面向 Flink 迭代计算的高效容错处理技术

郭文鹏[1)] 赵宇海[1)] 王国仁[2)] 韦刘国[1)]

[1)](东北大学计算机科学与工程学院 沈阳 110169)

[2)](北京理工大学计算机学院 北京 100081)

摘 要 迭代计算是相同逻辑的重复执行,在各种机器学习和数据挖掘方法中被广泛使用.在大数据的处理与分析领域中,分布式迭代计算更是当前的热点研究问题之一.容错机制是分布式系统高可用性的必要保证.现有分布式系统的容错机制虽然在高可用性上表现良好,但忽略了面向迭代计算的容错效率问题.本文针对批流混合大数据计算系统 Apache Flink 的迭代容错效率问题,进行了系统的研究.执行流处理任务时,Flink 采用"分布式快照"的检查点机制来完成容错.对于海量数据的迭代分析,检查点增加了不必要的延迟.执行批处理任务时,Flink 采用从头执行任务的方式来实现容错,该方式虽然实现简单,但带来了很大的时间开销.针对以上问题,本文首先提出了一种基于补偿函数的乐观迭代容错机制.该容错机制在迭代任务发生故障时采用乐观补偿的思想恢复任务,在迭代执行过程中不采用任何额外的容错手段(不会引入额外的容错开销),采用用户自定义的补偿函数收集健康节点上的迭代数据,并结合初始的迭代数据对故障节点上丢失的分区数据进行恢复,继续执行至迭代收敛状态,保证了迭代任务的高效顺利执行.由于乐观迭代容错机制并不保证得到的结果与无故障执行得到的结果完全一致,因此针对精度要求较高的迭代任务,本文结合 Flink 系统的迭代数据流模型,进一步提出一种基于头尾检查点悲观迭代容错机制.与传统的阻塞检查点(阻塞下游操作符)的工作方式不同,该容错机制以非阻塞的方式编写检查点,充分结合 Flink 迭代数据流的特点,将可变数据集的检查点注入迭代流本身.通过设计迭代感知,简化了系统架构,降低了检查点成本和故障恢复时间.本文基于 Flink 系统,在大量的真实数据集和模拟数据集上,从增量迭代和全量迭代两方面对提出的两种容错机制进行了全面的实验研究,验证了本文提出的迭代容错优化技术的高效性.实验结果证实,本文基于 Flink 系统提出的乐观容错机制和悲观容错机制在计算效率上均优于现有的分布式迭代容错机制.前者在全量迭代计算任务中运行时间最高可提升 22.8%,在增量迭代计算任务中最高可提升 33.8%;后者在全量迭代任务中最高可节省 15.3%的时间开销,在增量迭代任务中最高可节省 18.5%的时间开销.

关键词 分布式迭代计算;Apache Flink;乐观容错;悲观容错;检查点

中图法分类号 TP18 **DOI 号** 10.11897/SP.J.1016.2020.02101

Efficient Fault-Tolerant Processing Technology for Flink Iterative Computing

GUO Wen-Peng[1)] ZHAO Yu-Hai[1)] WANG Guo-Ren[2)] WEI Liu-Guo[1)]

[1)](*School of Computer Science and Engineering*, *Northeastern University*, *Shenyang* 110169)

[2)](*School of Computer Science and Technology*, *Beijing Institute of Technology University*, *Beijing* 100081)

Abstract Iterative calculation is the repeated execution of the same logic and is widely used in various machine learning and data mining methods. In the field of big data processing and analysis, distributed iterative computing is one of the current hot research issues. Fault tolerance is a necessary guarantee for high availability of distributed systems. Although the fault tolerance mechanism of existing distributed systems performs well in high availability, it ignores the problem of fault tolerance efficiency for iterative computing. This paper systematically studies the iterative

收稿日期:2019-09-02;在线发布日期:2020-02-13.本课题得到科技部重点研发项目"云计算和大数据"重点专项项目(2018YFB1004402)和国家自然科学基金(61772124)资助.**郭文鹏**,硕士研究生,主要研究方向为大数据.E-mail:17628822324@qq.com.**赵宇海**(通信作者),博士,教授,中国计算机学会(CCF)高级会员,主要研究领域为数据挖掘.E-mail:zhaoyuhai@mail.neu.edu.cn.**王国仁**,博士,教授,中国计算机学会(CCF)高级会员,主要研究领域为数据库.**韦刘国**,硕士研究生,主要研究方向为大数据.

fault-tolerant efficiency of batch-flow hybrid big data computing system Apache Flink. When performing stream processing tasks, Flink uses a "distributed snapshot" checkpoint mechanism to complete fault tolerance. For iterative analysis of massive data, checkpoints add unnecessary delay. When performing batch processing tasks, Flink uses the task execution method from the beginning to achieve fault tolerance. Although this method is simple to implement, it brings a lot of time overhead. In view of the above problems, this paper first proposes an optimistic iterative fault tolerance mechanism based on compensation functions. This fault-tolerant mechanism uses optimistic compensation to recover tasks when iterative tasks fail. It does not use any additional fault-tolerant methods (it does not introduce additional fault-tolerant overhead) during iterative execution, and uses user-defined compensation functions to collect healthy nodes. Iterative data, combined with the initial iterative data, recovers the lost partition data on the failed node, and continues execution to the iterative convergence state, ensuring the efficient and smooth execution of the iterative task. Because the optimistic iterative fault tolerance mechanism does not guarantee that the results obtained are completely consistent with the results obtained by fault-free execution, for the iteration tasks with higher accuracy requirements, this paper combines the iterative data flow model of the Flink system to further propose a head-to-tail checkpoint. Pessimistic iterative fault tolerance mechanism. Unlike traditional blocking checkpoints (blocking downstream operators), this fault-tolerant mechanism writes checkpoints in a non-blocking manner, fully combines the characteristics of Flink iterative data flow, and injects variable data set checkpoints into the iterative flow itself. By designing iterative awareness, the system architecture is simplified, and checkpoint costs and failure recovery times are reduced. This paper is based on the Flink system. On a large number of real data sets and simulated data sets, a comprehensive experimental study of the two proposed fault tolerance mechanisms from the aspects of incremental iteration and full iteration is conducted, and the effectiveness of the proposed iterative fault tolerance optimization technology is verified. Efficiency. The experimental results confirm that the optimistic and pessimistic fault-tolerant mechanisms proposed in this paper based on the Flink system are superior to the existing distributed iterative fault-tolerant mechanisms in terms of computational efficiency. The former can increase the running time by up to 22.8% in full iterative computing tasks and up to 33.8% in incremental iterative computing tasks; the latter can save up to 15.3% of the time overhead in full iterative tasks, and in incremental iterative tasks Saves up to 18.5% of time.

Keywords distributed iterative calculation; Apache Flink; optimistic fault tolerance; pessimistic fault tolerance; checkpoint

1 引 言

迭代计算通常是数据挖掘和机器学习算法的核心部分,在各类应用中都普遍存在[1].在搜索领域,由Google提出的著名的网页排序算法PageRank[2-3],其核心思想就是根据网络之中不同网页之间的链接关系进行迭代计算[4],最终的排名即是迭代最终收敛的值或重要性;在社交网络[5-7]领域,很多好友推荐算法都是通过利用现有用户的好友关系网络图通过迭代计算来挖掘用户之间可能存在的潜在链接关系;Random Walk[8-9]算法通过迭代计算来求解图中某节点到其它节点的概率;在影音推荐领域,常用的推荐算法是按照用户的喜好来对用户进行聚类,然后向用户推荐同类用户所喜欢的影音资源,这类方法统称为协同过滤推荐[10-11].其中基于矩阵分解的协同过滤算法,如交替最小二乘法(ALS)和奇异值分解等(SVD)等都包含迭代计算;在图论

中的连通分支算法也是基于迭代实现的. 在数据分析领域中，常用的 K-Means[12] 聚类算法、联合聚类、点对聚类等都包含迭代计算，每次迭代时更新顶点和模型参数的状态，直到满足收敛或停止标准.

随着数据的规模日益增长，分布式迭代计算成为大数据处理与分析的研究热点之一. 近年来，流行的大数据处理平台 Hadoop[13]、Spark[14] 和 Flink[15] 都具备处理迭代任务的能力. 现有的分布式迭代包含全量迭代和增量迭代两种. 全量迭代总是重新计算迭代的中间结果. 然而许多情况下，迭代任务的中间状态会以不同的速度汇聚. 例如，在大图的单源最短路径的计算中. 在这种情况下，系统总是重新计算整个中间状态包括不再变化的部分，从而导致资源浪费. 增量迭代可以有效地缓解该问题. 该模式使用两个数据迭代状态模拟迭代计算：解集和工作集. 解集保存当前中间结果，而工作集保存解集的更新结果. 在增量迭代期间，系统使用工作集有选择地更新解集的元素，并根据更新计算下一个工作集. 一旦工作集变空，迭代就会终止. 无论是全量迭代还是增量迭代，对海量数据而言，都是极其耗时的，并且消耗大量的计算资源.

由于分布式计算通常涉及大量计算节点的协同工作，容错性是分布式系统高可用性的必要保证. 主流的分布式大数据平台针对迭代计算任务采取了不同的容错机制. 分布式系统 Hadoop 的迭代容错机制主要是通过检查点机制的方式实现，在每个计算的结束阶段设置检查点，发生故障时从检查点读取数据重新执行. 反复从底层文件系统中读取数据会造成大量的磁盘 IO 开销. 分布式系统 Spark 框架中的 SparkStreaming 采用记录更新的手段实现容错，通过 Lineage[16] 技术来实现. 对于窄依赖（父 RDD 的每个分区只被子 RDD 的一个分区所引用）的数据因发生故障丢失时，只需要对丢失的那一部分数据进行恢复并重新计算；对于宽依赖（父 RDD 的每个分区可能被多个子 RDD 引用）则必须将其祖先 RDD 中的所有数据块全部恢复并重新进行计算. 在宽依赖场景下 Spark 引入了检查点机制，在执行过程中选取适当的时机备份，通过缩短 Lineage 链长度来减少容错开销. 但在执行过程中，频繁的数据备份操作也会产生极大的网络和磁盘 IO 开销. 分布式系统 Flink 系统现有的批处理和流处理的容错分别采用了逆向恢复容错技术和前向恢复技术. 批处理容错机制是当 Job 失败时通过使用重启策略对整个 Job 重启. 流处理容错机制是基于状态一致的分布式快照实现的，这些快照保存了执行图中所有算子及传输通道的状态，这些轻量级快照也可以被视做一种另类的检查点. Flink 的分布式快照采用 Chandy-Lamport[17] 算法实现. 该容错机制虽然高效，但需要额外的检查点协调者来实现，管理复杂，且会带来额外的写入开销. Flink 虽然针对其流处理也可以进行迭代计算，在实际应用场景下，大部分的迭代计算任务还是基于批处理执行. 综上，现有分布式系统的容错机制大多面向通用的计算任务，在迭代任务上的容错效率较低，没有结合迭代计算任务的特点，代价开销较大.

