%大小，前一时刻张量大小从完整张量数据的 50%变化至 90%. 从中可以看到在当前时刻张量增长的规模固定时，DITTD 的运行时间基本不受前一时刻张量大小的影响. 这是因为 DITTD 的性能主要与当前时刻张量的增长量有关. 当张量增长的规模保持不变时，DITTD 能提供性能稳定的增量式张量 Tucker 分解.

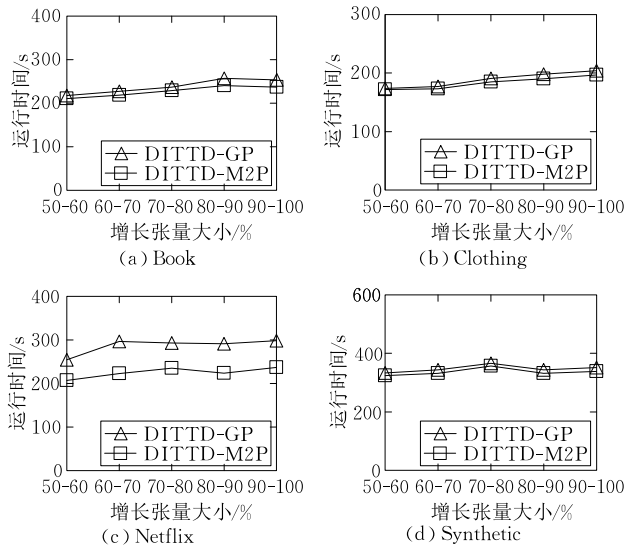


图 10　前一时刻张量大小(固定张量增长量)的影响

(4) 核心张量模大小的影响. 下面，本文考察核心张量模大小对 DITTD 方法运行性能的影响. 图 11 给出了在 4 个数据集上，DITTD 在核心张量模大小 $P=Q=R$ 从 3 增大至 11 情况下的运行时间结果. 从中可以看到，随着核心张量模大小的增长，DITTD 的运行时间也相应增加. 这是因为核心张量模大小的增长伴随着各个因子矩阵的增大，而核心张量以及因子矩阵的增大给分布式的增量式张量 Tucker 分解增加了计算量.

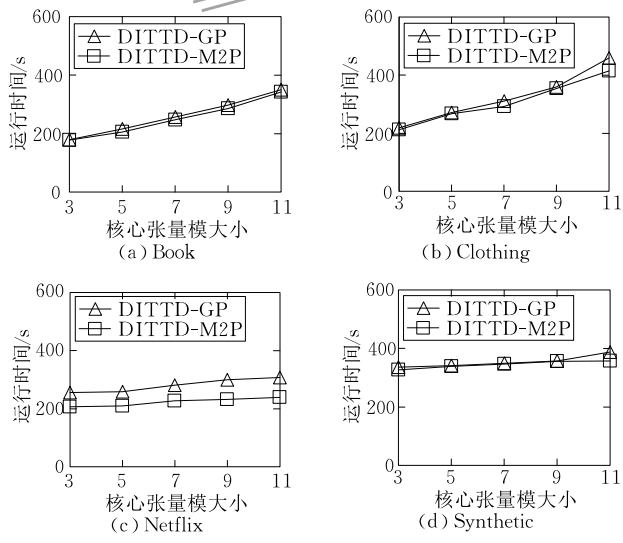
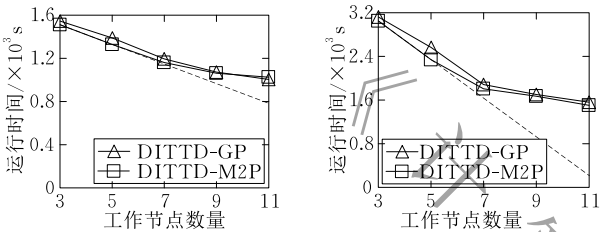


图 11　核心张量模大小的影响

(5) 工作节点数量的影响. 最后，本文考察分布式工作节点数量对 DITTD 方法运行性能的影响.

