doi:10.11959/j.issn.1000-436x.2017188

基于分配适应度的 Spark 渐进填充分区映射算法

卞琛[1]，于炯[1]，修位蓉[1]，廖彬[2]，英昌甜[1]，钱育蓉[1]
（1. 新疆大学软件学院，新疆 乌鲁木齐 830008；2. 新疆财经大学统计与信息学院，新疆 乌鲁木齐 830012）

摘 要：分析 Spark 的作业执行机制，建立了执行效率模型和 Shuffle 过程模型，给出了分配适应度（AFD, allocation fitness degree）的定义，提出了算法的优化目标。根据模型的相关定义求解，设计了渐进填充分区映射算法（PFPM, progressive filling partitioning and mapping algorithm），通过扩展式分区和渐进填充映射，建立适应 Reducer 计算能力的数据分配方案，有效缩减 Shuffle 过程的同步延时，提高集群计算效率。实验表明该算法提高了 Shuffle 过程数据分配的合理性，优化了并行计算框架 Spark 的作业执行效率。

关键词：并行计算；Spark；渐进填充；分区映射；分配适应度
中图分类号：TP393.09 **文献标识码**：A

Progressive filling partitioning and mapping algorithm for Spark based on allocation fitness degree

BIAN Chen[1], YU Jiong[1], XIU Wei-rong[1], LIAO Bin[2], YING Chang-tian[1], QIAN Yu-rong[1]
(1. College of Software, Xinjiang University, Urumqi 830008, China;
2. College of Statistics and Information, Xinjiang University of Finance and Economics, Urumqi 830012, China)

Abstract: The job execution mechanism of Spark was analyzed, task efficiency model and Shuffle model were established, then allocation fitness degree (AFD) was defined and the optimization goal was put forward. On the basis of the model definition, the progressive filling partitioning and mapping algorithm (PFPM) was proposed. PFPM established the data distribution scheme adapting Reducers' computing ability to decrease synchronous latency during Shuffle process and increase cluster the computing efficiency. The experiments demonstrate that PFPM could improve the rationality of workload distribution in Shuffle and optimize the execution efficiency of Spark.

Key words: parallel computing, Spark, progressive filling, partitioning and mapping, allocation fitness degree

1 引言

传统的并行计算系统由于其计算模型的天生缺陷，在大数据处理过程中存在 I/O 效率低下、并发控制困难、数据处理总体性能较低等诸多问题，难以有效应对实时、即席、交互式分析的复杂业务诉求[1,2]。因此，并行计算系统的性能优化成为大数据研究领域的热点问题。近年来，随着新兴存储技术的发展，充分利用内存改进系统性能成为并行计算新的研究方向。内存计算框架以其低延迟的优良特性，有效缓解了频繁磁盘 I/O 性能瓶颈，解放了多 CPU 内核配合大内存硬件架构的潜在高性能，成为工业界一致认可的高性能并行计算系统[3,4]。虽然内存计算框架的性能表现相对于传统的并行计算系统提高了数十倍，但与大

收稿日期：2016-11-16；**修回日期**：2017-06-14
通信作者：卞琛，bianchen0720@126.com
基金项目：国家自然科学基金资助项目（No.61262088, No.61462079, No.61562078, No.61363083, No.61562086）；新疆维吾尔自治区自然科学基金资助项目（No.2017D01A20）；新疆维吾尔自治区高校科研计划基金资助项目（No.XJED2016S106）；新疆财经大学科研博士启动基金资助项目（No.2015BS007）

Foundation Items: The National Natural Science Foundation of China (No.61262088, No.61462079, No.61562078, No.61363083, No.61562086), The Natural Science Foundation of Xinjiang Uygur Autonomous Region (No.2017D01A20), The Educational Research Program of Xinjiang Uygur Autonomous Region (No.XJEDU2016S106), The Doctoral Research Foundation of Xinjiang University of Finance and Economics (No.2015BS007)

数据时代的即时应用需求相比，还存在差距。因此，从计算模型的角度研究内存计算框架的性能优化方法具有一定的现实意义。

本文选取开源内存计算框架 Spark[5,6]为研究对象，但并不失一般性，本文的研究成果同样适用于 Flink[7]、Impala[8]、HANA[9]和 MapReduce[10]等其他类似系统。Spark 是继 Hadoop 之后出现的通用高性能并行计算框架，采用弹性分布式数据集（RDD, resilient distributed datasets）[10]作为数据结构，通过世系机制（lineage）[11,12]和检查点机制（checkpoint）[13,14]实现系统容错，编程模式则借鉴了函数式编程语言的设计思想，简化了多阶段作业的流程跟踪、任务重新执行和周期性检查点机制的实现。作为新的基于内存计算的分布式系统，Spark 参考 MapReduce 计算模型实现了自己的分布式计算框架；基于数据仓库 Hive 实现了 SQL 查询系统 Spark SQL[15]；参考流式处理系统 Storm[16]实现了流式计算框架 Spark Streaming[17]，并面向机器学习、图计算领域分别设计了算法库 MLlib[18]和 GraphX[19]。

Spark 的并行化设计思想源于 MapReduce，但与 MapReduce 不同的是，Spark 可以将作业的中间结果保存在内存中，计算过程中不需要再频繁读写 HDFS，从而避免了大量磁盘 I/O 操作，提高了作业的执行效率。因此，Spark 更适用于需要迭代执行的数据挖掘和机器学习算法。由于能够部署在通用平台上，并且具有可靠性（reliable）、可扩展性（scalable）、高效性（efficient）、低成本（economical）等优点[20]，Spark 在大数据分布式计算领域得到了广泛应用，并逐渐成为学术界事实上的大数据并行处理标准。虽然 Spark 具有众多优点，但与其他并行计算框架一样，宽依赖 RDD 的 Shuffle 过程仍是不可规避的性能瓶颈。在 Spark 的 Shuffle 过程中，Mapper 建立与 Reducer 一一对应的 Bucket，数据按 *key* 划分并填入不同的 Bucket，默认的一次分配过程使分配数据量与 Reducer 的计算能力严重不符，导致各 Reducer 的任务执行时间差异很大，从而增加了作业延时，降低计算效率。虽然系统为用户提供了自定义的分区函数，但由于不了解真实的数据分布，难以制定适应计算能力的数据分配方法，因此，分配数据量与 Reducer 计算能力的不适应问题不可规避。为解决这一问题，本文主要做了以下工作。

1) 首先对 Spark 的作业执行机制进行分析，建立执行效率模型，给出了 RDD 计算代价和作业执行时间的定义。

2) 通过分析宽依赖 RDD 的计算过程，建立了 Shuffle 过程模型，给出了数据分布、分区映射和分配适应度的定义，并证明这些定义与作业执行效率和资源利用率的关系，为算法设计提供基础模型。

3) 在相关模型定义和证明的基础上，提出了渐进分区映射算法的问题定义，以此作为算法设计的主要依据。

4) 通过算法的问题定义求解，构建基础元数据，设计了分区扩展算法、分区筛选算法和分区映射算法，并分析了算法的相关优化原则。

2 相关工作

在提出 MapReduce 的文献[21]中，Dean 等采用散列函数对数据进行一次简单的划分，由于这种方法实现简单且通用性高，成为开源的 Hadoop 系统默认的分区方案。Spark 作为类 MapReduce 系统，在实现中也自然承接了 MapReduce 的分区方法，但实际应用表明，在不了解数据分布的情况下，一次散列划分的方法很难实现数据的合理分配。

一些研究成果致力于解决类 MapReduce 系统的数据均衡问题，对原生的数据分配机制进行优化，提出了不同的方法。文献[22]对作业的 Shuffle 过程进行深入分析，探索 Map 和 Reduce 2 个阶段产生数据倾斜的原因，概括出数据倾斜的 5 种分类。文献[23]提出 SkewReduce 策略，通过建立用户定义代价模型评估分区容量，通过作业执行元数据的逐步收集，在达到特定契机时制定均衡的分区策略，该策略是以延迟数据传输为代价解决数据分配的均衡性问题。文献[24]提出 MapReduce 的增量式分区策略，将 Map 端数据划分为更细粒度的 micro-partition，通过已感知数据分布和已分配数据量，采用最大最小公平算法实现数据分配，达到 Reduce 端分配数据量逐渐均衡的目标。文献[25]提出基于任务执行进度的均衡调度策略 SkewTune，与上述的研究成果不同，SkewTune 并不期望在 Map 端制定均衡的分区策略，对原生的分区策略也没有做任何改进，而是通过对 Reduce 端的计算进度进行统计，决定是否将数据向其他工作节点迁移。该策略建立了 Reduce 端的任务剩余代价评估模型，任何 Reduce 端的任务完成，将触发其他未完成任务的剩余代价评估，并将未处理数据向已完成任务的工作节点迁移，从而达到数据分配的整体均衡。

由于会在作业执行过程中进行数据的二次迁移，因此，相比于设计分区策略解决数据均衡问题的方法，SkewTune 具有较大的计算代价。文献[26]在系统中添加 Sketch-based 数据结构，用于分区容量的实时统计和动态调配，并通过设计的分组算法将分区指派给相应的 Reduce 端，达到数据均衡分配的目标。