传统分布式系统的容错机制大多采用了悲观的检查点容错机制. 通过缓存管理检查点的实现，与流水线数据流无关. 然而，这些检查点是以阻塞的方式备份数据，开销较大，且忽略了迭代处理的迭代控制，这使系统设计复杂化，因为需要额外的组件来管理检查点. 此外，以分布式方式在海量数据集上执行迭代算法，算法的中间结果必须在机器之间进行分区存储. 执行失败将导致丢失这些分区的子集，要继续执行，系统必须首先恢复丢失的数据. 发生故障时，系统将暂停执行，从先前写入的检查点恢复一致状态并继续执行. 这种方法的缺点是，即使在无故障的情况下，它也会给执行带来开销. 对于海量数据的迭代算法，检查点不必要增加了计算的延迟. 现有分布式系统缺少了乐观的容错机制，悲观的容错机制忽略了分布式迭代数据流的特点，以阻塞的方式实现容错，代价开销大，容错效率低.

针对现有分布式系统迭代容错机制的不足，本文面向大规模分布式迭代计算任务主要贡献有：

（1）提出了基于补偿函数的乐观容错机制. 该容错机制在迭代执行过程中不采用任何额外的容错手段（不会引入额外的容错开销），在发生故障时，采用用户自定义的补偿函数收集健康节点上的迭代数据，并结合初始的迭代数据对丢失的分区数据进行恢复，保证了迭代任务高效顺利的执行.

（2）提出了一种基于头尾检查点的悲观容错机制. 与传统的阻塞检查点不同，该容错机制以无阻塞的方式编写检查点，不会破坏流水线操作任务. 将可变数据集的检查点注入迭代数据流本身，简化系统架构并有助于在迭代处理期间检查点的创建.

（3）将提出的面向迭代任务的乐观补偿函数容错机制和悲观的头尾检查点机制基于高度创新的开源流处理器 Flink 进行实现. 乐观的补偿函数机制

的实现设置了收集数据和补偿恢复丢失数据的接口,可供用户直接使用.头尾检查点机制的实现,在Flink迭代处理框架中新增了头尾检查点选项,用户可根据迭代任务类型直接选择.

(4)在真实数据集和模拟数据集上从全量迭代和增量迭代方面进行了系统的实验研究,验证了本文提出的容错技术的高效性.

本文第2节介绍相关工作;第3节介绍文中涉及的基本概念;第4节描述基于补偿函数的乐观容错机制;第5节介绍基于头尾检查点的悲观容错机制;第6节为实验分析部分;第7节总结了本文的工作.

2 相关工作

目前,国内外被各大企业和研究机构所使用的分布式计算系统主要包括批处理系统,如Hadoop.流处理系统,如Storm和Samza.混合处理系统(既支持批处理又支持流处理的框架),如Spark和Flink.各个计算系统都具有自己独特的容错机制,但总体上可以将这些容错机制分为两类,一类是基于检查点的容错机制,一类是基于记录更新的容错机制.

分布式系统环境下,假如某个计算节点出现故障,集群和任务将进入故障,容错恢复的目标是采取相应的措施,将任务和系统转换到正确执行的状态.分布式容错恢复技术整体上包括了前向恢复技术(Forward Recovery)[18]和逆向恢复(Backward Recovery)技术[19].文献[20]中提出了一种分布式日志恢复系统的数据恢复方法.三阶段提交协议比两阶段提交协议能更好地实现分布式事务执行的无阻塞,在分布式数据库发生故障时可以有效地恢复分布式数据库中的数据,保证了分布式日志恢复系统的高可用性和高可靠性.文献[21]中研究了分布式系统下基于检查点的容错服务,利用系统失效关联性特征来建立模型,得到减小分布式任务的完成时间的检查点放置策略,从而在保证系统可靠性的前提下,降低容错服务的实现代价,提高分布式系统的运行效率.Dudoladov等人在文献[22]中提出了一种使用算法补救的容错机制,该机制利用了数据挖掘和机器学习中使用大量虚拟算法的鲁棒性、自校正性,这些算法从各种中间一致状态收敛到正确的解.该函数在算法上创建这样的一致状态,而不是回滚到先前的检查点状态.该优化机制不会检查任何状态,并且在保证容错所需的开销方面具有最佳的无故障性能.

在过去几年中,出现了许多优化的迭代计算系统.像Twister[23]或HaLoop[24]这样的MapReduce[13]扩展,以及像Spark这样的系统能够有效地执行某类迭代算法,Flink系统在全量迭代的基础上,新增了增量迭代功能.增量迭代是通过部分计算取代全量计算,在计算过程中会将数据分为热点数据和非热点数据,每次迭代计算会针对热点数据展开,这种模式适合用于数据量比较大的计算场景,不需要对全部的输入数据集进行计算,所以在性能和速度上都会有很大的提升.而这些分布式计算系统的容错机制大多以悲观的方式实现,且面向的通用的计算任务.对于迭代任务而言,容错开销大,计算效率低.缺乏了对乐观容错机制的设计,且采用的悲观容错机制多数以阻塞的方式实现.没有结合迭代数据流的特点,引入了不必要的额外开销.此外,现有的分布式系统的检查点机制基于阻塞的方式来实现,即通过阻塞下游操作符等待完整数据的到来.以阻塞方式实现检查点产生了较大的时间开销.针对这些问题,本文提出了面向大规模迭代计算处理的高效容错技术.由于Flink作为高效的开源批处理器,在批处理和流处理的计算效率上均优于Hadoop、Storm和Spark[25].本文基于Flink系统实现了基于补偿函数的乐观容错机制和基于头尾检查点的悲观容错机制,并在大量数据集上从全量迭代和增量迭代方面进行了实验研究与分析.

3 基本概念

本节给出文中涉及到的一些基本概念并对要解决的问题给出形式化描述.

定义1(步函数). 在迭代计算中,对每一轮迭代输入的数据集进行转换操作的函数称为步函数(Step Function).

例如,PageRank全量迭代算法在执行时,每一轮迭代都要对顶点的rank值进行更新.更新rank值的操作有Join、Filter等操作符,这些操作符构成的数据流图即为该算法的步函数.

定义2(超级步). 在迭代计算中,从迭代输入开始,经过步函数的转换,到更新为下一轮迭代输入的整个流程称为超级步.

如图1为Flink官方文档展示的迭代超级步的粒度即同步的粒度.步函数是超级步的组成部分,从第一个超级步迭代输入开始,经过步函数的转换并

更新为第二个超级步的输入. 该流程即为一个超级步.

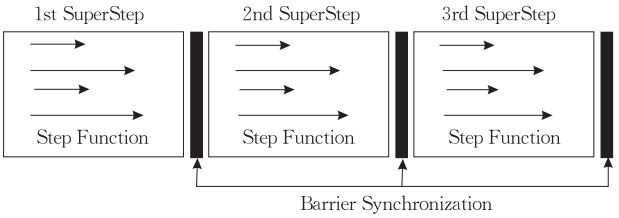


图 1　迭代超级步示意图

定义 3(乐观容错机制).　在分布式系统环境下,如果某个计算节点出现故障,则采取相应的容错机制来把系统恢复到无错误状态. 乐观容错机制采用乐观的态度,即假定所有的故障及其恢复策略都事先已知. 当系统发现错误后,试图把系统带入一个新状态. 该机制要求系统事先掌握可能出现的故障.

例如,本文提出的乐观补偿恢复机制,在迭代任务执行之前,需要对分布式系统可能出现故障丢失数据的情况进行采取补偿恢复措施. 如果任务没有出现故障,则顺利完成执行. 如果出现故障则采用预先定义的补偿措施对故障进行恢复,使用恢复后的数据继续迭代的执行.

定义 4(悲观容错机制).　悲观容错机制与定义 3 的乐观容错机制相反,该机制采用的是悲观的思想,假定故障的产生是未知的,在系统执行任务的过程中定期保存一些结果和历史记录信息. 一旦发生故障,则可恢复到最新记录的状态.

例如,Flink 系统采用的分布式快照机制就是一种悲观的容错机制,如图 2 所示,以固定的时间间隔将 barrier 注入数据流,进行 Checkpoint 备份. 当出现故障时,恢复到最近的 Checkpoint 时的状态.

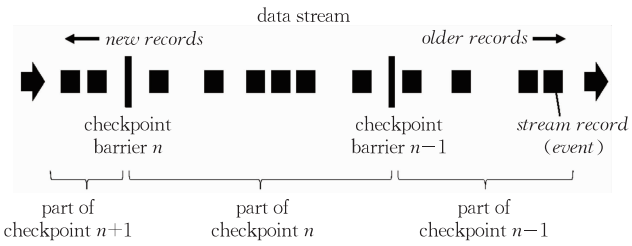


图 2　Flink 系统分布式快照示意图

定义 5(全量迭代).　全量迭代计算过程如图 3 所示,在数据流接入迭代算子的过程中,步函数每次都会处理全量的数据,然后计算下一次迭代的输入,即图中的 NextPartialSolution,最后根据触发条件输出迭代计算的结果.

例如,给定一组数据,迭代步函数为迭代数据加

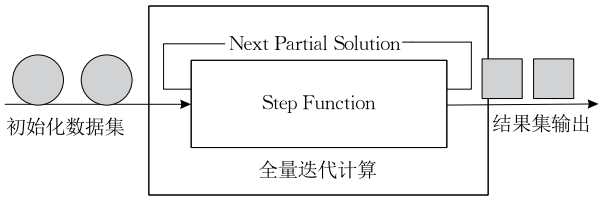


图 3　全量迭代计算示意图

1,输出迭代 10 次后的结果. 在迭代过程中,每一轮迭代都要对所有的数据进行加 1 操作. 这种对全量数据进行转换的迭代过程即为全量迭代.

定义 6(增量迭代).　增量迭代计算过程如图 4 所示,通过部分计算取代全量计算,在计算过程中会将数据集分为热点数据和非热点数据,每次迭代计算会针对热点数据展开,不需要对全部的输入数据集进行计算.

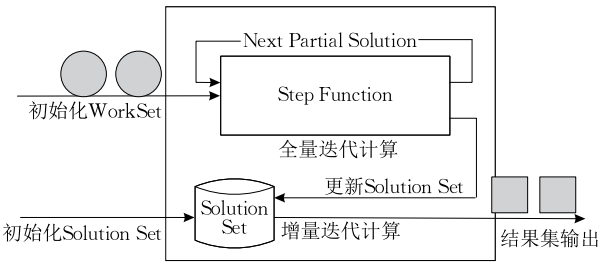


图 4　增量迭代计算示意图

例如,图论中的连通分支算法,该算法用于求解图的连通性问题. 迭代过程为:首先,初始化每个顶点所属的分类值(即所属的连通分量组),每个顶点初始值等于该顶点值的 Id. 其次,每个顶点搜索其相邻的顶点,如果顶点的分类值小于该顶点的分类值,则更新该顶点的分类值,并在连通图中传播. 最后,当没有顶点需要更新时,所有连通图包含的顶点具有相同的分类值,算法结束. 这种只需对部分数据进行转换的操作即为增量迭代.