图 12 给出了 DITTD 在工作节点数量从 3 变化至 11 情况下的运行时间结果. 其中,使用的数据集为 Synthetic. 图 12(a)使用相对前 50% 张量增长至 100%的 Synthetic 张量数据,图 12(b)使用模大小 $I=J=K=20\times10^3$ 的 Synthetic 张量数据. 可以观察到随着工作节点数量的增加,DITTD 的运行时间相应减少;但随着工作节点数量的增加,运行时间减少的幅度逐渐降低(参照图 12 中辅助虚线). 其原因是:随着工作节点数量的增加,整个分布式集群的计算资源增加,这加快了算法的运行;而随着工作节点数量的增加,也为整个分布式集群带来了额外的任务分配开销,这使得整体运行时间减少的相对幅度逐渐降低.



(a) Synthetic (50%~100%) (b) Synthetic ($I=20\times10^3$)

图 12 工作节点数量的影响

6 总 结

本文提出了一种分布式的增量式张量 Tucker 分解方法 DITTD,首次利用分布式框架高效地解决了海量高维且动态增长张量的 Tucker 分解问题. DITTD 方法主要分为两个步骤,即增量式张量划分和分布式 Tucker 分解计算. 在增量式张量的划分步骤中,本文对多模增长的张量进行分类处理,并指出最优张量划分问题是 NP 难的,而后给出基于贪心策略以及最大-最小分配策略的启发式张量划分算法,以尽可能地保证分布式节点的负载均衡. 在分布式的增量式张量 Tucker 分解计算过程中,本文指出当前增量式的张量 Tucker 分解方法存在中间结果爆炸的瓶颈问题,并将其进行等价转换优化,以尽可能少地产生中间结果. 同时,基于分布式的行式计算优化策略,设计了新颖的分布式 Tucker 分解计算算法,既减少了计算量又降低了网络通信量,以达到算法性能提升的目的. 最后,在真实与合成数据集上进行了全面的实验,实验结果验证了本文所提出的 DITTD 方法的效率和可扩展性能. 此外,设计高效的分布式的全量式张量 Tucker 分解方法也是一个重要且值得研究的问题. 后续工作计划对其进行深入地探索.

参 考 文 献

[1] Kolda T G, Bader B W. Tensor decompositions and applications. SIAM Review, 2009, 51(3): 455-500

[2] Papalexakis E E, Faloutsos C, Sidiropoulos N D. Tensors for data mining and data fusion: Models, applications, and scalable algorithms. ACM Transactions on Intelligent Systems and Technology, 2016, 8(2): 1-44

[3] Bahadori M T, Yu Q R, Liu Y. Fast multivariate spatio-temporal analysis via low rank tensor learning//Proceedings of the Advances in Neural Information Processing Systems. Montreal, Canada, 2014: 3491-3499

[4] Zhang Zhi-Qiang, Zhou Yong, Xie Xiao-Qin, et al. Research on advertising click-through rate estimation based on feature learning. Chinese Journal of Computers, 2016, 39(4): 780-794(in Chinese)
(张志强, 周永, 谢晓芹等. 基于特征学习的广告点击率预估技术研究. 计算机学报, 2016, 39(4): 780-794)

[5] Wang H. Facial expression decomposition//Proceedings of the IEEE International Conference on Computer Vision. Nice, France, 2003: 958-965

[6] Xia Zhi-Ming, Xu Zong-Ben. Information compression based on principal component analysis: From one-order to higher-order. Scientia Sinica Informationis, 2018, 48: 1622-1633(in Chinese)
(夏志明, 徐宗本. 基于 PCA 的信息压缩: 从一阶到高阶. 中国科学: 信息科学, 2018, 48: 1622-1633)