另外，一些研究成果期望通过了解近似的数据分布制定合理的分区策略解决数据均衡问题。文献[27]提出一种基于采样的分区策略。该策略通过在 Map 端增加独立的采样进程获得近似数据分布，采样达到阈值后对已生成的分区进行拆分和重组，从而提高数据分配的均衡性。文献[28,29]提出精细分区和动态拆分 2 种算法，系统首先通过精细分区算法生成固定数量的分区，同时进行采样获得近似数据分布，当 Map 任务完成一定比例后，触发动态拆分函数，达到数据合理分配的目标。但与上一研究成果不同的是，系统将采样函数附加到 Map 任务中，避免了复杂的通信开销。上述 2 种方法都是先采用原生的散列方法生成分区，当采样达到阈值后进行有且仅有一次的分区调整，因此，阈值的设定非常关键，如果调整时机过早，由于数据分布的精确度不足，数据分配的合理性难以保证，而调整时机过晚则会延迟数据传输，影响计算效率。文献[30]提出 SCID 策略，该策略首先基于蓄水池采样法获得近似的数据分布，在 Map 端根据数据量大小对元组进行排序，然后将数据迭代填充到所有分区，若填充数据量超过分区容量阈值，则启动一次拆分过程，从而确保每个分区数据量相对均匀。文献[31,32]提出基于数据块的采样分区方法，该方法将原生的键值对转换为<*blocking_key*,*entity*>形式，通过设计评估函数对块内数据进行评估，对不符合条件的数据块进行调整，但分区调整仅有一次，没有解决如何定义分区调整时机的问题。文献[33]提出 LIBRA 策略，该策略通过系统空闲资源槽执行采样程序，从而以更轻量级的方式获取近似数据分布，分区策略仍采用二次划分机制，对超限数据元组进行拆分，保障数据分配的均衡性。文献[34]提出先通过采样制定分区函数，再执行任务填充分区的方法。该方法在 Map 任务执行前先运行采样进程，对输入数据进行 25%的随机采样，通过采样结果获得数据分布并制定分区函数，然后启动 Map 任务，采用制定的分区函数填充数据。文献[35]提出 LEEN 策略，通过对输入数据的预扫描获取数据分布，在 Map 任务执行过程中对 *key* 值的频率进行统计，然后综合数据分布和 *key* 频率统计设定合理的分区函数。该策略有效提高了数据分配的均衡性，但由于在原生系统上嵌入了多个功能模块，算法的时间复杂度较高。

其他一些研究成果则考虑不同的工作场景和应用需求。文献[36]没有在即有的分区策略上做任何改进，而是通过调整 HDFS 即有的副本策略，提高数据访问本地性，缓解数据倾斜的影响。文献[37]通过采样感知数据的近似分布，综合距离判定和开销矩阵制定最优的调度策略，以减少通信开销的方法缓解数据倾斜的影响。文献[38]通过支持向量机模型对集群环境进行性能预测，并设计结构感知的数据分区方法。文献[39]针对进化采样方法样本密度不均衡问题，将高密度样本和低密度样本分类管理和采样，保障采样集的均衡性。文献[40]提出实体匹配应用中的数据均衡方案，通过自适应调整的数据窗口，实现近邻排序数据的均匀分配。

上述研究成果普遍以数据的均衡分配为目标，未考虑节点计算能力差异和当前工作负载状况，仅适用于同构且任务分布相对均匀计算集群。本文与以上研究成果的不同之处在于，充分考虑分区映射算法在异构集群和虚拟集群的适应性问题，从并行计算模型的基本原理入手，将节点计算能力和当前工作状况作为 Shuffle 过程数据分配的主要依据，设计了渐进填充分区映射算法，提高数据分配与节点计算能力的匹配度，优化作业执行效率。通过分析作业的执行过程，建立了执行效率模型，提出了 RDD 计算代价和作业执行时间的定义，建立 Shuffle 过程模型，提出了分配适应度的定义，并证明定义与作业执行效率的逻辑关系。根据渐进填充分区映射算法的问题定义进行求解，提出了分区扩展算法、分区筛选算法和分区映射算法，通过扩展式分区，为数据的渐进填充奠定基础。通过适度倾斜的数据分配，充分利用高效工作节点的计算能力，减少 Reduce 任务的同步开销，从而从整体上优化作业执行效率，改进系统性能。

3　问题的建模与分析

分析作业的并行执行机制，建立执行效率模型和 Shuffle 过程模型，提出渐进填充分区映射的问题定义，为渐进填充分区映射算法提供理论基础。

3.1 作业执行机制

Spark 的作业执行采用了延时调度机制，即当用户对一个 RDD 执行 Action（如 count、collect）操作时，调度器会根据 RDD 的世系构建一个由 Stage 组成的有向无环图（DAG），然后为工作节点分配任务执行程序。Spark 任务 DAG 的典型示例如图 1 所示。其中，实线圆角方框表示 RDD，填充矩形表示分区，虚线框为 Stage。Action 操作的执行将会以宽依赖为界来构建各个 Stage，每个窄依赖 Stage 都包含尽可能多的连续的 RDD，其内部的 RDD 分区前后连接构成流水线，而宽依赖 Stage 仅包含一个 RDD 的 Shuffle 操作。各 Stage 同步顺序执行，直到最终得出目标 RDD。各工作节点的任务分配根据数据存储本地性来确定，若一个任务需要处理的某个分区刚好存储在某个节点的内存中，则该任务会分配给这个节点。否则，将任务分配给具有最佳位置的工作节点。

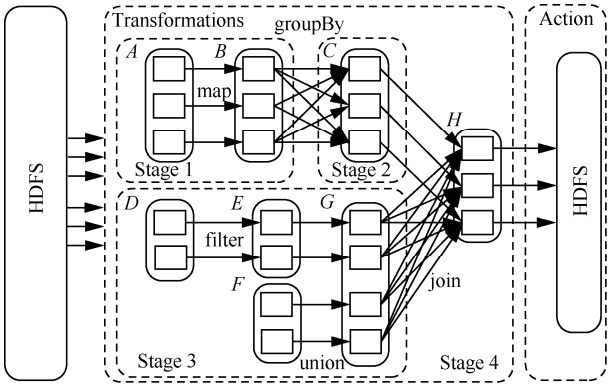

图 1 Spark 作业的有向无环图

3.2 执行效率模型

根据 Spark 的延迟调度机制，作业在执行到 Action 操作时，生成由多个 RDD 组成的 DAG，首先以宽依赖分界划分 Stage，每个 Stage 包括一个或多个 RDD，每个 RDD 又被划分成多个分区工作节点并行计算，因此，对于每一个作业，记其 Stage 集合为 $stages=\{stage_1,stage_2,\cdots,stage_i\}$，记每个 Stage 的 RDD 集合为 $stage_i=\{RDD_{i1}, RDD_{i2},\cdots,RDD_{ij}\}$，这里，$RDD_{ij}$ 表示任务 i 中第 j 个 RDD，对于每个 RDD，记其分区集合为 $RDD_{ij}=\{P_{ij1},P_{ij2},\cdots,P_{ijk}\}$，其中，$P_{ijk}$ 表示 RDD_{ij} 中的第 k 个分区。

定义 1 RDD 计算代价。Spark 任务中，分区是以一个或多个父节点为输入数据计算生成，设 $Parents_{ijk}$ 为分区 P_{ijk} 的父节点集合。分区的计算首先要读取所有的输入数据，然后根据闭包和操作类型进行计算，因此，分区 P_{ijk} 的计算代价应为数据读取代价与数据处理代价之和。以分区计算时间作为衡量计算代价的唯一指标，即

$$T_{P_{ijk}}=\mathrm{read}(Parents_{ijk})+\mathrm{proc}(Parents_{ijk}) \tag{1}$$

其中，数据处理代价不仅是计算时长，还包括可能产生的内存回收及溢写磁盘的延时。

RDD 的所有分区由集群工作节点并行计算生成，因此，其计算代价为所有分区计算代价的最大值，即

$$T_{RDD_{ij}}=\max(T_{P_{ij1}},T_{P_{ij2}},\cdots,T_{P_{ijk}}) \tag{2}$$

定义 2 作业执行时间。如图 1 所示，Spark 以宽依赖为分界点，将作业划分为多个 Stage 执行，Stage 分为窄依赖和宽依赖 2 种。对于窄依赖 Stage，每个 Stage 包括多条流水线（每条流水线包括多个 RDD 的不同分区）。设 $stage_i$ 为窄依赖，共有 h 个 RDD，所有 RDD 划分为 x 条流水线，单条流水线的分区集合为 $pipe_{ix}=\{P_{i1x},P_{i2x},\cdots,P_{ijx}\}$，那么单条流水线的执行时间可表示为

$$T_{pipe_{ix}}=\sum_{j=1}^{h}T_{P_{ijx}} \tag{3}$$

对于 $stage_i$，记其流水线集合为 $Pipes_i=\{pipe_{i1}, pipe_{i2},\cdots, pipe_{ix}\}$，那么 $stage_i$ 的执行时间应为各流水线执行时间最大值，即

$$T_{stage_i}=\max(T_{pipe_{i1}},T_{pipe_{i2}},\cdots,T_{pipe_{ix}}) \tag{4}$$

设 $stage_{i+1}$ 为宽依赖，则其中仅包含一个 RDD 的计算任务，记为 $RDD_{(i+1)j}$，那么 $stage_i$ 的执行时间与 $RDD_{(i+1)j}$ 的计算代价相同，即

$$T_{stage_{i+1}}=T_{RDD_{(i+1)j}} \tag{5}$$

若 Spark 将作业划分为 n 个 Stage（其中，包括多个窄依赖和多个宽依赖的 Stage），各 Stage 同步顺序执行，因此作业的执行时间为

$$T_{\mathrm{job}}=\sum_{i=1}^{n}T_{stage_i} \tag{6}$$

3.3 Shuffle 过程模型

根据 3.2 节 Spark 作业的执行过程，宽依赖 Stage 的执行分解成 Map 的 Reduce 2 个阶段，而 Shuffle 是 Map 和 Reduce 之间的桥梁。Map 阶段将前一 Stage 的计算结果转化为<*key*,*value*>的键值对，并将这些键值对以 *key* 为关键字写入不同的 Bucket 中，

每个 Bucket 映射到一个 Reducer，当一个 Map 任务结束后，各 Reducer 从 Bucket 中拉取数据，作为后续计算的输入。本文对图 1 中 $B \to C$ 宽依赖 RDD 进行分解，具体操作过程如图 2 所示。