4　基于补偿函数的乐观容错机制

本节针对现有分布式计算系统悲观容错的迭代容错机制额外开销大,需要额外的组件控制检查点协同者,实现复杂等特点. 提出了一种面向迭代任务的乐观补偿函数容错机制. 该容错机制在迭代执行过程中,不会引入任何额外的开销. 如果出现故障,则采用用户自定义的补偿函数收集健康节点上的数据,并结合初始迭代数据对丢失的分区数据进行恢复. 现有的分布式系统未引入乐观恢复机制的原因之一在于,补偿函数完全由用户编写,实现难度大.

本节的乐观补偿函数机制在实现时为用户提供了收集数据和恢复数据的接口，可供用户直接使用.该机制保证了迭代任务高效顺利的执行，得到的结果可以收敛到无故障执行时的近似状态.

本节首先分析了分布式迭代的收敛性，证明了补偿函数的乐观容错机制的正确性.然后从全量迭代和增量迭代的角度实现了补偿函数容错机制.

4.1 分布式迭代计算的收敛性

在计算机科学中，迭代是对一段程序的反复执行.迭代可以表示一种状态，该状态以可变重复的形式存在.迭代计算是数学领域中的常见的计算方式，常见的应用有矩阵求解特征值问题以及方程组求解等问题.迭代的求解思路是不断趋近，选择一个粗略的初始值，采用迭代公式不断地更新该值，如果该值满足收敛条件即精度满足或者迭代次数满足则终止.否则，将继续更新和计算该值.大数据系统的分布式迭代数据流示意图如图5所示.

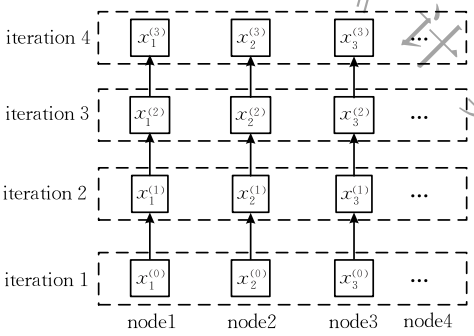


图5 Flink系统中的分布式迭代数据流

其中，分布式迭代计算可以表示为

$$x^k = f(x^{(k-1)}),\ k = 1,2,\cdots,n \tag{1}$$

其中变量 x 表示的迭代过程中不断更新计算的数值，即迭代变量.公式左边的 x^k 表示迭代计算到第 k 次时的值，公式右边 f 是第 k 次迭代结果计算的通用表达式，即迭代函数.分布式迭代算法执行过程中，每一轮迭代所需的数据可以表示为 x^k，记为迭代变量.可以使用向量 $\boldsymbol{R}^n=(x_1^{(k)},x_2^{(k)},\cdots,x_n^{(k)})$ 来表示迭代变量，其中每个 $x_i^{(k)}$ 都是一个迭代元素.f 为 $\boldsymbol{R}^n$ 的映射，f 是需要反复执行的一系列操作集合 $(f_1,f_2,\cdots,f_n)$，每个 f_i 函数只负责计算向量 $\boldsymbol{x}^k$ 的第 i 个元素，因此式(1)等价于方程组(2)的形式：

$$x_i^{(k)}=f_i(x_1^{(k-1)},x_2^{(k-1)},\cdots,x_n^{(k-1)}),\ i=1,2,\cdots,n \tag{2}$$

分布式迭代计算的初始向量可以表示为 $\boldsymbol{x}^{(0)}=(x_1^{(0)},x_2^{(0)},\cdots,x_n^{(0)})$，利用式(2)可逐次计算迭代向量 $\boldsymbol{x}^{(k)}=(x_1^{(k)},x_2^{(k)},\cdots,x_n^{(k)})$，$k=1,2,\cdots,n$.若向量序列 $\{\boldsymbol{x}^k\}$ 无限趋近于向量 $\boldsymbol{x}^*=(x_1^*,x_2^*,\cdots,x_n^*)$，即 $\boldsymbol{x}^*=f(\boldsymbol{x}^*)$，则向量 $\boldsymbol{x}^*$ 为迭代计算的解.在某些特殊形况下求得最佳 $\boldsymbol{x}^*$ 并非容易，可能需要大量的迭代过程.但根据迭代求解的特点可知，满足迭代算法精度的近似值 $\boldsymbol{x}^k$ 可以充当算法的解.

迭代计算的构建比较简单，但许多迭代模型的计算过程都并非趋近于特定解，也即不会收敛.现实生活中只有收敛的迭代模型对用户有真实的意义，故迭代的收敛标准和条件对于迭代计算而言，尤为关键.

为了便于理解，可以将式(2)转化为

$$\boldsymbol{x}^{(k)}=\boldsymbol{M}\boldsymbol{x}^{(k-1)}+\boldsymbol{\beta},\ k=1,2,\cdots,n \tag{3}$$

其中 $\boldsymbol{M}$ 称为迭代矩阵，$\boldsymbol{\beta}$ 为一向量.当式(3)收敛时，则有：

$$\boldsymbol{x}^*=\boldsymbol{M}\boldsymbol{x}^*+\boldsymbol{\beta} \tag{4}$$

记误差向量 $\boldsymbol{e}^{(k)}=\boldsymbol{x}^{(k)}-\boldsymbol{x}^*$，那么 $\boldsymbol{x}^{(k)}\to\boldsymbol{x}^*$ 当且仅当 $\boldsymbol{e}^{(k)}-0$.由式(3)和(4)可得误差向量的递推公式：

$$\boldsymbol{e}^k=\boldsymbol{M}\boldsymbol{e}^{(k-1)},\ k=1,2,\cdots,n \tag{5}$$

对式(5)递推得到：

$$\boldsymbol{e}^k=\boldsymbol{M}\boldsymbol{e}^{(0)},\ k=1,2,\cdots,n \tag{6}$$

因此，当 $k\to\infty$ 时，$\boldsymbol{e}^k\to0$ 的充分必要条件是：$\boldsymbol{M}^k\to0$.

推论1. 迭代计算是否逼近某个值与迭代构成的矩阵 $\boldsymbol{M}$ 相关，即迭代是否收敛与迭代计算的函数密不可分.因此，迭代计算过程具有很好的健壮性，在迭代循环的迭代过程中迭代变量产生一些误差，模型的最终收敛也不受影响.现实迭代算法中的迭代矩阵 $\boldsymbol{M}$ 通常由概率组成，因此可以得到 $\boldsymbol{M}^k\to0$，在海量数据的迭代处理过程中，$\boldsymbol{e}^k\to0$.

由上述迭代计算的收敛性分析可知，基于补偿函数的乐观的容错机制具有一致性的收敛状态.乐观容错恢复的原理图如图6所示.在第二次迭代过程中node3上发生故障，通过补偿函数对健康节点node1和node2上的数据进行收集，并结合初始的

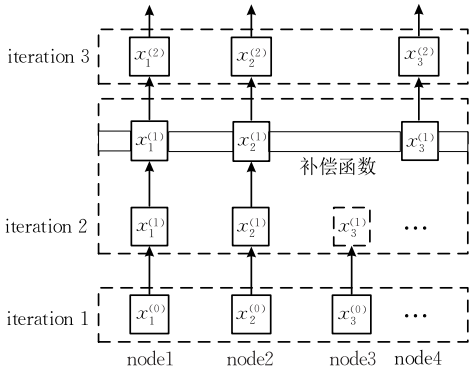


图6 乐观容错恢复原理图

完整数据对丢失的数据进行补偿恢复得到新的迭代数据，继续执行迭代计算. 迭代计算的收敛性特点确保了结果总是朝着正确的方向无限逼近，乐观容错机制使用该特点，在发生故障时，通过补偿丢失数据的近似值作为新的变量继续迭代，保证了大规模分布式迭代计算结果的正确性.

4.2　基于乐观容错机制的全量迭代算法

本节主要基于分布式全量迭代算法实现基于补偿函数的乐观容错机制，以典型的 PageRank 算法为例，PageRank 算法的介绍详见文献[2-3]. 本节进一步基于 Flink 系统设计和实现了具有乐观容错机制的 PageRank 算法.

本节以图 7 所示的网页链接为例，介绍基于补偿函数乐观容错机制的 PageRank 算法的设计与实现. 当前网络中共有 A,B,C,D,E,F 共 5 个网页，使用有向图 $G=(V,E)$ 表示该网络. 若网页 A 包含一个到 B 网页的链接，则会有一条边 $\langle A,B\rangle$. 网页中顶点集合 $V=\{A,B,C,D,E\}$，网页中边集合 $E=\{(A,B),(A,C),(A,D),(A,E),(B,A),(B,D),(B,F),(C,A),(C,F),(D,B),(D,C),(D,E),(D,F),(E,C),(F,E)\}$.

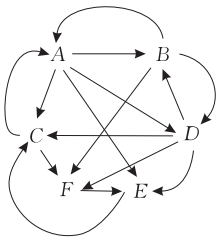


图 7　网页链接示例图

如果一个网页有 x 个链出网页，则该从网页跳转这 x 个网页的概率都为 $1/x$，该网页贡献给其跳转网页 $Rank_i/x_i$. 例如网页 A 包含 4 个跳转链接，则从 A 网页跳转到网页 B、C、D、E 的概率为均为 $1/4$，由此可以推出一个网络内网页互相跳转的概率矩阵 $\boldsymbol{M}$. $\boldsymbol{M}_{i,j}$ 表示从网页 i 跳转到网页 j 的概率.

$$\boldsymbol{M}_{i,j}=\begin{cases}1/x_i, & (j,i)\in E\\ 0, & (j,i)\notin E\end{cases} \tag{7}$$

使用概率矩阵 $\boldsymbol{M}$ 表示图 7 中网络中各网页之间的跳转概率可以表示为

$$\boldsymbol{M}=\begin{pmatrix}0 & 1/3 & 1/2 & 0 & 0 & 0\\ 1/4 & 0 & 0 & 1/4 & 0 & 0\\ 1/4 & 0 & 0 & 1/4 & 1 & 0\\ 1/4 & 1/3 & 0 & 0 & 0 & 0\\ 1/4 & 0 & 0 & 1/4 & 0 & 1\\ 0 & 1/3 & 1/2 & 1/4 & 0 & 0\end{pmatrix}$$

由于网络中可能会出现这样一些网页，它们除了本身之外没有其它的出链，或者几个网页构成的循环圈，这样会导致这个或这些网页的 Rank 值只增不减，为了规避这种情况，PageRank 算法引入了一个阻尼因子 α，假定用户会有 α 的概率通过网页之间的链接去访问网络中的其它网页，有 $(1-\alpha)$ 的概率通过直接输入浏览器地址访问. PageRank 值的计算公式为

$$PR(p_i)=\frac{1-\alpha}{N}+\alpha\sum_{p_j\in M(p_i)}\frac{PR(p_j)}{L(p_j)} \tag{8}$$

其中 $PR(p_i)$ 表示网页 i 的 Rank 值，$L(p_j)$ 表示网页 j 的链出网页数. 在实际应用中，阻尼因子 α 一般取为 0.85. 而在多次迭代过程中一般很难达到精确结果，所以一般取两轮迭代的无穷范数作为收敛精度，当相邻两轮迭代变量 Rank 之间的绝对值之差小于给定的阈值或满足最大迭代次数时，迭代终止.