[7] Zheng Jian-Wei, Wang Wan-Liang, Yao Xiao-Min, et al. Face recognition using tensor local Fisher discriminant analysis. Acta Automatica Sinica, 2012, 38(9): 1485-1495(in Chinese)
(郑建炜, 王万良, 姚晓敏等. 张量局部 Fisher 判别分析的人脸识别. 自动化学报, 2012, 38(9): 1485-1495)

[8] Signoretto M, Dinh Q T, De Lathauwer L, et al. Learning with tensors: A framework based on convex optimization and spectral regularization. Machine Learning, 2014, 94(3): 303-351

[9] Kang Liang-Yi, Wang Jian-Fei, Liu Jie, et al. Survey on parallel and distributed optimization algorithms for scalable machine learning. Journal of Software, 2018, 29(1): 109-130(in Chinese)
(亢良伊, 王建飞, 刘杰等. 可扩展机器学习的并行与分布式优化算法综述. 软件学报, 2018, 29(1): 109-130)

[10] Li Guo-Liang, Zhou Xuan-He. Survey of data management techniques for supporting artificial intelligence. Journal of Software, 2021, 32(1): 21-40(in Chinese)
(李国良, 周煊赫. 面向 AI 的数据管理技术综述. 软件学报, 2021, 32(1): 21-40)

[11] Yang K, Gao Y, Ma R, et al. DBSCAN-MS: Distributed density-based clustering in metric spaces//Proceedings of the IEEE International Conference on Data Engineering. Macau, China, 2019: 1346-1357

[12] Alam M, Perumalla K S, Sanders P. Novel parallel algorithms for fast multi-GPU-based generation of massive scale-free networks. Data Science and Engineering, 2019, 4(1): 61-75

[13] Fanaee-T H, Gama J. Multi-aspect-streaming tensor analysis. Knowledge-Based Systems, 2015, 89: 332-345

[14] Xiao H, Wang F, Ma F, et al. eOTD: An efficient online Tucker decomposition for higher order tensors//Proceedings of the IEEE International Conference on Data Mining. Singapore, 2018: 1326-1331

[15] Shin K, Sael L, Kang U. Fully scalable methods for distributed tensor factorization. IEEE Transactions on Knowledge and Data Engineering, 2016, 29(1): 100-113

[16] Jeon I, Papalexakis E E, Faloutsos C, et al. Mining billion-scale tensors: Algorithms and discoveries. The VLDB Journal, 2016, 25(4): 519-544

[17] Ge H, Zhang K, Alfifi M, et al. DisTenC: A distributed algorithm for scalable tensor completion on Spark//Proceedings of the IEEE International Conference on Data Engineering. Paris, France, 2018: 137-148

[18] Oh S, Park N, Lee S, et al. Scalable Tucker factorization for sparse tensors-algorithms and discoveries//Proceedings of the IEEE International Conference on Data Engineering. Paris, France, 2018: 1120-1131

[19] Sun J, Tao D, Papadimitriou S, et al. Incremental tensor analysis: Theory and applications. ACM Transactions on Knowledge Discovery from Data, 2008, 2(3): 1-37

[20] Zhou S, Vinh N X, Bailey J, et al. Accelerating online CP decompositions for higher order tensors//Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. San Francisco, USA, 2016: 1375-1384

[21] Song Q, Huang X, Ge H, et al. Multi-aspect streaming tensor completion//Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Halifax, Canada, 2017: 435-443

[22] Hitchcock F L. The expression of a tensor or a polyadic as a sum of products. Journal of Mathematics and Physics, 1927, 6(1-4): 164-189

[23] Carroll J D, Chang J J. Analysis of individual differences in multidimensional scaling via an N-way generalization of "Eckart-Young" decomposition. Psychometrika, 1970, 35(3): 283-319

[24] Harshman R A. Foundations of the PARAFAC procedure: Models and conditions for an "explanatory" multimodal factor analysis. UCLA Working Papers in Phonetics, 1970, 16: 1-84