定义 3　数据分布。用于描述不同 *key* 在 Mapper 上的分布情况。设数据中共有 l 个不同的 *key*，即 $keys=\{1,2,\cdots,l\}$，记参与宽依赖 RDD 计算的 Mapper 集合为 $mp=\{1,\ 2,\cdots,m\}$，对于任意 Mapper，其数据分布向量可表示为

$$\boldsymbol{D}_m=(d_{m1},d_{m2},\cdots,d_{ml})^{\mathrm{T}} \tag{7}$$

其中，d_{ml} 表示第 l 个 *key* 在第 m 个 Mapper 上的数据量。那么数据的整体分布（在所有 Mapper 上的分布）为 $m\times l$ 矩阵，可表示为

$$\boldsymbol{D}=\begin{bmatrix} d_{11} & d_{21} & \cdots & d_{m1} \\ d_{12} & d_{22} & \cdots & d_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ d_{1l} & d_{2l} & \cdots & d_{ml} \end{bmatrix} \tag{8}$$

矩阵中同行元素表示相同 *key* 在不同 Mapper 上的数据分布，在实际的映射过程中，相同 *key* 也由同一 Reducer 执行计算，因此，本文对于相同 *key* 的数据量进行归并，任意 *key* 的数据总量可表示为

$$ck_l=\sum_{m\in mp} d_{ml}, l\in keys \tag{9}$$

那么，将数据按 *key* 进行划分，可表示为

$$S=\{ck_1,ck_2,\cdots,ck_l\}, l\in keys \tag{10}$$

定义 4　分区映射。用于描述 Map 任务<*key*, *value*>与处理数据的 Reducer 之间的映射关系。根据图 1 描述的结构，分区映射实际也表示每个 Reducer 所对应 Bucket 的生成规则。在 Spark 原生系统中，分区规则可以由用户指定或采用默认规则。默认规则将 *key* 取散列值后，与 Reducer 的数量取模，放入对应的 Bucket 中，以此决定该 *key* 由哪个 Reducer 完成后续计算，因此 Shuffle 过程的分区函数可表示为

$$f(Bucket)=hash(key)\bmod(n) \tag{11}$$

其中，n 表示作业中 Reducer 的个数。经分区函数计算后，相同 *key* 的记录集中在同一个 Bucket，等待 Reducer 拉取数据。由于所有 Mapper 使用的分区函数相同，可以保证多个 Mapper 上的相同 *key* 都分配给同一个 Reducer 所对应的 Bucket 中（这些 Bucket 分布在不同的工作节点上）。

根据数据分布的定义，记参与计算的 Reducer 集合为 $rd=\{1,2,\cdots,n\}$，设数据中 l 个 *key* 已按其散列值顺序排列，那么任意 Reducer 的分区映射关系可表示为

$$input_{rd_i}\mapsto\{ck_i,ck_{n+i},\cdots,ck_{j\times n+i}\}, j\in\left[0,\frac{l}{n}\right] \tag{12}$$

因所有 Map 任务生成的数据都要对应 Reducer 进行后续计算，所以分区映射满足

$$S=input_{rd_1}\cup input_{rd_2}\cup\cdots\cup input_{rd_n} \tag{13}$$

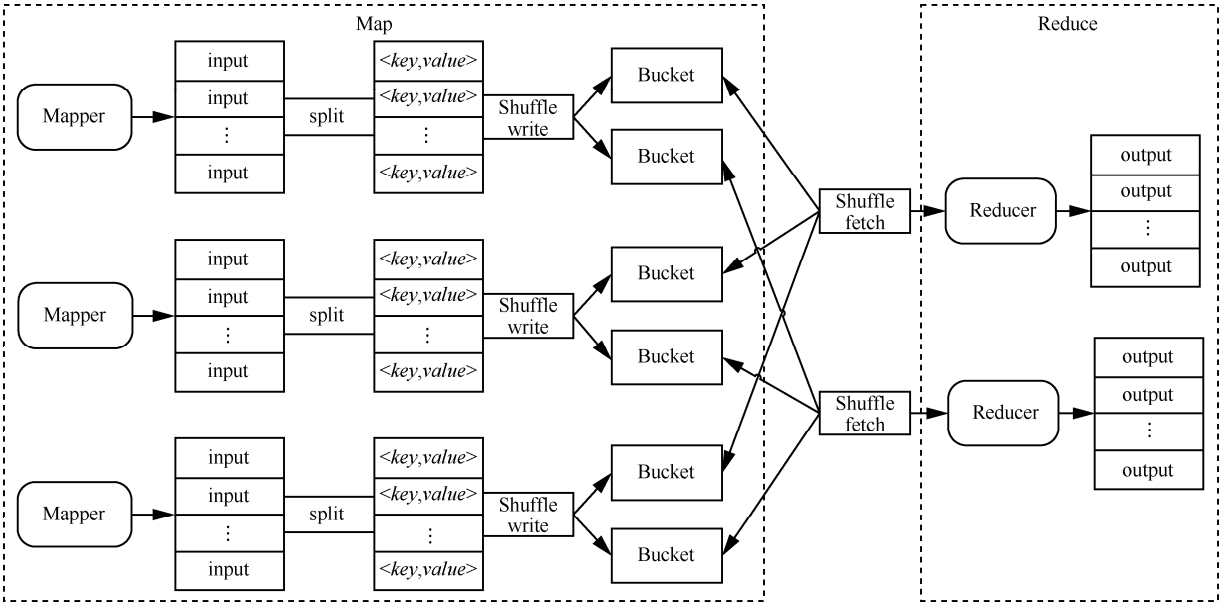

图 2　宽依赖 Stage 的任务分解

定义 5 分配适应度。用于衡量 Reducer 计算能力与其分配数据量的匹配程度。根据定义 4，记 S 为 Map 任务输出的数据总量，由于数据要先拉取再计算，对于执行后续计算的 Reducer 集合 $rd=\{1,2,\cdots,n\}$，记 $DTR=\{tr_1,tr_2,\cdots,tr_n\}$为 rd 中每个 Reducer 在数据拉取时长中的平均数据传输率，记 $CPS=\{cp_1,cp_2,\cdots,cp_n\}$表示每个 Reducer 的数据处理能力。那么所有 Reduce 任务执行时间的均值可定义为

$$E=\frac{S}{\sum_{i\in rd}tr_i}+\frac{S}{\sum_{i\in rd}cp_i} \tag{14}$$

对于第 i 个 Reducer，其任务的计算时间可表示为

$$T_{\text{finish}_i}=\frac{input_{rd_i}}{tr_i}+\frac{input_{rd_i}}{cp_i},i\in rd \tag{15}$$

因此，Reduce 任务执行时间的方差可表示为

$$DX_i=(T_{\text{finish}_i}-E)^2 \tag{16}$$

那么，该 Reducer 的分配适应度可表示为

$$V_i=\frac{1}{DX_i}=\frac{1}{(T_{\text{finish}_i}-E)^2} \tag{17}$$

定理 1 对于所有参与计算的 Reducer，其分配适应度越高，作业的执行时间越短。

证明 设分区映射过程发生于宽依赖 $Stage_i$，基于定义 3，$Stage_i$ 的执行时间与 RDD_{ij} 的计算代价相同，从任务分配的角度来看，参与计算的每个 Reducer 负责 RDD_{ij} 一个分区的计算工作，因此 $Stage_i$ 的执行时间也可表示为

$$T_{stage_i}=\max(T_{\text{finish}_1},T_{\text{finish}_2},\cdots,T_{\text{finish}_n}) \tag{18}$$

根据式(17)，Reducer 的分配适应度与方差成反比，因此适应度越大，方差越小，表示 reduce 任务完成时间越趋近均值，因此当所有 Reducer 的分配适应度取最大值时，宽依赖 Stage 的执行时间最短，进而提高作业的执行效率。证毕。

3.4 目标定义

前面几节已经对作业执行机制、任务执行效率和宽依赖 Shuffle 过程做了比较详细的阐述，基于这些定义，渐进填充分区映射算法的优化目标可形式化为

$$\text{object}\,(\forall i)\max(V_i),i\in rd \tag{19}$$

$$\text{s.t.}\ S=input_{rd_1}\cup input_{rd_2}\cup\cdots\cup input_{rd_n} \tag{20}$$

目标是最大化分配适应度，约束条件是所有数据都应由 Reducer 拉取并计算。很显然，在分区映射中，根据节点计算能力做适度倾斜，可以使 Reducer 得到最大化的分配适应度。

4 渐进填充分区映射算法

本节基于模型的相关定义及定理证明，首先构建算法所需的基础数据，然后提出渐进填充分区映射算法，最后对算法符合的基本原则进行分析。

4.1 算法的总体描述

根据 3.4 节的定义，渐进填充分区映射算法的优化目标是提高分配适应度。为此，本文对原生系统做了部分改动。

1) 对原生系统中按 *key* 划分 Bucket 的策略不做修改，相同的 *key* 依旧在同一个 Bucket 中。为实现适应计算能力的分区映射方案，对系统的分区函数进行优化（如 4.3 节所示），通过引入扩展系数 δ，将数据划分到默认区和扩展区，使 Reducer 与 Bucket 的关系改为一对多，进而通过多轮渐进填充提高分配数据量与 Reducer 计算能力的适应度。

2) 弱化原生系统中的 Stage 边界，即无需等待所有 Map 任务同步，Reducer 可在某个 map 任务完成后即开始拉取数据。这样，一方面使 Mapper 和 Reducer 实现局部并行，加速作业执行；另一方面能够读取任务执行的元数据，作为计算能力评估的主要依据，为渐进填充分区映射奠定基础。

3) 在系统中引入分配系数 λ，用于确定扩展区的分配轮数，分区筛选算法和分区映射算法通过 λ 轮分配，将扩展区数据映射到 Reducer 集合。

4) 在 Mapper 上增加计数器 *mapcounter*，用于统计不同 Bucket 的数据分布；在 Reducer 增加计数器 *fetchcounter*，用于统计 Reducer 已拉取的数据量。以这些元数据为基础，建立每轮分区映射的筛选规则和映射规则。