记每第 i 轮迭代的 Rank 值为 $R^{(i)}$. 以图 7 网络拓扑为例，每个网页初始的 Rank 值可以表示为 $\boldsymbol{R}^{(0)}=\{1/6,1/6,1/6,1/6,1/6,1/6\}^{\mathrm{T}}$，经历 10 次迭代网页 Rank 值变化如图 8 所示.

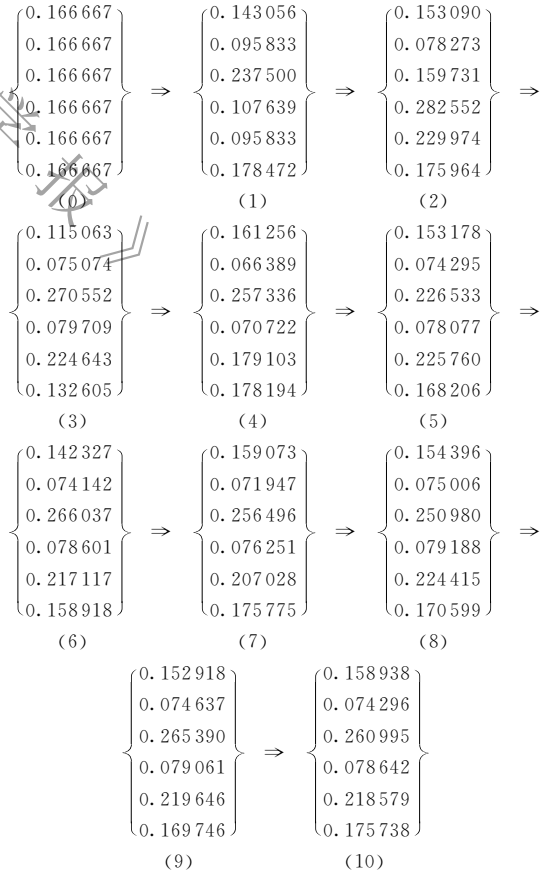


图 8　PageRank 迭代 Rank 值变化图

由图 8 可得，在第 10 次迭代后，Rank 值变为 {0.158938，0.074296，0.260995，0.078642，0.218579，

0.175738}$^{\mathrm{T}}$. 经计算可得,在第 16 次迭代时,其 Rank 值的收敛精度达到了 10^{-3},Rank 值收敛为{0.160329, 0.075958, 0.267509, 0.080386, 0.226056, 0.177387}$^{\mathrm{T}}$.

假设在某次迭代过程中,导致分区数据丢失,可以设计补偿函数统计当前丢失的数据 Rank 值个数为 n 以及丢失的总概率为 p. 为丢失的网页补偿一个相同的 Rank 值 p/n,然后和未丢失的数据一起继续执行迭代任务.

以图 7 中的网络拓扑为例,假设在第 4 迭代,集群中的计算节点 node1 发生故障,并且 node1 上保存的是顶点 B 和 C 的数据. 在发生故障后,对丢失的数据进行补偿恢复,得到新一轮迭代的输入. 迭代过程中网页的 Rank 值变化如图 9 所示.

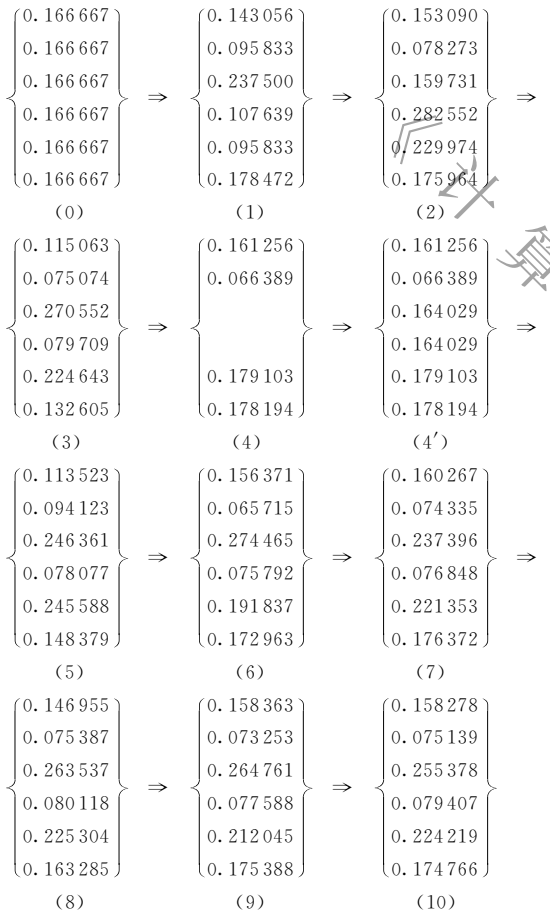


图 9 PageRank 补偿后 Rank 值变化图

由图 9 可以看出第 4 次迭代,节点 node1 出现故障后对节点 B、C 的 Rank 值补偿为 0.164029,再次经过 6 次迭代后得到的 Rank 值与图 8 的正确 Rank 值比较接近,且在补偿后迭代执行在第 14 次时,其 Rank 值收敛精度达到 10^{-3},Rank 值收敛为{0.160837, 0.076144, 0.268749, 0.080589, 0.226785, 0.177956}$^{\mathrm{T}}$.

采用本文提出的乐观补偿函数对丢失的网页恢复并继续迭代得到的网页排名与无故障迭代计算得到的结果一致. PageRank 算法对应的补偿函数具体执行过程如算法 1 所示.

算法 1. PageRank 全量迭代算法补偿函数.

输入:顶点集合 V,当前迭代变量 $R^{(k)}$

输出:对故障节点丢失数据补偿后的新迭代变量 $R_{\mathrm{new}}^{(k)}$

1. SumRank←sum($R^{(k)}$)//统计健康节点 Rank 值
2. LostNum←count(V)−count($R^{(k)}$)//计算丢失的顶点数量
3. CompensationRank←(1−SumRank)/LostNum
4. //对丢失顶点的 Rank 值进行补偿
5. FOR each v_i in V//遍历节点上的顶点
6. if v_i in $R^{(k)}$//如果未丢失
7. add $R_i^{(k)}$ to $R_{\mathrm{new}}^{(k)}$//直接加入到 $R_{\mathrm{new}}^{(k)}$
8. else//如果丢失
9. add Compensation Rankto $R_{\mathrm{new}}^{(k)}$
10. //将丢失补偿后的 Rank 加入到 $R_{\mathrm{new}}^{(k)}$

4.3 基于乐观容错机制的增量迭代算法

本节主要基于分布式增量迭代算法实现基于补偿函数的乐观迭代容错机制,使用典型的 ConnectedComponents 增量迭代算法为例展开介绍. ConnectedComponents 算法第 3 部分已经介绍;本节进一步基于 Flink 系统实现了基于乐观容错机制的 ConnectedComponents 算法.

本节以图 10 所示的连通图为例,以具体实例介绍了基于乐观容错机制的 ConnectedComponents 算法的设计与实现.

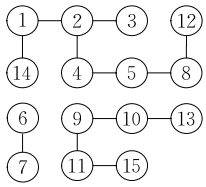


图 10 ConnectedComponents 算法示例图

假设当前共有 Id 为 1～15 的 15 个顶点组成的图. 如图 7 所示为顶点之间的连通关系. 记顶点的分类值为 CId,初始时所有顶点的 Cid=ID.

使用 ConnectedComponents 算法对图 10 中的顶点数据进行迭代,则每个顶点在迭代过程中对应的 Cid 值变化如图 11 所示. 在经历第 5 次迭代后,每个顶点的 Cid 值不再更新,迭代结束. ConnectedComponents 算法属于增量迭代,当某个顶点在本轮迭代过程中,其 CId 值没有发生变化,代表该顶点的 CId 值已经是其所在的连通子图中所有顶点中的最小 Id 值. 故在下次迭代时可以忽略该顶点. 在迭代过程中,若集群中某节点生故障而导致部分数据

丢失时，使用补偿函数将丢失的顶点补偿顶点的初始值，该节点周围的节点可能已收敛到最终的结果．故对于丢失的节点只需要经历较少的收敛次数即可再次得到最终收敛的结果．

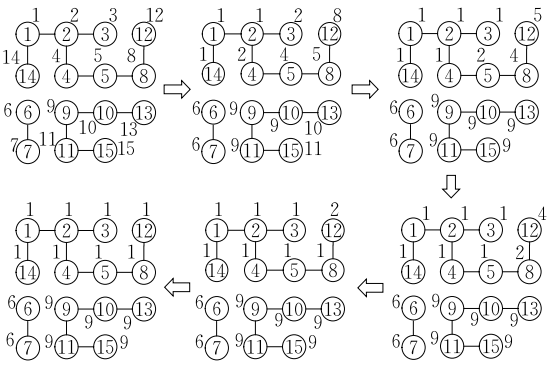


图 11　ConnectedComponents 迭代 Cid 值变化图

以图 11 为例，假如在第 3 次迭代时，某台节点 node2 发生故障，node2 上存放的顶点有 3，6，9．在发生故障后，对丢失的数据进行补偿恢复，得到新一轮迭代的输入．继续执行迭代的过程如图 12 所示．经过补偿后的顶点同在第 5 次迭代收敛，且结果一致．