[25] Tucker L R. Some mathematical notes on three-mode factor analysis. Psychometrika, 1966, 31(3): 279-311

[26] Kroonenberg P M, De Leeuw J. Principal component analysis of three-mode data by means of alternating least squares algorithms. Psychometrika, 1980, 45(1): 69-97

[27] De Lathauwer L, De Moor B, Vandewalle J. A multilinear singular value decomposition. SIAM Journal on Matrix Analysis and Applications, 2000, 21(4): 1253-1278

[28] Bader B W, Kolda T G. Efficient MATLAB computations with sparse and factored tensors. SIAM Journal on Scientific Computing, 2008, 30(1): 205-231

[29] Choi J H, Vishwanathan S. DFacTo: Distributed factorization of tensors//Proceedings of the Advances in Neural Information Processing Systems. Montreal, Canada, 2014: 1296-1304

[30] Beutel A, Talukdar P P, Kumar A, et al. FlexiFaCT: Scalable flexible factorization of coupled tensors on Hadoop//Proceedings of the SIAM International Conference on Data Mining. Pennsylvania, USA, 2014: 109-117

[31] Papalexakis E E, Faloutsos C, Mitchell T M, et al. Turbo-SMT: Accelerating coupled sparse matrix-tensor factorizations by 200x//Proceedings of the SIAM International Conference on Data Mining. Pennsylvania, USA, 2014: 118-126

[32] Karlsson L, Kressner D, Uschmajew A. Parallel algorithms for tensor completion in the CP format. Parallel Computing, 2016, 57: 222-234

[33] Shin K, Kang U. Distributed methods for high-dimensional and large-scale tensor factorization//Proceedings of the IEEE International Conference on Data Mining. Shenzhen, China, 2014: 989-994

[34] Smith S, Ravindran N, Sidiropoulos N D, et al. SPLATT: Efficient and parallel sparse tensor-matrix multiplication//Proceedings of the IEEE International Parallel and Distributed Processing Symposium. Hyderabad, India, 2015: 61-70

[35] Smith S, Karypis G. A medium-grained algorithm for distributed sparse tensor factorization//Proceedings of the IEEE International Parallel and Distributed Processing Symposium. Chicago, USA, 2016: 902-911

[36] Ballard G, Klinvex A, Kolda G. TuckerMPI: A parallel C++/MPI software package for large-scale data compression via the Tucker tensor decomposition. ACM Transactions on Mathematical Software, 2020, 46(2): 1-31

[37] Zhang J, Oh J, Shin K, et al. Fast and memory-efficient algorithms for high-order Tucker decomposition. Knowledge and Information Systems, 2020, 62(1): 1-30

[38] Kang U, Papalexakis E, Harpale A, et al. GigaTensor: Scaling tensor analysis up by 100 times-algorithms and discoveries//Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Beijing, China, 2012: 316-324

[39] Jeon B S, Jeon I, Sael L, et al. SCouT: Scalable coupled matrix-tensor factorization-algorithm and discoveries//Proceedings of the IEEE International Conference on Data Engineering. Helsinki, Finland, 2016: 811-822

[40] Park N, Jeon B, Lee J, et al. BIGtensor: Mining billion-scale tensor made easy//Proceedings of the ACM International on Conference on Information and Knowledge Management. Indianapolis, USA, 2016: 2457-2460

[41] Acer S, Torun T, Aykanat C. Improving medium-grain partitioning for scalable sparse tensor decomposition. IEEE Transactions on Parallel and Distributed Systems, 2018, 29(12): 2814-2825

[42] Nion D, Sidiropoulos N D. Adaptive algorithms to track the PARAFAC decomposition of a third-order tensor. IEEE Transactions on Signal Processing, 2009, 57(6): 2299-2310