如图 3 所示，渐进填充分区映射算法的主要过程如下。

1) 根据 Map 任务数 m、Reducer 数量 n 和扩展系数 δ，构建算法所需的元数据。

2) 根据扩展系数 δ，执行分区扩展算法，Mapper 将数据填充到相应的 Bucket 中。

3) 根据分区扩展算法的机制，默认区数据直接映射，而扩展区在达到特定时机时开始数据分配，每轮分配中，根据分区筛选算法（如 4.4 节所示）从 $\delta\times n$ 个 Bucket 中挑选出待分配列表。

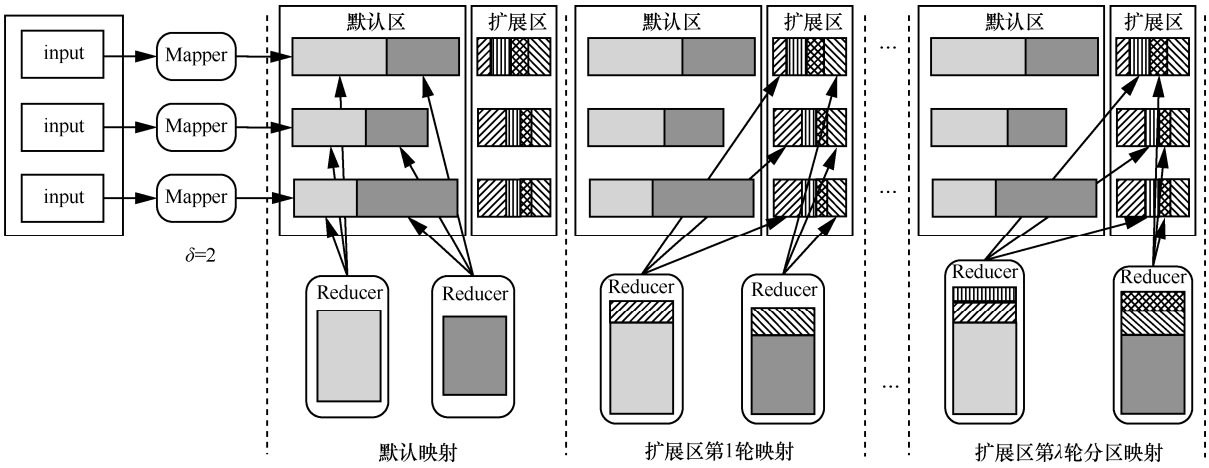

图 3　渐进填充分区映射算法的具体过程

4) 根据分区映射算法（如 4.5 节所示），将步骤 3)中待分配列表中 Bucket 映射到不同的 Reducer。

5) 迭代步骤 3)和步骤 4)，经过 λ 轮分配，全部 Bucket 分别映射到 n 个 Reducer，数据分配过程完成。

4.2　构建算法元数据

本节讨论算法基础数据的构建问题，为便于描述，记用户设置的扩展系数为 δ，作业的 Mapper 集合为 mp={1, 2,⋯,m}，Reducer 集合为 rd={1, 2,⋯,n}，每个 Mapper 将划分到 $s=\delta\times n$ 个 Bucket 中（划分方式如 4.3 节所示）。迭代渐进填充分区映射算法所需的元数据如下。

1) 全局数据分布。定义 $\boldsymbol{b}_i=(bk_{i1},bk_{i2},\cdots,bk_{ij})$为第 $i(i\in[1,m])$ 个 Mapper 的数据分布向量，其中，bk_{ij} 为第 $j(j\in[1,s])$ 个 Bucket 的数据量，bk_{ij} 的计算由计数器 *mapcounter* 完成。对所有 Mapper 数据分布向量按 Bucket 编号进行累积，得到数据统计向量 $\boldsymbol{B}=(\sum_{i\in mp}b_{i1},\sum_{i\in mp}b_{i2},\cdots,\sum_{i\in mp}b_{ij})$，表示全局数据分布。在任务执行过程中，*mapcounter* 每个心跳周期更新一次数据分布向量 $\boldsymbol{b}_i$，发送给 Master，再由 Master 完成数据统计向量 $\boldsymbol{B}$ 的更新工作。因此，数据统计向量的更新是随时间推移逐步求精的过程，在所有 Map 任务都完成时，才能获得最准确的全局数据分布。

2) 分区映射矩阵。基于全局数据分布的定义，构建一个 $n\times s$ 的矩阵 $\boldsymbol{BR}_{n\times s}$，用于表示 Bucket 与 Reducer 之间的映射关系。

$$\boldsymbol{BR}_{n\times s}=\begin{bmatrix} br_{11} & br_{12} & \cdots & br_{1s} \\ br_{21} & br_{22} & \cdots & br_{2s} \\ \vdots & \vdots & \ddots & \vdots \\ br_{n1} & br_{n2} & \cdots & br_{ns} \end{bmatrix} \tag{21}$$

其中，$br_{ij}\in\{0,1\}(i\in[1,n],j\in[1,s])$，状态 1 表示第 j 个 Bucket 已映射到编号 i 的 Reducer，状态 0 则表示第 j 个 Bucket 与编号 i 的 Reducer 无映射关系。在初始状态下，$\boldsymbol{BR}_{n\times s}$ 的所有元素值均为 0，每轮分配将选择 n 个 Bucket 与 Reducer 建立映射关系，并将 $\boldsymbol{BR}_{n\times s}$ 相应位置的 n 个元素值置为 1。由于每个 Bucket 只能映射到一个 Reducer，因此分区映射矩阵 $\boldsymbol{BR}_{n\times s}$ 每列至多有一个元素值为 1。在分区筛选时，待分配 Bucket 必须符合条件：$\sum_{i\in rd}br_{ij}=0$，表示第 j 个 Bucket 未与任何 Reducer 建立映射关系。经过 λ 轮分配，$\forall j$ 均满足 $\sum_{i\in rd}br_{ij}=1$，即所有 Bucket 都要映射到相应的 Reducer 上。

3) 数据拉取向量。定义向量 $\boldsymbol{F}=(f_1,f_2,\cdots,f_k)$，其中，$f_k$ 表示当前时间点编号为 $k(k\in[1,n])$ 的 Reducer 已拉取的数据量，f_k 数据的更新工作由计数器 *fetchcounter* 完成，f_k 的更新次数与分配轮数 λ 相同，每轮分区筛选启动即开始统计各 Reducer 已拉取的数据量。数据拉取向量体现了各 Reducer 拉取数据的快慢程度，在分区映射算法中，将成为 Reducer 计算能力排序的主要依据。

4) MR 关联表。定义三元组 mr_i=<mts_i,rts_i,*flag*>，用于表示工作节点 i 上 Mapper 与 Reducer 的对应关系。其中，mts_i 表示节点 i 分配的 Map 任务集合，rts_i 表示节点 i 分配的 Reduce 任务集合，*flag* 则为 Map 任务是否完成的标识量。因此，所有工作节点 Mapper 与 Reducer 对应关系为 $MR=\{mr_1,mr_2,\cdots,mr_j\}$。

5) 历史排序集。Master 上维护一个<*id*,*rank*>的键值对集合 *ES*，用于表示历史作业执行所生成的节点计算能力排名，其中，*id* 表示节点编号，*rank*

表示排名。*ES* 通过集群作业的逐渐累积生成，每当有作业执行完成，读取作业的日志信息，将节点根据其任务完成的先后顺序进行排列，并将新生成的节点排名更新到相应的集合项中。更新过程中若 *ES* 集合的排名为空则直接写入，否则取 *ES* 集合已有值与新生成排名的平均值。

元数据为渐进填充分区映射算法的后续步骤提供数据和计算支持，其中，全局数据分布是分区筛选算法的主要依据，筛选过程则要通过分区映射矩阵确定待分配候选列表。而分区映射算法也要根据分区映射矩阵和全局数据分布统计各 Reducer 的已分配数据量。在每轮分区映射过程中，通过 MR 关联表、数据拉取向量和历史排序集对 Reducer 排序，从而确定筛选分区与 Reducer 的映射关系。

4.3 分区扩展算法

分区扩展算法的主要目的是将数据分别填充到默认区和扩展区，使 Reducer 与 Bucket 的关系变为一对多，从而为后续启发式数据分配奠定基础。分区扩展算法的主要思想如下所示。

1) 确定扩展系数 δ，默认区和扩展区生成 Bucket，默认区的 Bucket 数量与 Reducer 的个数 n 相同，扩展区的 Bucket 数量为 $n\delta$。

2) Mapper 通过计算 $hash(key)\bmod(n+\delta)$ 获得写入数据的 Bucket 编号，若编号小于 n，则数据写入默认区，本次过程结束；若编号大于等于 n，则表示数据应放入扩展区，继续执行步骤 3)。

3) 对于 $hash(key)\bmod(n+\delta)\geqslant n$ 的情况，继续计算 $hash(key)\bmod(n\delta)$，确定扩展区中该数据所在的 Bucket 编号并填入数据。

4) 默认区中 Bucket 的数量与 Reducer 个数相同，按原生系统的处理方式根据编号直接建立映射关系，任意一个 Map 任务完成后各 Reducer 即开始数据拉取。扩展区的 Bucket 则暂不建立映射关系，交由后续的分区筛选算法和分区映射算法进行操作。

分区扩展算法的操作过程如算法 1 所示。

算法 1 扩展式数据分区算法

输入 默认区 *default*;
扩展区 *extension*;
Reducer 个数 *n*;
源数据 *key*;
扩展系数 δ;

1) *bukNo*←−1;//初始化
2) *default.creatBucket*(*n*);
3) *extension. creatBucket*($n\delta$);
4) *bukNo*= *hash*(*key*) mod ($n+\delta$);
5) if(*bukNo*<*n*) then
6) *write*(*key*, *default*[*bukNo*]);
7) else
8) *bukNo*= *hash*(*key*) mod ($n\delta$);
9) *write*(*key*, *extension*[*bukNo*]);
10) end if