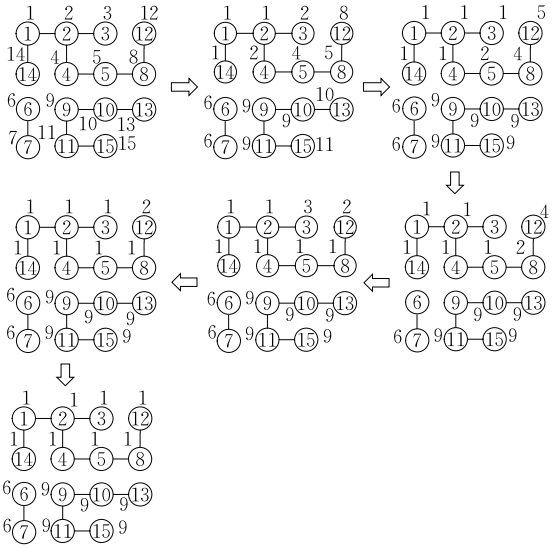


图 12　ConnectedComponents 补偿后 CId 值变化图

基于 ConnectedComponents 的补偿函数的具体执行过程如算法 2 所示．

算法 2.　ConnectedComponents 增量迭代算法补偿函数．

输入：顶点集合 V，当前迭代变量 $CId^{(k)}$

输出：对故障节点丢失数据补偿后的新迭代变量 $CId^{(k)}_{\text{new}}$

1. FOR each Id_i in V//遍历节点上的顶点

2. if v_i in $CId^{(k)}$//如果未丢失

3. add $CId_i^{(k)}$ to $CId^{(k)}_{\text{new}}$//直接加入到 $CId^{(k)}_{\text{new}}$

4. else //如果未丢失

5. add v_i to $CId^{(k)}_{\text{new}}$//直接加入到 $CId^{(k)}_{\text{new}}$

6. //将丢失的顶点的 Id 作为补偿值，并加入 $CId^{(k)}_{\text{new}}$

本文基于 Flink 系统实现了全量迭代算法的乐观补偿容错机制，现有的分布式系统没有新增乐观容错机制的原因之一在于补偿函数需要完全由用户定义和实现．本文实现的乐观补偿函数容错机制为用户提供了补偿函数接口，该接口中定义了抽象的收集数据方法并将初始数据集作为参数传入该方法，便于用户直接使用．执行过程中发生故障时，主节点 JobManager 会通过心跳信息监测到具体发生故障的 TaskManager．此时会判断用户是否编写了补偿函数，如果有，则会触发收集数据的操作，在 ExecutionGraph 中向分配了任务的健康节点发出收集数据的消息．TaskManager 收到消息后，会根据迭代任务的类型来收集数据，如果是全量迭代则会在 IterationIntermediateTask 中收集数据．如果是增量迭代，则会在其 IterationHeadTask 中收集数据．

数据收集完成后，调用用户编写的补偿函数，对丢失的数据进行恢复．将恢复得到的数据作为新的迭代输入继续执行．

5　基于头尾检查点的悲观容错机制

如前所述，现有分布式计算系统迭代容错效率低，采用的悲观检查点机制时以阻塞的方式写入外部存储，额外开销大，没有针对迭代计算的特点制定特定的容错机制．本节提出了一种基于头尾检查点的悲观容错机制，该机制以一种不受阻塞的方式编写检查点，将可变的数据集输入迭代数据流，降低了检查点成本和故障恢复开销．进一步基于 Flink 系统实现了头尾检查点机制．本节首先介绍了 Flink 系统中的迭代模型．其次介绍并分析了阻塞检查点和非阻塞检查点的代价开销．最后提出了尾部检查点和头部检查点机制并进行了代价开销分析．

5.1　Flink 系统中的迭代模型

本节以 Flink 系统迭代处理图算法为例，介绍了 Flink 系统的迭代模型．对于顶点数据集 $Vertex$，使用 v_id 表示顶点 id，$value$ 表示顶点的迭代变量值．对于边数据集 $Edge$，使用 s_id 表示源顶点，d_id 表示目标顶点．$payload$ 表示边的权重（该值是可选的）．通常对于图迭代计算问题，一般表示为顶

点为其它顶点生成消息，并在每个超级步中接收消息更新其值. 使用关系运算符，这类迭代计算表示为

$$V^{(i+1)} \leftarrow f(V^{(i)} \bigcup (V^{(i)}) \bowtie E) \tag{9}$$

其中，$V^{(i)}$ 是当前顶点的值，E 是边. 首先顶点为其他顶点产生消息，即 $V^{(i)} \bowtie E$. 然后，顶点收集消息以及当前值 $V^{(i)} \bigcup (V^{(i)}) \bowtie E$. 最后，使用步函数 f 来更新顶点的值.

图 13 显示了数据流系统中图迭代的通用编程框架. 要在数据流系统中执行图算法，输入数据常从外部存储（HDFS）加载以构建 *Edge* 数据集，而 *Vertex* 数据集由用户指定的初始值根据应用程序构建. 迭代运算符用于将新生成的顶点集 *Vertex′* 替换上一次迭代的顶点集 *Vertex*. 这里将 *Vertex* 和 *Vertex′* 称为迭代数据集. 在 join 阶段期间，*Edge* 和 *Vertex* 数据集彼此连接以生成中间数据集，即在顶点之间交换的信息. 这里连接运算符伴随用户自定义的函数以产生有效的值. 例如，指定用户定义的连接函数，使其与 Flink 中的 join 运算符相关联，而通过在 Spark 中的 join 之后应用 map 函数来实现. 在 groupBy 阶段，中间数据集由 groupBy 运算符应用，该运算符按目标顶点对数据进行分组以构造每个顶点的邻域. 在聚合阶段，用户定义的聚合函数应用于 *Vertex* 数据集和其邻居数据集的并集，以便计算正在处理的顶点的新值. 但是 union 运算符是可选的，因为在某些应用程序中，顶点的值仅依赖于它的邻居. 其中 Join 阶段对应了式(1)中的 $V^{(i)} \bowtie E$，Group 阶段对应了 $V^{(i)} \bigcup (V^{(i)}) \bowtie E$，Aggregation 阶段对应了 $f(V^{(i)} \bigcup (V^{(i)}) \bowtie E)$.

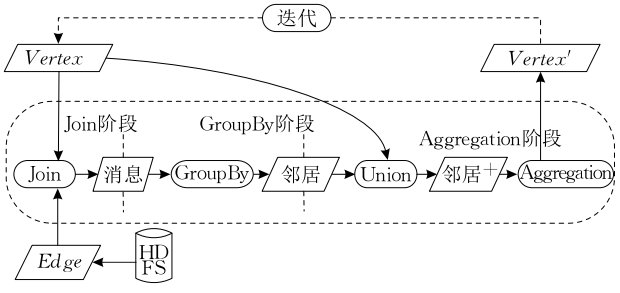


图 13 图迭代处理通用编程框架

通常，迭代输入经过每个超级步的转换后，得到的输出在迭代算法的执行过程期间作为下一个超级步的输入. 因此，在下一轮迭代过程中，*Vertex′* 数据集将替换原有的 *Vertex*，替换过程通过反向通道来完成. 其中，*Flink* 系统的迭代数据流的执行过程如图 14 所示.

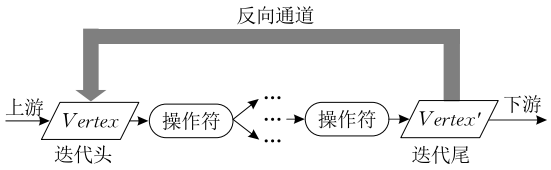


图 14 迭代数据流

5.2 阻塞检查点与非阻塞检查点

在阻塞运算符模型[26]中，每个运算符在任何下游运算符开始使用结果之前生成其完整结果. 该模型简化了检查点策略的实施，并被 Dryad[27]、Mahout 和 Pregelix 等系统广泛采用. 遵循此原则，为了编写检查点，在开始下一个超级步之前保存迭代数据集. 在迭代数据流中，检查点可以通过反向通道写入，如图 14 所示. 我们假设该集群中所有的节点工作均匀，并且工作负载在所有节点之间完美平衡. 然后，阻塞检查点的开销 O_b 如下：

$$O_b = \frac{D^i}{nv} \tag{10}$$

其中 D^i 是在超级步 i 结束之后且在超级步 $i+1$ 之前的检查点的数据大小，n 是集群中节点的数量，v 是每个节点使用的外部存储系统的写入速率.

阻塞运算符模型极大简化了容错任务，因为它可以防止下游任务消耗其上游输出的一部分数据而导致其余部分发生故障变得不可用的情况[20]. 但是此阻塞模型通常会增加执行延迟，如例 1 所示. 这种高延迟的原因是只有当迭代数据集完全可用时才会写入检查点，并且在完成检查点后，后续的超级步才可以启动计算.

例 1. 假设图迭代处理算法在由 10 个节点上组成的集群上运行. 另外，迭代数据集 *Vertex′* 的数据量即检查点的大小为 10 GB，并且每个节点上的 HDFS 的写入速率是 50 MB/s. 根据式(12)可知，在阻塞运算符模型中写入检查点的额外开销是 20.48 s. 如果一次迭代任务没有任何检查点的超级步的执行时间为 2 min，那么检查点额外开销所花费的占比为 14.6%.

Flink 系统实现的检查点容错机制，虽然不会以无阻塞的方式破坏迭代管道. 但它忽略了迭代控制，并使系统设计复杂化，需要额外的组件来协调故障恢复的检查点，特别是对于迭代图算法. 此外，节点上的磁盘故障是本地实现策略的灾难，因为故障磁盘上的数据将完全丢失，并且后向重新计算可能是耗时的.

5.3　尾部检查点与头部塞检查点

本节提出的检查点机制，通过在数据流执行过程中将检查点写入到外部存储，与迭代无关的阻塞检查点不同，该机制在数据流中，可以感知迭代. 编写检查点将可变迭代数据集保存到外部存储是一项特俗任务，该检查点的写入隐含地包含在流水线执行中. 它不仅在不破坏流水线任务的情况下继承了低延迟的优势，而且只有在当前迭代中的检查点完成后，迭代协调器才能启动下一次迭代，此外，HDFS 等外部存储为容错提供了高可用和可靠性.