[43] Phan A H, Cichocki A. PARAFAC algorithms for large-scale problems. Neurocomputing, 2011, 74(11): 1970-1984

[44] Yu R, Cheng D, Liu Y. Accelerated online low rank tensor learning for multivariate spatiotemporal streams//Proceedings of the International Conference on Machine Learning. Lille, France, 2015: 238-247

[45] Nimishakavi M, Mishra B, Gupta M, et al. Inductive framework for multi-aspect streaming tensor completion with side information//Proceedings of the ACM International Conference on Information and Knowledge Management. Lingotto, Italy, 2018: 307-316

[46] Yang K, Gao Y, Shen Y, et al. DisMASTD: An efficient distributed multi-aspect streaming tensor decomposition//Proceedings of the IEEE International Conference on Data Engineering. Chania, Greece, 2021: 1080-1091

附录 1.

接下来,在定理 2 中给出对更新式(1)进行优化转化的理论推导过程.

定理 2. $(\mathcal{X}_{i_1\cdots i_N}^{(t+1)})_{(n)}(\mathcal{G}_{-i_n})_{(n)}^{\dagger}$ 等价 $(\mathcal{X}_{i_1\cdots i_N}^{(t+1)})_{(n)}(\boldsymbol{A}_{(N)}\otimes\cdots\boldsymbol{A}_{(n+1)}\otimes\boldsymbol{A}_{(n-1)}\cdots\otimes\boldsymbol{A}_{(1)})(\boldsymbol{G}_{(n)})^{\dagger}$.

证明. 根据矩阵伪逆、张量-矩阵乘法、矩阵克罗内克积以及因子酉矩阵的性质,可以有以下推导:

$$
\begin{aligned}
&(\mathcal{X}_{i_1\cdots i_N}^{(t+1)})_{(n)}(\mathcal{G}_{-i_n})_{(n)}^{\dagger}\\
&\Leftrightarrow(\mathcal{X}_{i_1\cdots i_N}^{(t+1)})_{(n)}(\mathcal{G}_{-i_n})_{(n)}^{\mathrm{T}}((\mathcal{G}_{-i_n})_{(n)}(\mathcal{G}_{-i_n})_{(n)}^{\mathrm{T}})^{\dagger}\\
&=(\mathcal{X}_{i_1\cdots i_N}^{(t+1)})_{(n)}(\boldsymbol{G}_{(n)}(\boldsymbol{A}_{(N)}\otimes\cdots\boldsymbol{A}_{(n+1)}\otimes\boldsymbol{A}_{(n-1)}\cdots\otimes\boldsymbol{A}_{(1)})^{\mathrm{T}})^{\mathrm{T}}\\
&\quad((\boldsymbol{G}_{(n)}(\boldsymbol{A}_{(N)}\otimes\cdots\boldsymbol{A}_{(n+1)}\otimes\boldsymbol{A}_{(n-1)}\cdots\otimes\boldsymbol{A}_{(1)})^{\mathrm{T}})\\
&\quad(\boldsymbol{G}_{(n)}(\boldsymbol{A}_{(N)}\otimes\cdots\boldsymbol{A}_{(n+1)}\otimes\boldsymbol{A}_{(n-1)}\cdots\otimes\boldsymbol{A}_{(1)})^{\mathrm{T}})^{\mathrm{T}})\\
&=(\mathcal{X}_{i_1\cdots i_N}^{(t+1)})_{(n)}(\boldsymbol{A}_{(N)}\otimes\cdots\boldsymbol{A}_{(n+1)}\otimes\boldsymbol{A}_{(n-1)}\cdots\otimes\boldsymbol{A}_{(1)})\\
&\quad\boldsymbol{G}_{(n)}^{\mathrm{T}}(\boldsymbol{G}_{(n)}(\boldsymbol{A}_{(N)}\otimes\cdots\boldsymbol{A}_{(n+1)}\otimes\boldsymbol{A}_{(n-1)}\cdots\otimes\boldsymbol{A}_{(1)})^{\mathrm{T}}\\
&\quad(\boldsymbol{A}_{(N)}\otimes\cdots\boldsymbol{A}_{(n+1)}\otimes\boldsymbol{A}_{(n-1)}\cdots\otimes\boldsymbol{A}_{(1)})\boldsymbol{G}_{(n)}^{\mathrm{T}})\\
&=(\mathcal{X}_{i_1\cdots i_N}^{(t+1)})_{(n)}(\boldsymbol{A}_{(N)}\otimes\cdots\boldsymbol{A}_{(n+1)}\otimes\boldsymbol{A}_{(n-1)}\cdots\otimes\boldsymbol{A}_{(1)})\\
&\quad\boldsymbol{G}_{(n)}^{\mathrm{T}}(\boldsymbol{G}_{(n)}((\boldsymbol{A}_{(N)}^{\mathrm{T}}\otimes\cdots\boldsymbol{A}_{(n+1)}^{\mathrm{T}}\otimes\boldsymbol{A}_{(n-1)}^{\mathrm{T}}\cdots\otimes\boldsymbol{A}_{(1)}^{\mathrm{T}})\\
&\quad(\boldsymbol{A}_{(N)}\otimes\cdots\boldsymbol{A}_{(n+1)}\otimes\boldsymbol{A}_{(n-1)}\cdots\otimes\boldsymbol{A}_{(1)}))\boldsymbol{G}_{(n)}^{\mathrm{T}})\\
&=(\mathcal{X}_{i_1\cdots i_N}^{(t+1)})_{(n)}(\boldsymbol{A}_{(N)}\otimes\cdots\boldsymbol{A}_{(n+1)}\otimes\boldsymbol{A}_{(n-1)}\cdots\otimes\boldsymbol{A}_{(1)})\\
&\quad\boldsymbol{G}_{(n)}^{\mathrm{T}}(\boldsymbol{G}_{(n)}((\boldsymbol{A}_{(N)}^{\mathrm{T}}\boldsymbol{A}_{(N)})\otimes\cdots(\boldsymbol{A}_{(n+1)}^{\mathrm{T}}\boldsymbol{A}_{(n+1)})\otimes\\
&\quad(\boldsymbol{A}_{(n-1)}^{\mathrm{T}}\boldsymbol{A}_{(n-1)})\cdots\otimes(\boldsymbol{A}_{(1)}^{\mathrm{T}}\boldsymbol{A}_{(1)})\boldsymbol{G}_{(n)}^{\mathrm{T}})\\
&=(\mathcal{X}_{i_1\cdots i_N}^{(t+1)})_{(n)}(\boldsymbol{A}_{(N)}\otimes\cdots\boldsymbol{A}_{(n+1)}\otimes\boldsymbol{A}_{(n-1)}\cdots\otimes\boldsymbol{A}_{(1)})\\
&\quad\boldsymbol{G}_{(n)}^{\mathrm{T}}(\boldsymbol{G}_{(n)}(I_{R_N\times R_N}\otimes\cdots I_{R_{n+1}\times R_{n+1}}\otimes I_{R_{n-1}\times R_{n-1}}\cdots\otimes\\
&\quad I_{R_1\times R_1})\boldsymbol{G}_{(n)}^{\mathrm{T}})\\
&=(\mathcal{X}_{i_1\cdots i_N}^{(t+1)})_{(n)}(\boldsymbol{A}_{(N)}\otimes\cdots\boldsymbol{A}_{(n+1)}\otimes\boldsymbol{A}_{(n-1)}\cdots\otimes\boldsymbol{A}_{(1)})\\
&\quad\boldsymbol{G}_{(n)}^{\mathrm{T}}(\boldsymbol{G}_{(n)}\boldsymbol{G}_{(n)}^{\mathrm{T}})\\
&=(\mathcal{X}_{i_1\cdots i_N}^{(t+1)})_{(n)}(\boldsymbol{A}_{(N)}\otimes\cdots\boldsymbol{A}_{(n+1)}\otimes\boldsymbol{A}_{(n-1)}\cdots\otimes\boldsymbol{A}_{(1)})\boldsymbol{G}_{(n)}^{\dagger}.
\end{aligned}
$$