算法中的默认区和扩展区只是个逻辑概念，并不存在界限分明的独立区域，数据写入方式也没有任何区别，仅是由于分区函数不同导致填充的数据不同。此外，由算法描述可以看出，大部分数据填充至默认区，而只有少量数据填入扩展区。这是由于全局数据分布是后续轮次分配的重要依据，而全局数据分布又随 Map 任务执行过程中逐步求精，因此，全局数据分布的使用时机越晚，其精确度越高，有利于提高后续分配的准确性。在分区扩展算法中，默认区为直接映射，又包含大部分数据，因此，扩展区的分配可以延后到任意 Reducer 默认区数据拉取完毕的时刻，即不影响 Shuffle 效率的最晚时间。

4.4 分区筛选算法

由于分区扩展算法生成的默认区为直接映射，分区筛选算法主要完成扩展区的分配工作，并以全局数据分布作为主要依据。根据 4.3 节的表述，扩展区第一轮分区筛选的执行时机确定为任意 Reducer 默认区数据拉取完毕的时间，这样不仅使分区筛选获得较为精确的全局数据分布，更最大限度提高了 Map 任务与拉取数据的重叠。而对于最后一轮筛选，依旧兼顾全局数据分布的精确度及 Shuffle 过程的执行效率，将执行时机定义为 Mapper 集合中仅有一个任务未完成的时间点。除首轮和末轮筛选外，其余 $\lambda-2$ 轮筛选则根据 Mapper 集合中任务完成的比例均分执行，即每 $\frac{m}{\lambda-1}$ 个 Map 任务完成触发一次分区筛选，Map 任务的完成情况从 Master 的统计数据中获取。此外，扩展区 Bucket 总个数为 $n\delta$，而分配轮数为 λ，因此，将 $n\delta$ 除以 λ 即可得到每轮筛选的 Bucket 数量，记为 C_n。

接下来基于算法元数据的相关定义，提出启发式的分区筛选算法。算法的主要步骤如下。

1) 获取当前时间点的全局数据分布 $\boldsymbol{B}$ 和分区映射矩阵 $\boldsymbol{BR}$。

2) 在分区映射矩阵查找符合条件 $\sum_{i\in rd} br_{ij}=0$ 的所有 Bucket，放入待选集合 *bkList*。

3) 根据全局数据分布 ***B*** 中的数据量，将 *bkList* 按数据总量倒序排列。

4) 从 *bkList* 中筛选前 C_n 个 Bucket 放入候选列表 *bkcandi*，并将 *bkcandi* 提交给分区映射算法。

分区筛选算法的操作过程如算法 2 所示。

算法 2　分区筛选算法

输入　全局数据分布 ***B***；

分区映射矩阵 ***BR***；

输出　候选列表 *bkcandi*；

1) *bkList*←*new List*<*Bucket*>; *bkcandi*←*new List*<*Bucket*>; //初始化

2) for *i*=0 to *s*−1 do

3)　for *j*=0 to *n*−1 do

4)　*sum*+=*BR*[*i*,*j*];

5) end for

6)　if *sum*==0 then　//筛选未分配 Bucket

7)　*bkList.add*(*Bucket*[*i*]);

8)　end if

9) end for

10) *bkList.sortBy*(***B***);　//根据数据量倒序排列

11) for *i*=0 to C_n−1 do　//取前 C_n 个 Bucket 填充候选列表

12) *bkcandi.add*(*bkList*[*i*]);

13) end for

14) return *bkcandi*;

4.5　分区映射算法

渐进填充（progressive filling）[41]是实现最大最小公平分配的理想算法，最早在网络资源分配中应用，现如今已成为云计算集群资源分配的标准解决方案。本文将这一思想引入分区映射算法以解决适应计算能力差异的 Shuffle 数据分配问题。

基于 3.3 节分配适应度的定义，对高效工作节点的 Reducer 做适度倾斜的数据分配，可获得最优的分配适应度，提高作业的执行效率。因此，如何评估 Reducer 的计算能力成为设计算法要解决的首要问题。由于计算能力依赖于计算机硬件和网络环境，硬件设备类型多样，网络环境纷繁复杂，针对硬件和网络环境的评估方法不仅复杂度高，也难以取得普适性结果。因此，算法并不致力于得出 Reducer 计算能力的具体值，只需生成按计算能力排序的 Reducer 序列，即可分区映射提供基本的分配依据。为此，算法采取一个简单易行又具有较高准确度的评估方案，基于 3 项评价指标：1) 作业执行的历史记录（4.2 节元数据中的历史排序集 *ES*）；2) 当前作业 Map 任务的完成情况；3) Reducer 已拉取的数据量。操作过程如下：首先，查找 MR 关联表，将 *flag* 为真的 Reducer 放入集合 *fastRT*，将 *flag* 为假的 Reducer 放入集合 *slowRT*；然后查找数据拉取向量，分别将集合 *fastRT* 和 *slowRT* 依据已拉取数据量顺序排列；其次，将 *fastRT* 的排序结果追加到 *slowRT*，形成基本排序集。最后，将基本排序集与历史排序集 *ES* 进行加权平均，从而生成最终排序集合 *RTC*。需要说明的是，*RTC* 与 *ES* 本身并不同构，因为 *RTC* 是基于 Reducer 生成排名，而 *ES* 是基于工作节点的排序集。由于 Reducer 与工作节点有固定的对应关系，因此，让 Reducer 自然继承节点计算能力排名，即可将 *ES* 转换为基于 Reducer 的集合，进而实现两者加权平均的操作。

算法将基本排序集与历史排序集做加权平均（目前采用等值平均，权重比为 1:1），目的是进一步提高计算能力评估的精度，历史排序集为节点排名的均值化结果，能够真实反映历史时间周期内的节点计算能力，而基本排序集则表示当前作业执行过程中 Reducer 的性能表现，因此，通过两者加权平均来评估现阶段 Reducer 计算能力排名的方法具有较高的准确性。而且随着分配轮数的增加，每轮分配时产生的 *RTC* 都是对前期计算能力排序的准确性进行逐步修正，从而为达到适应计算能力的数据分配目标奠定了良好的基础。

接下来求解 3.4 节带约束条件的优化目标，提出改进的贪婪算法解决每轮分区映射问题，从而在整体上为所有 Reducer 建立适应计算能力的数据分配。算法的主要步骤如下。

1) 获取当前时间点的全局数据分布、分区映射矩阵、数据拉取向量、MR 关联表和历史排序集，获取分区筛选算法的候选列表 *bkcandi*。

2) 基于数据拉取向量、MR 关联表和历史排序集，将 Reducer 按计算能力顺序排列，填充到集合 *RTC*。根据全局数据分布和分区映射矩阵，计算各 Reducer 已分配数据量，并将 Reducer 按已分配数据量顺序排列填充到集合 *RTD* 中。

3) 对所有 Reducer，获取其在 *RTC* 集合中的索引，用该索引值减去其在 *RTD* 中的索引值，得到索引差。

4) 遍历所有索引差，选择索引差最大的 Reducer，在 Reducer 内存容量许可的条件下，将 *bkcandi* 中最大的 Bucket 分配给该 Reducer。

5) 更新集合 *RTD*，重复步骤 3)和步骤 4)，直到 *bkcandi* 中所有 Bucket 分配完毕。

分区映射算法的操作过程如算法 3 所示。

算法 3 分区映射算法

输入 全局数据分布 ***B***；
分区映射矩阵 ***BR***；
数据拉取向量 ***F***；
MR 关联表 *MR*；
历史排序集 *ES*；
分区筛选候选列表 *bkcandi*；
Reducer 集合 *rd*；

输出 更新的分区映射矩阵 ***BR***；

1) *RTC*←*new List*<*Reducer*>; *RTD*←*new List*<*Reducer*>;*candi*←−1; //初始化

2) *RTC.fillBy*(*MR*,***F***,*ES*); //填充集合 *RTC*

3) *RTD.fillBy*(***B***,***BR***); //根据全局数据分布和分区映射矩阵填充集合 *RTD*

4) while(*bkcandi*!=*null*)

5) for *i*=0 to *rd.Length*−1 do

6) *ci*=*RTC.indexof*(*rd*[*i*]);

7) *di*=*RTD.indexof*(*rd*[*i*]);

8) if *candi*<*ci*-*di* then

9) *candi*=*i*;

10) end if

11) end for

12) *rd*[*candi*].*Load*=*compute*(***B***,***BR***); //计算该 Reducer 已分配的数据量

13) if *rd*[*candi*].*Load*+*bkcandi*[0]<A_{ir} then

14) *rd*[*candi*].*Load*+=*bkcandi*[0]; //空闲内存容量感知

15) else

16) *rd*[*candi*].*Load*+=*bkcandi.fit*(); //选择内存能够承载的最大 Bucket

17) end if

18) *update* (*RTD*, ***BR***);

19) end while

20) return ***BR***;

在分区映射算法中，每次选择计算能力排名和已分配数据量排名正向差异（指计算能力排名大于分配数据量排名）最大的 Reducer，在 Reducer 内存不会溢出的前提下，将分区筛选候选列表中最大的 Bucket 分配给该 Reducer。从整体上来看，每轮分区映射的目标都是提高 Reducer 数据量排名与计算能力排名的适宜度，通过多轮迭代渐进填充，最终完成适应 Reducer 计算能力的数据分配。

4.6 算法开销分析

首先分析算法的时间复杂度，从分区扩展算法的操作过程来看，算法步骤为常数次且均为简单操作，因此分区扩展算法的复杂度为 $O(1)$。分区筛选算法的复杂度由 2 个操作决定：1) 分区映射矩阵 ***BR*** 的遍历；2) *bkList* 集合的排序。分区映射矩阵 ***BR*** 共有 $C_n n$ 个元素，因此遍历过程的时间复杂度为 $O(C_n n)$，*bkList* 集合的排序采用堆排序法，时间复杂度为 $O(n\log n)$。在分区映射算法中，*RTC* 和 *RTD* 集合的排序方法也采用堆排序法，时间复杂度为 $O(n\log n)$；算法描述第 3、4 行循环嵌套的执行次数为 $C_n n$，时间复杂度为 $O(C_n n)$，其中，C_n 表示每轮筛选的分区数量，n 代表执行作业的 Reducer 个数，需要说明的是，可以将分区筛选算法 ***BR*** 遍历和分区映射算法嵌套循环的执行分派给集群空闲节点计算，当分配给 k 个工作节点计算时，只需要做一个简单的同步操作，可以将时间复杂度降低至 $O\left(\frac{C_n n}{k}\right)$。