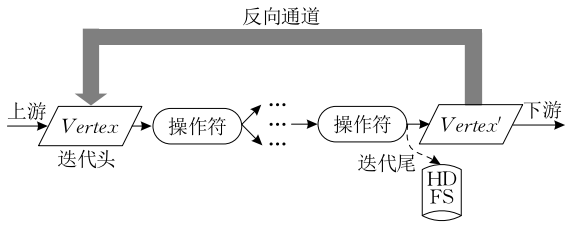
对于尾部检查点，如图 15(a)所示，检查点的写入与 $Vertex'$ 数据集的生成同步进行，在超级步的尾部写入外部存储. 即超级步尾部数据流速为 v_t，因为 $Vertex'$ 是以流水线的方式生成的. 如果产生 $Vertex'$ 的流速大于写入外部存储器的最高速率 v，即 $v \geq v_t$. 则可以在没有任何运行时间开销的情况下写入检查点. 否则，数据被累积，进程等待写入外部存储. $Vertex'$ 完全写入需要的开销为 $\frac{D^i}{nv_t}$，在此期间，已写入的数据量为 $\frac{D^i v}{nv_t}$. 因此，待写入的数据量为 $\frac{1}{n}\left[D^{(i)}-\frac{D^{(i)}(v_t-v)}{v_t}\right]$. 将剩余数据写入磁盘所需的时间是额外的开销. 给定写入速率 v，尾部检查点的开销 O^t_{ub} 为

$$O^t_{ub}=\begin{cases}\dfrac{D^{(i)}(v_t-v)}{nvv_t}, & v<v_t\\ 0, & v\geq v_t\end{cases} \tag{11}$$

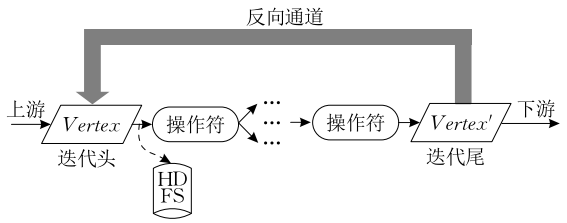
例 2.　在例 1 的基础上，如果用于生成 $Vertex'$ 数据集的流水线速率是 60 MB/s. 则根据式(11)可以计算得，尾部检查点的开销是 3.41 s.

对于头部检查点，如图 15(b)所示，检查点的写入与 $Vertex$ 数据集的完成同步进行. 由于 $Vertex$ 数据集由管道中的下游节点使用. 如果检查点的写入速率 v 大于超级步头部的流水线速率 v_h，即 $v \geq v_h$. 则可以在没有任何运行时开销的情况下完成检查点的写入. 如果 $v<v_t$，则在消耗完整个 $Vertex$ 数据集之后还存在剩余数据，剩余的数据量为 $\frac{1}{n}\left[D^i-\frac{D^{(i)}v}{v_h}\right]$，写入该数据的开销为 $\frac{D^{(i)}(v_h-v)}{nvv_h}$. 但是如果此时间小于超级步 i 的正常执行时间 t_i，即 $\frac{D^{(i)}(v_h-v)}{nvv_h}>t_i$，则仍然没有运行时开销，因为将数据写入外部存储和下游操作符的处理是并行完成的. 否则，迭代协调器需要等待写入外部存储才能完成，以便继续执行下一个超级步. 在这种情况下，时间导致运行时开销. 然而，需要考虑头部检查点的干扰 $\delta_i(\delta_i<t_i)$，这可能会延迟作业的运行时间，因为它仍然占用计算或存储资源. 头部检查点的开销 O^h_{ub} 近似为

$$O^h_{ub}=\begin{cases}\dfrac{D^{(i)}(v_h-v)}{nvv_h}-t_i+\delta_i, & \dfrac{D^{(i)}(v_h-v)}{nvv_h}>t_i-\delta_i\\ 0, & \text{其他}\end{cases} \tag{12}$$



(a) 尾部检查点



(b) 头部检查点

图 15　尾部检查点和头部检查点

例 3 显示了头部检查点将检查点写入操作和每个超级步骤的图形计算并行化，以便显着减少检查点的运行时开销.

例 3.　在例 1 的基础上，如果使用 $Vertex$ 数据集的流水线流的速率是 60 MB/s，则根据等式(12)，尾部检查点的开销是 0.

根据前面对阻塞检查点和头尾检查点的代价分析，我们可以推理出以下定理.

定理 1.　无阻塞检查点的开销不高于阻塞检查点的开销.

证明.　分别对比尾部检查点和阻塞检查点，头部检查点和阻塞检查点的开销. 尾部检查点与阻塞检查点开销对比：如果 $v<v_t$，那么 $\frac{v_t-v}{v_t}<1$，因此 $O^t_{ub}=\frac{D^{(i)}(v_t-v)}{nvv_t}<\frac{D^{(i)}}{nv}=O_b$ 并且 $O_b>0$. 因此 $O^t_{ub}<O_b$. 当 $v \ll v_t$ 时，$O^t_{ub}\approx O_b$. 头部检查点与阻塞检查点对比：如果 $\frac{D^{(i)}(v_h-v)}{nvv_h}>t_i-\delta_i$，那么，$O^h_{ub}=\frac{D^{(i)}(v_h-v)}{nvv_h}-t_i+\delta_i<\frac{D^{(i)}(v_h-v)}{nvv_h}<\frac{D^{(i)}}{nv}=O_b$. 此外 $O^h_{ub}=0$ 并且 $O_b>0$，因此，$O^h_{ub}<O_b$.

综上所述，由 $O_{ub}^{t} \lesssim O_b$ 和 $O_{ub}^{h} < O_b$ 可知，阻塞检查点模型开销高于非阻塞检查点.

定理 2. 若超级步简化为恒定速率的管道，即 $v_h = v_t$，则头部检查点开销不高于尾部检查点.

证明. 如果 $v < v_h$ 且 $\frac{D^{(i)}(v_h - v)}{nvv_h} > t_i$，那么 $O_{ub}^{h} = \frac{D^{(i)}(v_h - v)}{nvv_h} - t_i + \delta_i < \frac{D^{(i)}(v_t - v)}{nvv_t} = O_{ub}^{t}$. 否则，$O_{ub}^{h} = 0$ 并且 $O_{ub}^{t} \geq 0$. 因此 $O_{ub}^{h} \leq O_{ub}^{t}$.

定理 1 表明流水线数据流系统应采用无阻塞模型来保存迭代数据集以进行图形处理. 定理 2 表明，在超级步的头部检查用于图处理的迭代数据集可能导致低开销. 在某些情况下，头部检查点没有开销(如例 2 和例 3)，而尾部检查点会产生显着的成本.

6 实验分析

对于本文提出的两种容错机制，在 Flink 系统上进行了实现，通过修改 Flink 底层源码，提供了用户可以直接使用的补偿函数接口和可设置的头尾检查点参数. 通过在源码中增加一些 kill 计算节点的方法，模拟了计算节点出现故障. 使用优化的容错机制与 Flink 系统原有的容错机制在故障发生后任务恢复耗费的的迭代次数和时间进行了对比与分析，分别采用全量迭代 PageRank 算法和增量迭代 ConnectedComponents 算法在不同规模的数据集上进行了实验.

6.1 数据集

本文针对 PageRank 算法和 Components 算法采用了两类数据集. 分别是真实数据集和模拟数据集，真实数据集 gemsec-Facebook 是斯坦福大学 2017 年 11 月收集的有关 FaceBook 页面的数据；wiki-topcats 是斯坦福大学 2011 年 9 月收集的维基百科的超链接网络图；Hollins 数据集是霍林斯大学教育网的网页链接关系数据；as-Skitter 数据集为 Internet 拓扑，包含 http://www.caida.org/tools/measurement/skitter 网站 2005 年每天运行的网页链接关系. cit-Patents 数据集为国家经济研究局维护的美国专利数据集，涵盖了 1963 年至 1999 年的专利及引用数据. web-Google 数据集为谷歌网页数据.

真实数据集主要用于全量迭代 PageRank 算法实验分析. 模拟数据集 dataset1-3 是在实验时随机生成的具有较多连通分量的图数据集. 其主要用于 ConnectedComponents 算法，因为在执行该算法时，收敛较快，故随机生成了规模较大的连通图数据集. 详细信息如表 1 所示.

表 1 实验环境配置

配置	参数
机器节点数量	1 主节点，6 从节点
内存	32GB×6
操作系统	CentOs7
JDK 版本	1.8.0_191
开发环境	IntelliJIDEA
Flink 版本	1.4.2
Hadoop 版本	2.7.3

6.2 实验环境设置

本文采用的全量迭代算法 PageRank 和增量迭代算法 Components 基于 Flink1.4.2 实现，本文对底层源码进行了修改，新增了迭代任务的乐观恢复容错功能. 并采用 java 语言编写具有乐观恢复容错机制的 PageRank 算法和 Components 算法的案例与未优化的 Flink 进行了实验对比. 实验分析的环境设置和使用的数据集如表 1 和表 2 所示.

表 2 数据集

数据集	Nodes	Edges
gemsec-Facebook[28]	50515	819306
wiki-topcats[29]	1791489	28511807
Hollins[30]	6012	23875
as-Skitter[31]	1696415	11095298
cit-Patents[32]	3774768	16518948
web-Google[33]	875713	5105039
dataset-1	2020	40000
dataset-2	500122	5000121
dataset-3	10000000	12000000

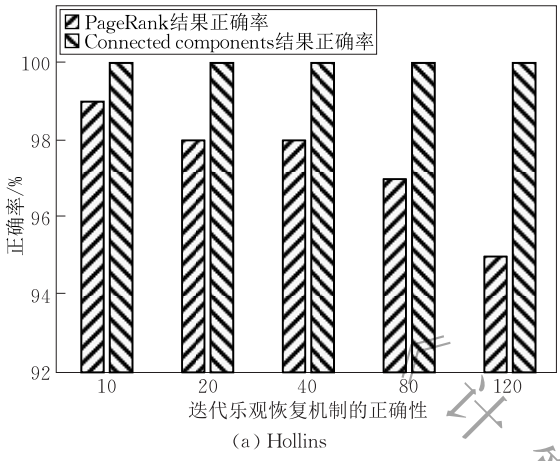
6.3 实验结果与分析

本文通过修改 Flink1.4.2 的源码，新增了补偿函数接口. 新增参数来设置在指定的迭代次数 kill 节点，模拟故障的发生. 乐观容错机制与 Flink1.4.2 原有的重启恢复容错机制在正确性、运行时间、故障发生后恢复所用的迭代次数上进行了分析. 在不同规模的数据集上展示的实验效果表明，网络中顶点之间的边数越多，对于全量迭代算法来说，收敛速度越慢. 对于全量迭代算法来说，收敛速度越快.

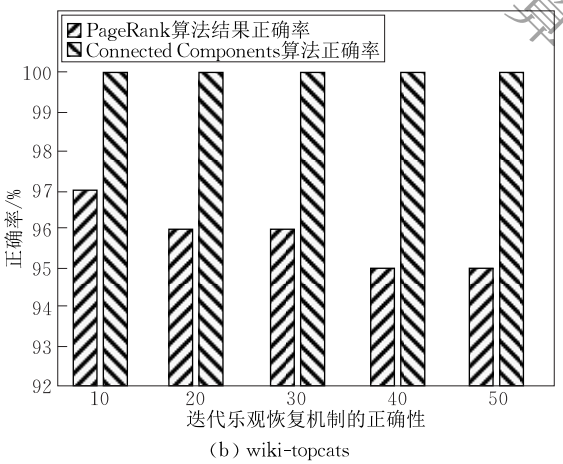
(1) 正确性评估

基于乐观容错机制全量迭代算法 PageRank 通过使用表 1 中的小数据集(Hollins)和大数据集(wiki-topcats)在分布式集群上运行. PageRank 算法迭代收敛的阈值 ξ 取 $1/(100 \times N)$ 的小数单位，N 为网页总数. 在所有实验中，模拟故障发生时丢失的节点上的数据量为 1/20. 为了充分验证算法的正确

性(和正常迭代结果的重复率)，对不同数据集的总迭代次数(Hollins：142 次，wiki-topcats：64 次)按不同的迭代间隔触发节点 s1 发生故障，并采用乐观的容错机制进行恢复. 将得到的最终结果与正常迭代的结果进行对比，由于 PageRank 算法结果本身就是近似值，故两次迭代排名误差小于 10 的网页认为排名正确. 小数据集结果的正确性和大数据集的正确性分别如图 16 所示.