即 $(\mathcal{X}_{i_1\cdots i_N}^{(t+1)})_{(n)}(\mathcal{G}_{-i_n})_{(n)}^{\dagger}$

$\Leftrightarrow(\mathcal{X}_{i_1\cdots i_N}^{(t+1)})_{(n)}(\boldsymbol{A}_{(N)}\otimes\cdots\boldsymbol{A}_{(n+1)}\otimes\boldsymbol{A}_{(n-1)}\cdots\otimes\boldsymbol{A}_{(1)})(\boldsymbol{G}_{(n)})^{\dagger}$.

证毕.

YANG Ke-Yu, Ph. D. His research interests include DB for AI technologies and database systems.

GAO Yun-Jun, Ph. D., professor. His research interests include database, big data management and analytics, and AI Interaction with DB technology.

CHEN Lu, Ph. D., professor. Her research interests include database and big data management and analytics.

GE Cong-Cong, Ph. D. candidate. Her research interests include data integration and data quality.

SHEN Yi-Feng, M. S. His research interest is parallel and distributed data processing.

Background

Tensor Tucker decomposition is a fundamental machine learning method for multi-dimensional data analysis, which aims at discovering the latent representations for the given tensor. It is widely applied in many real-life applications, such as recommendation systems, image compression, computer vision, etc.

Nowadays, the overwhelmingly increasing data brings new challenges to traditional tensor decomposition. Many traditional tensor decomposition methods are only suitable for static data and not efficient for dynamic incremental data. This is because those traditional methods could only re-compute the whole tensor decomposition from scratch whenever data

grows. Besides, several incremental tensor Tucker decomposition methods have been proposed for one-mode incremental tensor. However, the tensor in real life could be developed in multiple modes. There is only one method suitable for multi-mode incremental tensor Tucker decomposition. Nevertheless, all the existing incremental tensor Tucker decomposition methods are designed for the centralized environment, and thus, they are not suitable for large-scale dynamic incremental data. Considering the continuous expansion nature of data, it requires an efficient distributed incremental tensor Tucker decomposition method.

This paper proposes a Distributed Incremental Tensor Tucker Decomposition method, i.e., DITTD, which is the first attempt to tackle this problem. First, DITTD divides the incremental tensor based on its positional relationship with the previous one. Then, DITTD tries to achieve the load balancing among the workers in the distributed environment. It requires that DITTD generates the partitioning result for the incremental tensor, such that the number of non-zero elements in each tensor partition is equal. However, the optimal tensor partitioning problem is NP-hard. Thus, DITTD utilizes two heuristic tensor partitioning methods, i.e., Greedy tensor Partitioning algorithm (GP for short) and Max-min Matching tensor Partitioning algorithm (M2P), to partition the incremental tensor as well as possible. After the tensor partitioning, DITTD provides a novel incremental Tucker decomposition computation method to avoid the explosion of intermediate data of Tucker decomposition. This method provides an equivalent conversion for the update rule of factor matrices and designs a row-wise computation strategy for the distributed Tucker decomposition. Based on the above-mentioned techniques, DITTD can reduce the computation of intermediate data and the network communication among the workers. Thus, DITTD improves the efficiency of the distributed Tucker decomposition for incremental tensor. Finally, extensive experiments on both real and synthetic data sets demonstrate the efficiency and scalability of DITTD.

This work was supported in part by the National Key R&D Program of China under Grant No. 2018YFB1004003, and the NSFC under Grants Nos. 62025206 and 61972338.