算法的存储开销主要是元数据所占的存储空间，其中全局数据分布 ***B*** 和分区映射矩阵 ***BR*** 均有 $n\times s$ 个元素，对于数据拉取向量 ***F***、MR 关联表和历史排序集 *ES*，其元素个数均与 Reducer 数量相同。若算法的扩展系数 δ 设定 8，分配轮数 λ 为 4，由于元数据中的单个元素均为简单类型，因此即使在上千节点的大型集群上，矩阵 ***B*** 和 ***BR*** 也仅占 8 MB 的空间，而数据拉取向量 ***F***、MR 关联表和历史排序集各自仅占 1 MB 的空间。

算法的通信开销主要元数据的记录更新。根据算法存储开销的分析，元数据仅存储简单类型，每条记录的数据量很少，记录更新过程仅相当于一次平衡心跳，因此通信开销可以忽略不计。

由上述分析可以看出，PFPM 算法具有较低的时间复杂度，无附加通信开销，仅在 Master 上产生极微量的存储开销。因此，PFPM 算法完全适应任务密集的并行计算集群。

5 实验评价与比较

本节将通过实验进行比较和评价，验证渐进填

充分区映射算法的有效性。

5.1　实验环境

实验环境用 1 台服务器和 8 个工作节点建立计算集群，服务器作为 Spark 的 Master 和 Hadoop 的 NameNode。为体现工作节点的计算能力不同，8 个工作节点由 1 个高效节点、6 个普通节点和 1 个慢节点组成，其中，普通节点的配置如表 1 所示，高效节点配备 4 颗 CPU 阵列、64 GB 内存和 4 个千兆网卡，而慢任务节点仅有双核 CPU、2 GB 内存和 1 个百兆网卡。任务执行时间的数据来源于 Spark 的控制台，而内存使用状况的监控由 nmon 完成。在 Spark 框架下，任务的执行速度很快，通常会在几秒内完成，这并不利于本文准确监控任务执行时间和资源使用状况，因此，实验选择在小型集群上进行测试，使本文能够观察到作业执行的更多细节。

表 1　　**普通节点配置参数**

参数	值
CPU	Intel CORE i7/2.2 GHZ
RAM	4 GB
NIC	1 000 Mbit/s
Hard Disk	200 GB/SATA3.0(6 Gbit/s)
OS	Ubuntu 12.04
Spark	Apache Spark 1.6.2
Hadoop	Apache Hadoop 2.6
Scala	Scala-2.10.4
JDK	OpenJDK 1.8.0 25

实验数据选取 Zipf 数据集和有向图 2 种类型，其中，Zipf 数据集主要用于执行 WordCount 作业，总量为 8.6 GB，共包括 9 个子数据集，每个子数据集满足指数为 γ 的标准 Zipf 分布，γ 取值范围为 0.2~1.0 的小数，增量为 0.1，γ 的取值越大，表示数据分布越倾斜。有向图数据集主要用于执行 PageRank 作业，包括 SNAP 提供的 4 个标准数据集，如表 2 所示。

表 2　　**测试数据集**

数据名称	节点数	边数
Cit-Patents	3 774 768	16 518 948
soc-pokec-relationships	1 632 803	30 622 564
Wiki-Talk	2 394 385	5 021 410
Web-Google	875 713	5 105 039

5.2　分区映射算法

根据第 4 节的算法描述，通过渐进填充的分区映射，能够有效提高分配适应度，使分配数据量与节点计算能力更加匹配，从而优化作业执行效率。本节通过 3 个实验验证分区映射算法的有效性，主要从扩展系数、分配轮数、分配数据量和作业执行时间 4 个方面进行深入地评测和分析。

1) 扩展系数对比

分区映射算法的第 1 步是分区扩展，因此，首先进行扩展系数的对比实验。实验选取 2 个不同分布的数据集 Zipf-0.3 和 Zipf-0.7，执行 WordCount 作业，验证扩展系数对作业执行效率的影响，分配轮数取值与扩展系数相同，实验结果如图 4(a)所示。

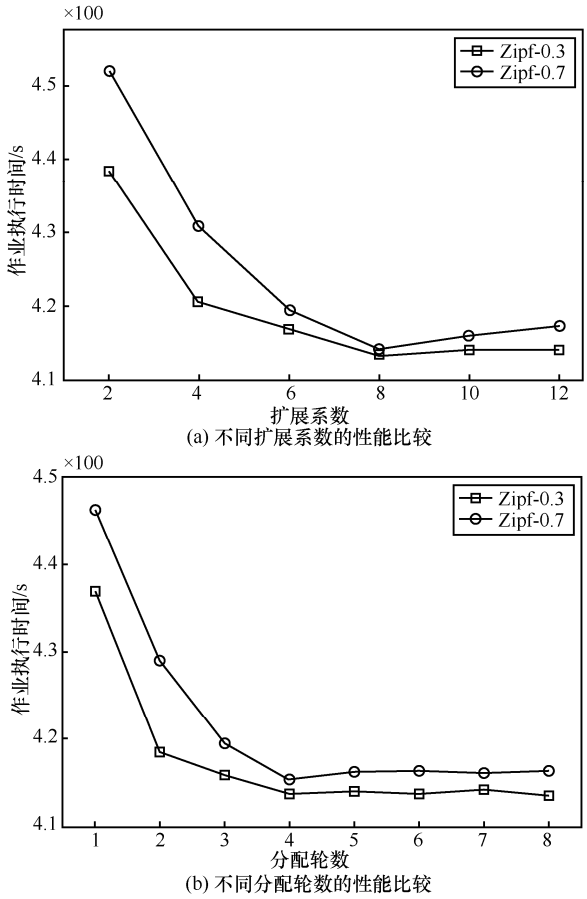

图 4　参数对比实验

由实验结果可以看出，不同数据分布条件下，扩展系数取值为 8 时趋于稳定。在前 4 个监测点，随着扩展系数的增大，2 个数据集的作业执行时间逐渐缩减，这是由于扩展系数越大，扩展区 Bucket 的粒度越小，分配轮数越多；而基于小粒度的多轮分配，能够及时修正过早分配产生的误差，提高映射数据量与

节点计算能力的匹配度，从而能够有效缩减作业执行时间。在后 2 个监测点，作业执行时间出现小幅上升，主要有以下 2 个原因：1) 扩展系数具有最优上限，在此基础上继续减小分区粒度、增加分配轮数也无法提高映射数据量与节点计算能力的匹配度，作业执行效率得不到任何优化；2) 扩展系数达到稳定最优值之后，多轮分区映射的附加开销开始逐渐显现，因此作业执行时间又出现了小幅上升。

2) 分配轮数对比

在分区扩展的基础上，渐进填充分区映射算法需要通过多轮分配，将扩展区数据映射到 Reducer，接下来通过实验评估分配轮数对作业执行性能的影响。实验仍然选择 Zipf-0.3 和 Zipf-0.7 数据集执行 WordCount 作业，扩展系数取值为 8，实验结果如图 4(b)所示。

由实验结果可以看出，在前几轮分配中，作业执行效率的优化效果较为明显，而执行时间的加速比随分配轮数的增大而递减。首先，扩展区的分配是在默认区映射及局部拉取之后，由于默认区数据量较大，当扩展区开始分配时，全局数据分布统计的准确度相对较高，因此，扩展区的首轮分配的优化效果最好。而随着分配轮数的增加，可修正的分配误差越来越小，后续轮次分配产生作业加速比也越来越小。其次，在分区筛选算法中，每轮分配总是选择数据量较大的 Bucket 加入候选列表。默认区映射完毕后，分配数据量排名与计算能力排名的差异性最大，通过扩展区首轮分配，映射容量较大的 Bucket，可以对分配数据量排名进行快速调整，而后续分配中映射 Bucket 的数据量越来越小，加上可修正误差越来越小，因此优化效果也逐渐减弱。从变化趋势上来看，分配轮数取值为 4 时，作业执行时间趋于稳定，继续增加分配轮数也无法进一步优化效率，与扩展系数的判定相同，分配轮数也同样具有最优上限。

3）性能对比及分配数据量监测

分区映射算法的最终目的是提高分配数据量与节点计算能力的适应度，缩减作业执行时间。实验选取 5 个 Zipf 数据集执行 WordCount 作业，进行作业执行时间和分配数据量 2 个方面的测评与分析。其中，Spark 的并行度参数为 16，PFPM 的扩展系数设定为 8，分配轮数为 4，作业执行时间对比如图 5 所示，高效节点和慢节点分配数据量对比如图 6 所示。

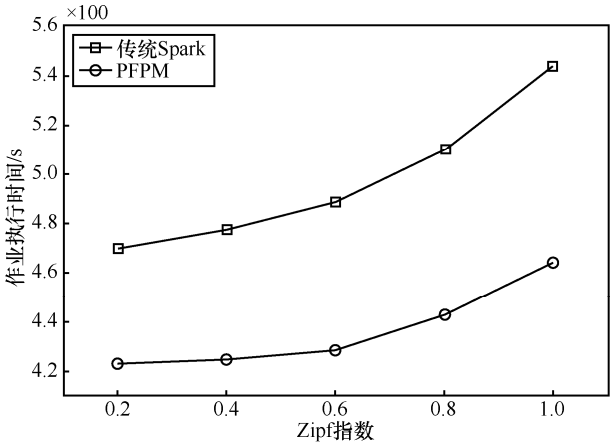

图 5 作业执行时间

由图 5 可以看出，在当前实验条件下，传统 Spark 的作业执行时间普遍高于 PFPM，这是由于 PFPM 算法在适应节点能力的数据量分配方面进行了优化，因此，作业执行效率较高。从整体变化趋势上来看，传统 Spark 和 PFPM 的作业执行时间都随数据分布指数增大而增大，数据分布越倾斜，作业执行时间越长。