(a) Hollins



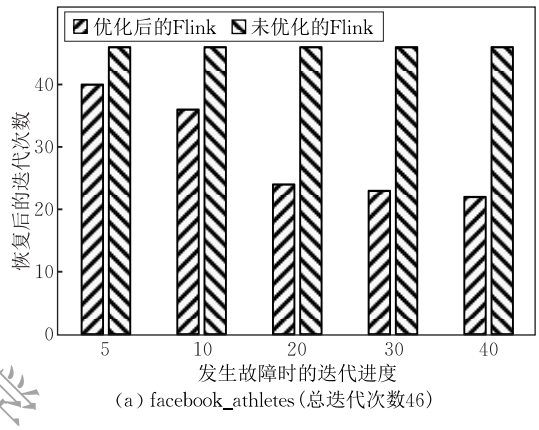
(b) wiki-topcats

图 16　迭代乐观复制机制的正确性

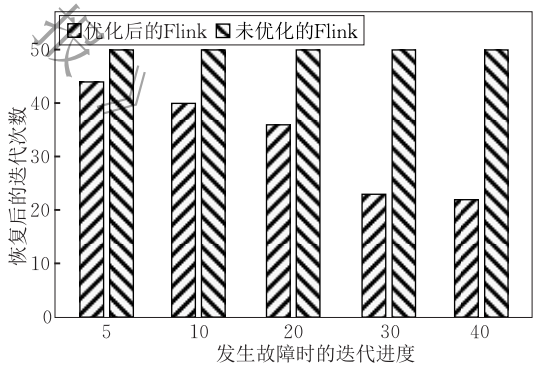
图 16(a)为小型数据集 Hollins 的实验效果，图 16(b)为大型数据集 wiki-topcats 的实验效果. 基于乐观容错机制的 ConnectedComponents 增量迭代算法的乐观恢复在发生故障时，补偿恢复得到的结果和正常迭代得到的最终结果完全一致. 由图 16 可以得出，由于 PageRank 算法得到的最终结果是近似值，故对于 PageRank 算法实验的结果和原执行结果在同一收敛度且结果相差不超过 5%，可以认为两种结果均正确. 连通分量算法则完全保证了一致性结果即正确率为 100%. 因此，补偿函数得到的最终结果是正确的. 此外，通过使用不同规模的数据集进行实验，发现对于网络顶点数量较多的顶点，收敛速度越快.

(2)恢复性能评估

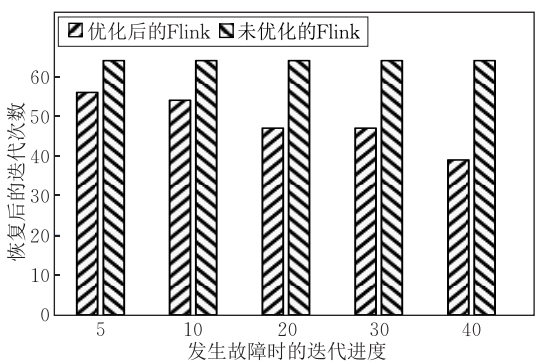
基于乐观容错机制全量迭代算法 PageRank 和增量迭代算法 ConnectedComponents 算法在模拟故障发生时，恢复任务需要继续迭代至收敛. 依然采用正确性评估中的实验条件，使用表 2 的真实数据数据集和模拟数据集进行实验，针对不同规模数据集的迭代次数可以观察出优化后的 Flink 恢复后执行的迭代次数均少于 Flink 原有的迭代次数. 基于补偿函数的乐观容错机制在 PageRank 算法和 ConnectedComponents 算法上的迭代次数提升效果分别如图 17 和图 18 所示. 实验结果表明，在迭代次数上，乐观容错机制故障恢复后比原有的 Flink 容



(a) facebook_athletes(总迭代次数46)



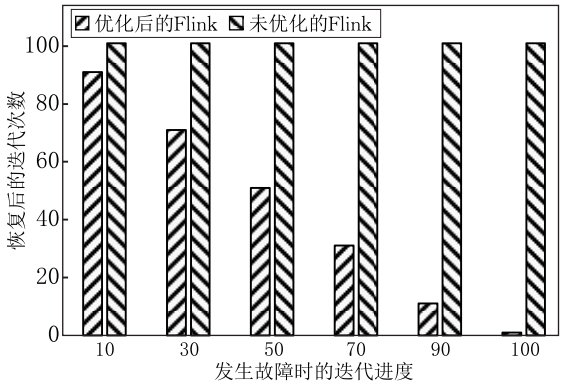
(b) facebook_artist(总迭代次数50)



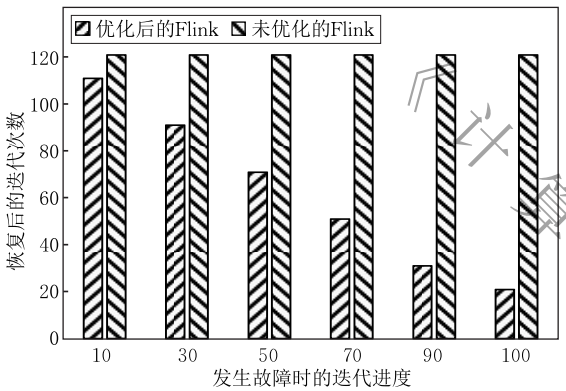
(c) wiki-topcats(总迭代次数64)

图 17　PageRank 算法恢复迭代次数比较

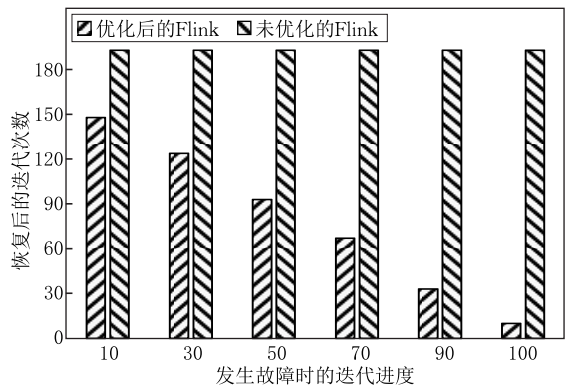
错机制在全量迭代上平均节省了 35.87%，在增量迭代上随迭代进度呈线性提升. 在不同规模的数据集上验证了乐观容错恢复机制补偿后的快速的收敛恢复速度.



(a) dataset 1(总迭代次数101)



(b) data2(总迭代次数121)

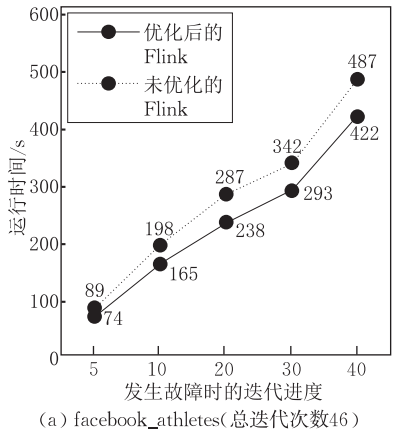


(c) dataset3总迭代次数(193)

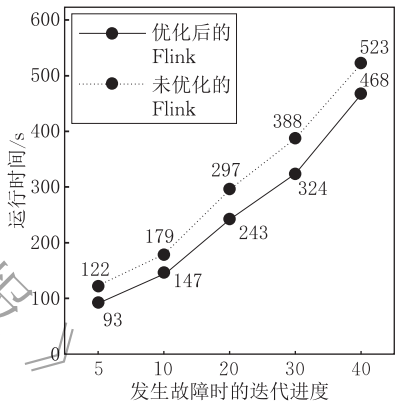
图 18 PageRank 算法恢复迭代次数比较

(3) 迭代恢复时间评估

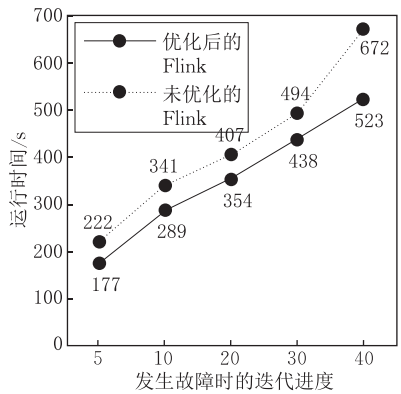
分布式迭代计算的运行时间是影响计算效率的关键. 现实世界中的分布式集群出现的故障时间是无法精确预估的. 为了保证实验的完整性，实验假定故障在总的迭代次数执行一半时发生故障. 使用 PageRank 算法和 ConnectedComponents 算法对大规模数据集进行了实验分析，如图 19 和图 20 分别为不同数据集下的实验效果. 图 21 和图 22 综合对比了不同数据集下的全量迭代和增量迭代使用乐观容错机制提升的效率. 实验效果表明，在运行时间上，基于乐观的容错机制比 Flink 原有的容错机制在全量迭代上平均提升了 16.81%，在增量迭代平均提升了 24.2%.