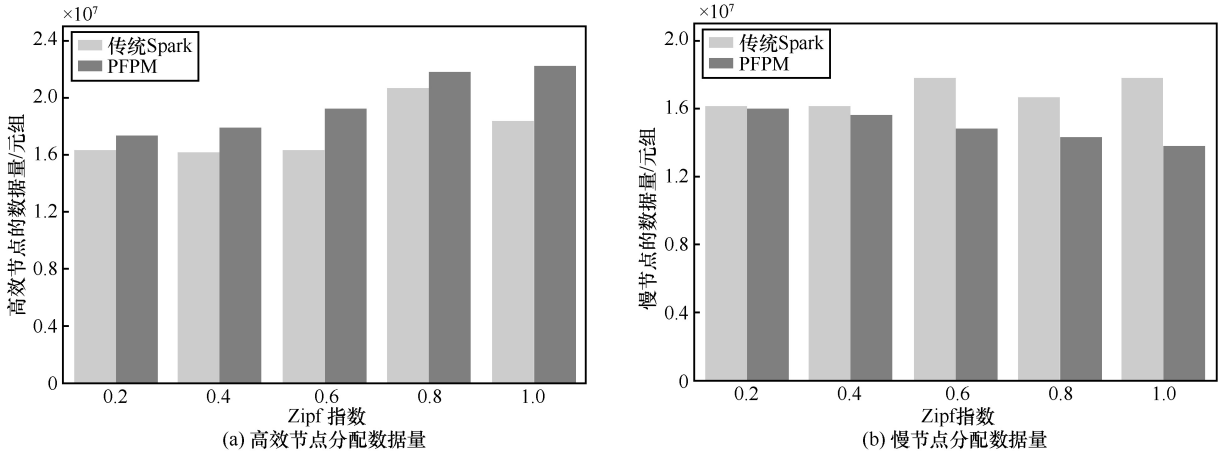

图 6 分配数据量

这是由于传统 Spark 的一次分区映射机制导致不合理的数据量分配，作业的同步开销增大，执行效率降低。而 PFPM 尽管可以提高分配数据量与节点计算能力的匹配度，但由于无法预知精确的数据分布，只能求得数据分配的近似最优解，多轮渐进填充也仅维护分配数据量排名与计算能力排名的一致性，并不能保证数据分配完全合理，因此，PFPM 的作业执行效率仍受数据分布指数的影响。从变化趋势的对比来看，传统 Spark 作业执行时间上升趋势较为明显，而 PFPM 的上升趋势较为平缓，这是因为传统 Spark 对于数据分布指数的敏感度更高，而 PFPM 能够在数据原始分布条件下，通过渐进填充不断调整数据分配，在一定程度上消除了数据倾斜分布的影响，因此，对数据分布指数的敏感度较低，适应性更强。

由图 6(a)可以看出，在高性能工作节点的数据分配上，传统 Spark 分配数据量并无明显规律，因为传统 Spark 的一次分区映射机制，数据分配既与原始数据的分布有关，又与 Reducer 注册的顺序有关。通过观察 PFPM 的数据分配发现，PFPM 在高性能节点上的数据分配量都大于传统 Spark，这是由于在任何数据分布指数条件下，PFPM 都会增加高效节点的数据分配量，以提高数据分配与节点计算能力的适应度。由图 6(b)可以看出，PFPM 在慢任务节点上的数据分配量都小于传统 Spark，由此证明 PFPM 的分区映射方案与节点计算能力相关。综合对比来看，PFPM 算法提高了高性能节点的数据分配量，而慢节点的数据分配量有所降低，但高性能节点增加的数据量与慢节点减少的数据量并不相等，这是因为随着集群负载不同、状态不同，节点计算能力也是动态变化的，在不同的时间点，工作节点的计算能力表现也不同，如某个系统服务运行或突发的网络访问，都会对节点的计算能力产生影响，因此，高性能节点的数据量增幅比与慢节点的缩减比不相同。

5.3　对比实验

数据分配的合理性最终反映到作业执行时间上，由于 WordCount 作业仅包含 1 个宽依赖同步操作，Reduce 任务也只进行简单的加法，作业执行效率的优化效果并不能完全体现出来，本节选用复杂度更高的 PageRank 算法，PageRank 的每轮迭代都包含 join 和 reduceByKey 这 2 个宽依赖操作，因此，更有利于验证算法的有效性。实验选择了 4 个不同大小的数据集测试算法性能，验证理论模型的正确性。传统 Spark 的并行度参数为 16，PFPM 的增量系数设定为 8，实验结果如图 7 所示。

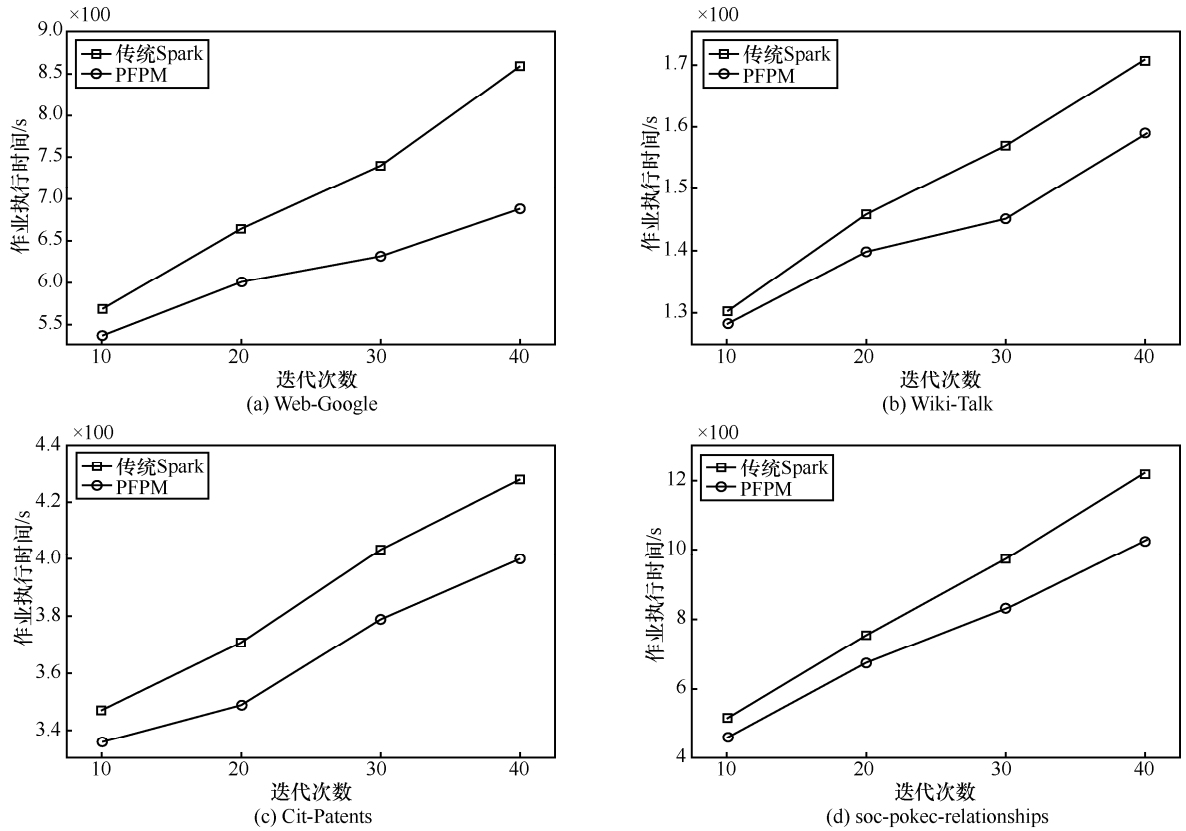

图 7　PageRank 作业的性能对比实验

由图 7 可以看出，对于每一个数据集，传统 Spark 与 PFPM 算法的作业执行时间都随迭代次数的增加而增长。PFPM 算法的作业执行时间明显小于传统 Spark，从而证明算法对 Spark 框架的性能具有优化效果。从作业执行的总体趋势来看，迭代次数越多，作业执行时间的优化效果越明显。从不同迭代次数的优化效果来看，作业执行时间的缩减比例基本随迭代次数增加呈线性增长，这是由于 PageRank 每轮的数据分布都相同，因此 PFPM 每轮迭代的优化效果也基本相同。从图 7 整体的对比结果来看，在不同数据集环境下，随着迭代次数的增加，传统 Spark 任务执行时间的上升趋势较为明显，而 PFPM 算法弱化 Stage 边界，提前进行分区映射和数据拉取，又通过多轮渐进填充建立适度倾斜的数据分配方案，因此任务执行时间的上升趋势相对缓和。由此可以看出，传统 Spark 对宽依赖操作的敏感度很高；而对于 PFPM，宽依赖操作越多，渐进填充分区映射的优势越能得以体现，作业执行的加速效应也越明显。

6 结束语

本文针对并行计算框架 Spark 原生分区映射方案所导致的作业延时问题，首先对作业的执行机制进行分析，建立执行效率模型，给出了 RDD 计算代价和作业执行时间的定义。通过分析宽依赖 RDD 的计算过程，建立了 Shuffle 过程模型，给出了数据分布、分区映射和分配适应度的定义，证明这些定义与作业执行效率的关系，为算法设计提供基础模型。在相关模型定义和证明的基础上，提出了渐进填充分区映射算法的问题定义，以此作为算法设计的主要依据。通过算法的问题定义求解，构建基础元数据，设计了分区扩展算法、分区筛选算法和分区映射算法。最后，通过不同的实验证明算法的有效性，实验结果表明，PFPM 算法优化了 Shuffle 过程的分区映射方案，提高了作业执行效率。

下一步工作主要集中在以下 2 个方面。

1) 异构集群多作业并发环境下，研究工作节点计算能力利用率最大化的数据分配策略。

2) 分析内存计算框架不同类型操作资源需求的一般规律，设计适应作业类型的数据分配和任务调度策略。

参考文献：

[1] 孟小峰, 慈祥. 大数据管理：概念、技术与挑战[J]. 计算机研究与发展, 2013, 50(1): 146-169.
MENG X F, CI X. Big data management: concepts, techniques and challenges[J]. Journal of Computer Research and Development, 2013, 50(1): 146-169.

[2] 付钰, 李洪成, 吴晓平, 等. 基于大数据分析的 APT 攻击检测研究综述[J]. 通信学报, 2015, 36(11): 1-14.
FU Y, LI H C, WU X P, et al. Detecting APT attacks: a survey from the perspective of big data analysis[J]. Journal on Communications, 2015, 36(11): 1-14.

[3] STRANDE S M, CICOTTI P, SINKOVITS R S, et al. Gordon: design, performance, and experiences deploying and supporting a data intensive supercomputer[C]//The 1st Conference on the Extreme Science and Engineering Discovery Environment. 2012: 1-8.

[4] 杜小勇, 陈峻, 陈跃国. 大数据探索式搜索研究[J]. 通信学报, 2015, 36(12): 77-88.
DU X Y, CHEN J, CHEN Y G. Exploratory search on big data[J]. Journal on Communications, 2015, 36(12): 77-88.

[5] ZAHARIA M, CHOWDHURY M, DAS T, et al. Fast and interactive analytics over hadoop data with spark [J]. Login, 2012, 37(4): 45-51.

[6] ZAHARIA M, XIN R, WENDELL P, et al. Apache Spark: a unified engine for big data processing[J]. Communications of the ACM, 2016, 59(11): 56-65.

[7] CARBONE P, EWEN S, HARIDI S, et al. Apache flink: stream and batch processing in a single engine[J]. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2015, 36(4): 28-38.

[8] TUMMALAPALLI S, MACHAVARAPU V R. Managing mysql cluster data using cloudera impala[J]. Procedia Computer Science, 2016, 85(5): 463-474.

[9] SIKKA V, LEHNER W, SANG K C, et al. Efficient transaction processing in SAP HANA database: the end of a column store myth[C]//The 2012 ACM SIGMOD International Conference on Management of Data. 2012: 731-742.

[10] DEAN J, GHEMAWAT S. MapReduce: simplifed data processing on large clusters[C]//The Conference on Operating System Design and Implementation (OSDI). 2004: 137-150.

[11] ZAHARIA M, CHOWDHURY M, DAS T, et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing[C]//The 9th USENIX Conference on Networked Systems Design and Implementation.2012:2.

[12] LIN X, WANG P, WU B. Log analysis in cloud computing environment with hadoop and spark[C]//The 5th IEEE International Conference on Broadband Network & Multimedia Technology (IC-BNMT). 2013:273-276.

[13] DONG X, XIE Y, MURALIMANOHAR N, et al. Hybrid checkpointing using emerging nonvolatile memories for future exascale system[J]. ACM Transactions on Architecture and Code Optimization (TACO), 2011, 8(2): 1-29.

[14] 田俊峰, 张亚姣. 基于马尔可夫的检查点可信评估方法[J]. 通信学报, 2015, 36(1): 234-240.
TIAN J F, ZHANG Y J. Checkpoint trust evaluation method based on Markov[J]. Journal on Communications, 2015, 36(1): 234-240.

[15] ARMBRUST M, XIN R S, LIAN C, et al. Spark SQL: relational data processing in spark[C]//The 2015 ACM SIGMOD International Conference on Management of Data. 2015: 1383-1394.

[16] IQBAL M H, SOOMRO T R. Big data analysis: apache storm perspective[J]. International Journal of Computer Trends & Technology, 2015, 19(1): 9-14.

[17] ZAHARIA M, DAS T, LI H Y, et al. Discretized streams: fault-tolerant streaming computation at scale[C]//ACM Symposium on Operating Systems Principles. 2013:423-438.

[18] MENG X, BRADLEY J, YAVUZ B, et al. MLlib: machine learning in apache Spark[J]. Journal of Machine Learning Research, 2015, 17(1): 1235-1241.

[19] GONZALEZ J E, XIN R S, DAVE A, et al. GraphX: graph processing in a distributed dataflow framework[C]//The 11th USENIX conference on Operating Systems Design and Implementation. 2014: 599-613.

[20] 廖彬, 于炯, 孙华, 等. 基于存储结构重配置的分布式存储系统节能算法[J]. 计算机研究与发展, 2013, 50(1): 3-18.
LIAO B, YU J, SUN H, et al. Energy-efficient algorithms for distributed storage system based on data storage structure reconfiguration[J]. Journal of Computer Research and Development, 2013, 50(1): 3-18.

[21] DEAN J, GHEMAWAT S. MapReduce: simplified data processing on large clusters[J]. Operating Systems Design & Implementation, 2004, 5(1): 147-152.

[22] KWON Y, BALAZINSKA M, HOWE B, et al. A study of skew in MapReduce application[J]. Open Cirrus Summit, 2011, 1:1-5.

[23] KWON Y, BALAZINSKA M, HOWE B, et al. Skew-resistant parallel processing of feature-extracting scientific user-defined functions[C]//The 1st ACM Symposium on Cloud Computing. 2010: 75-86.

[24] 王卓, 陈群, 李战怀, 等. 基于增量式分区策略的 MapReduce 数据均衡方法[J]. 计算机学报, 2016, 39(1): 19-35.
WANG Z, CHEN Q, LI Z H, et al. An incremental partitioning strategy for data balance on MapReduce[J]. Chinese Journal of Computers, 2016, 39(1): 19-35.

[25] KWON Y, BALAZINSKA M, HOWE B, et al. SkewTune: mitigating skew in MapReduce applications[C]//The 2012 ACM SIGMOD International Conference on Management of Data. 2012: 25-36.

[26] YAN W, XUE Y, MALIN B. Scalable and robust key group size estimation for reducer load balancing in MapReduce[C]//IEEE Int Conference on Big Data. 2013: 156-162.

[27] RAMAKRISHNAN S R, SWART G, URMANOV A, et al. Balancing reducer skew in MapReduce workloads using progressive sampling[C]//The 3rd ACM Symposium on Cloud Computing (SOCC'12). 2012: 1-14.

[28] GUFLER B, AUGSTEN N, REISER A, et al. Handing data skew in MapReduce[C]//The 1st International Conference on Cloud Computing and Services Science. 2011: 574-583.

[29] GUFLER B, AUGSTEN N, REISER A, et al. Load balancing in MapReduce based on scalable cardinality estimates[C]//The 28th IEEE International Conference on Data Engineering (ICDE). 2012: 522-533.

[30] TANG Z, ZHANG X S, LI K, et al. An intermediate data placement algorithm for load balancing in Spark computing[J]. Future Generation Computer Systems, 2016.

[31] KOLB L, THOR A, RAHM E. Load balancing for MapReduce-based entity resolution[C]//The 28th IEEE International Conference on Big Data Engineering (ICDE). 2012: 618-629.

[32] KOLB L, THOR A, RAHM E, et al. Block-based load balancing for entity resolution with MapReduce[C]//The 20th ACM International Conference on Information and Knowledge Management (CIKM). 2011: 2397-2400.

[33] CHEN Q, YAO J Y, XIAO Z. Libra: lightweight data skew mitigation in MapReduce[J]. IEEE Transactions on Parallel & Distributed Systems, 2015, 26(9): 2520-2533.

[34] RACHA S C. Load balancing MapReduce communications for efficient executions of applications in a cloud[M]. India, Bangalore: Indian Institute of Science, 2012:12-16.

[35] IBRAHIM S, JIN H, LU L, et al. Handling partitioning skew in MapReduce using LEEN[J]. Peer-to-Peer Networking and Applications, 2013, 6(4): 409-424.

[36] DAI W, IBRAHIM I, BASSIOUNI M. Improving load balance for data-intensive computing on cloud platforms[C]//2016 IEEE International Conference on Smart Cloud. 2016: 140-145.

[37] TANG Z, ZHANG X S, LI K L, et al. A data skew oriented reduce placement algorithm based on sampling[J]. IEEE Transactions on Cloud Computing, 2016.

[38] FAN Y Q, WU W G, XU Y L, et al. Improving MapReduce performance by balancing skewed loads[J]. Communications, 2014, 11(8): 85-108.

[39] TRIGUERO I, GALAR M, VLUYMANS S. Evolutionary undersampling for extremely imbalanced big data classification under apache spark[C]//2016 IEEE Congress on Evolutionary Computation. 2016: 715-722.

[40] MESTRE D G, PIRES C E, NASCIMENTO D C, et al. An efficient spark-based adaptive windowing for entity matching[J]. Journal of Systems and Software, 2017, 128(6): 1-10.

[41] GHODSI A, ZAHARIA M, SHENKER S, et al. Choosy: max-min fair sharing for datacenter jobs with constraints[C]//The 8th ACM European Conference on Computer Systems. 2013:365-378.

作者简介：

卞琛（1981-），男，江苏南京人，博士，新疆大学副教授，主要研究方向为内存计算、分布式系统等。

于炯（1964-），男，北京人，新疆大学教授、博士生导师，主要研究方向为网格计算、大数据与云计算等。

修位蓉（1979-），女，四川宜宾人，新疆大学讲师，主要研究方向为数据挖掘、大数据分析等。

廖彬（1986-），男，四川内江人，博士，新疆财经大学副教授，主要研究方向为绿色计算、数据库系统理论及数据挖掘等。

英昌甜（1989-），女，新疆乌鲁木齐人，新疆大学博士生，主要研究方向为内存计算、大数据等。

钱育蓉（1980-），女，新疆乌鲁木齐人，博士，新疆大学副教授，主要研究方向为云计算、图像处理等。