(a) facebook_athletes(总迭代次数46)



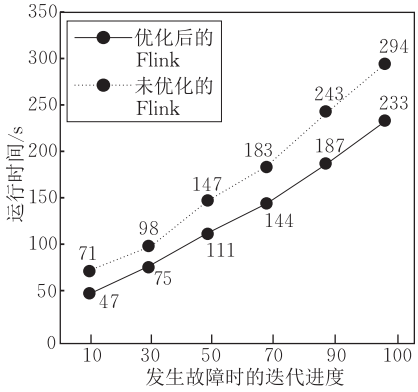
(b) facebook_artist(总迭代次数50)



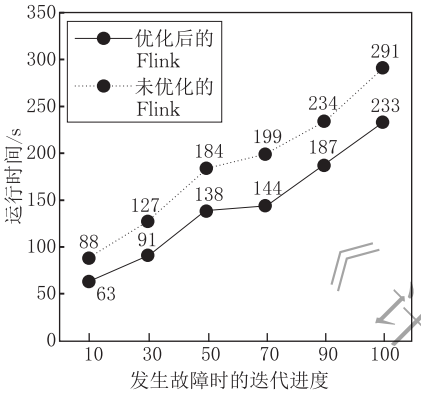
(c) wiki-topcats(总迭代次数64)

图 19 PageRank 算法运行时间比较

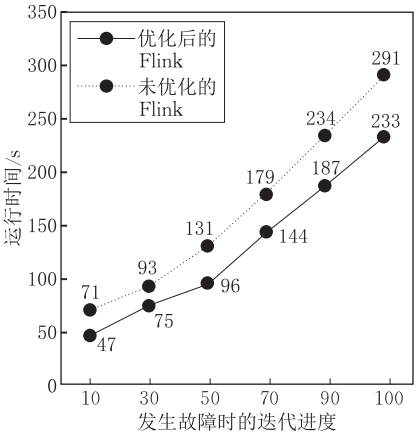
表 3 和表 4 分别列举了乐观迭代容错优机制和悲观迭代容错优化机制的具体实验条件以及提升的百分比. 基于头尾检查点的容错机制在故障发生的不同阶段时间开销不同. 此外，不同的硬件环境下，



(a) dataset1(总迭代次数101)



(b) dataset2(总迭代次数343)



(c) dataset3(总迭代次数375)

图 20　ConnectedComponent 算法运行时间比较

检查点的写入速率和效率也会有影响. 为了尽量避免该影响,实验以迭代次数$\sqrt{n}$(其中 n 为总迭代次数)为间隔使用 ConnectedComponents 算法和 PageRank 算法在数据 i-topcats 和 as-Skitter 上进行实验. 其中运行时间分别如图 23 所示. 实验效果表明,尾部检查点的平均时间开销在全量和迭代和增量迭代上均优于现有的阻塞检查点,平均节省开销分别为 13.82%和 10.87%. 面向迭代任务的尾部检查点的容错效率高于头部检查点.

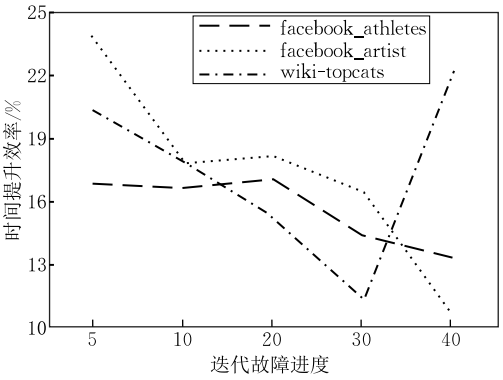


图 21　全量迭代不同数据集下提升的效率

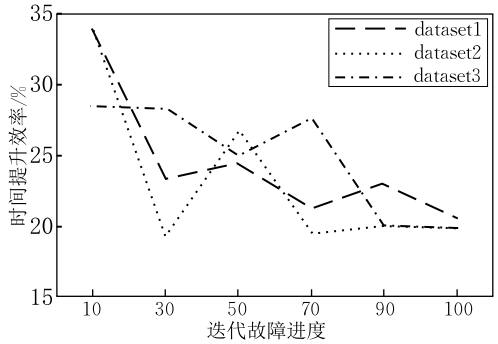
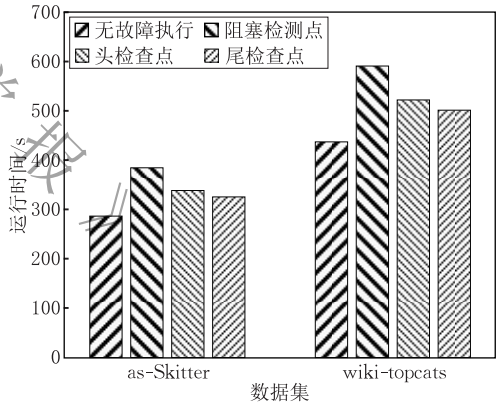
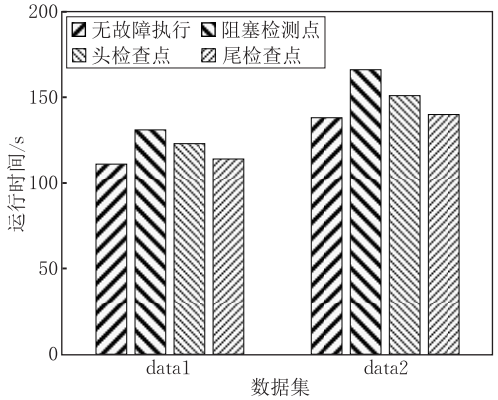


图 22　增量迭代不同数据集下提升的效率



(a) PageRank检查点时间对比



(b) ConnectedComponents检查点时间对比

图 23　不同检查点容错机制运行时间对比

表 3 乐观容错机制实验条件及性能优化对比

(a) PageRank 算法执行时间对比

	web-Google	soc-LJ
总迭代次数	64	58
EPSILON	0.000000001	0.000000001
Pages	875713	4847571
Links	5105039	68993773
模拟故障	第 32 次	第 29 次
未优化时间	365 s	1867 s
优化后时间	343 s	1764 s
提升百分比	5.75%	5.52%

(b) ConnectedComponents 算法执行时间对比

	dataset1	dataset2	soc-LJ
总迭代次数	101	121	12
Vertex	2020	500122	4847571
Edge	40000	5000121	68993773
模拟故障	第 50 次	第 60 次	第 6 次
未优化时间	111 s	164 s	301 s
优化后时间	136 s	138 s	266 s
提升百分比	22.52%	15.85%	11.63%

表 4 悲观容错机制实验条件及开销对比

(a) PageRank 算法检查点所占开销

	sc-Skitter	wiki-topcats
正常时间	292 s	434 s
头检查点	340 s	526 s
尾检查点	327 s	503 s
阻塞检查点	393 s	697 s
头检查点开销	16.4%	21.2%
尾检查点开销	11.9%	15.8%
阻塞检查点	34.6%	37.7%
头尾平均容错	14.2%	18.5%

(b) Connected Components 算法检查点所占开销

	data1	data2
正常时间	112 s	138 s
头检查点	124 s	153 s
尾检查点	115 s	140 s
阻塞检查	133 s	165 s
头检查点开销	9.6%	10.9%
尾检查点开销	2.7%	1.4%
阻塞检查点	18.8%	19.6%

7 总 结

本文提出了面向大规模迭代计算的乐观容错机制和悲观容错机制. 与现有的大数据计算平台的容错机制不同,乐观容错机制只在故障发生时进行补偿恢复,减少了不必要的容错代价和开销,在故障率较低的情况下,提拱了高效的处理性能. 悲观容错机制主要采用迭代数据流任务的特点,将检查点注入迭代数据流中,无需阻塞操作,以较低的开销保证了迭代任务的正确执行. 大量的实验表明,在处理大规模的迭代计算时,乐观恢复的容错机制随迭代规模的增大和迭代进度的推移节省的迭代次数越多,全量迭代运行时间平均提升 35.87%. 增量迭代随迭代进度呈线性提升. 在迭代中期出现故障时,全量迭代运行时间平均提升 16.81%. 增量迭代平均运行时间提升 24.2%. 头尾检查点容错机制虽然与不同的硬件环境和网络带宽有关,但其总体代价开销在不同情况下均小于阻塞检查点. 尾部检查点面向增量迭代任务平均节省的时间代价开销为 13.82%,面向全量迭代任务平均提升 10.87%. 由于受网络带宽和外部设备的影响,悲观的容错机制误差较较大,但尾部检查点和头部检查点的代价开销均小于阻塞检查点. 后期将会采取某种技术手段,调整网络带宽并结合磁盘是否空闲来减少外界条件不同引入的误差. 目前的补偿函数迭代次数优化的比例通常高于迭代时间优化的比例,可能的原因是补偿函数数据收集完成之后没有释放中间数据的内存,导致之后的迭代时间优化效率偏低. 下一步将针对现有的内存消耗型补偿算法进行优化.

参 考 文 献

[1] Zhang Y, Gao Q, Gao L, et al. iMapReduce: A distributed computing framework for iterative computation. Journal of Grid Computing, 2012, 10(1): 47-68

[2] Massucci F A, Docampo D. Measuring the academic reputation through citation networks via PageRank. Journal of Informetrics, 2019, 13(1): 185-201

[3] Haveliwala T H. Topic-sensitive PageRank: A context-sensitive ranking algorithm for web search. IEEE Transactions on Knowledge and Data Engineering, 2003, 15(4): 784-796

[4] Gonzalez Sanchez R. Measurements and analysis of online social networks. Materials Transactions JIM, 2014, 21(3): 159-168

[5] Li J R, Chen L, Wang S P, et al. A computational method using the random walk with restart algorithm for identifying novel epigenetic factors. Molecular Genetics and Genomics, 2018, 293(1): 293-301

[6] Adewole K S, Anuar N B, Kamsin A. Malicious accounts: Dark of the social networks. Journal of Network & Computer Applications, 2017, 79(1): 41-67

[7] Shu Kai, Wang Su-Hang, Tang Ji-Liang, et al. User identity linkage across online social networks: A review. ACM SIGKDD Explorations Newsletter, 2017, 18(2): 5-17

[8] Xu C, Holzemer M, Kaul M, and Markl V. Efficient fault-tolerance for iterative graph processing on distributed dataflow systems//Proceedings of the IEEE 32nd International Conference Data Engineering. Helsinki, Finland, 2016: 613-624

[9] Peng W, Li M, Chen L, et al. Predicting protein functions by using unbalanced random walk algorithm on three biological networks. IEEE/ACM Transactions on Computational Biology and Bioinformatics, 2017, 14(2): 360-369

[10] Yu Jie-Geng, Ying Lin, Yan Ma, et al. Interactive visualization of DGA data based on multiple views. Journal of Physics Conference, 2017, 787(1): 012001

[11] Wang Qian, Wang Jun-Bo. Improved collaborative filtering recommendation algorithm. Computer Science, 2010, 37(6): 226-228(in Chinese)
(王茜，王均波. 一种改进的协同过滤推荐算法. 计算机科学，2010，37(6)：226-228)

[12] Jeong Y J, Lee J, Moon J, et al. *K*-Means data clustering with memristor networks. Nano Letters, 2018, 18(7): 4447-4453

[13] Dittrich J, Quiané-Ruiz J A. Efficient big data processing in Hadoop MapReduce. Proceedings of the VLDB Endowment, 2012, 5(12): 2014-2015

[14] Zaharia M, Xin R S, Wendell P, et al. Apache spark: A unified engine for big data processing. Communications of the ACM, 2016, 59(11): 56-65

[15] Carbone P, Katsifodimos A, Ewen S, et al. Apache flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2015, 36(4): 28-38

[16] Scherbaum J, Novotny M, Vayda O. Spline: Spark lineage, not only for the banking industry//Proceedings of the IEEE International Conference on Big Data and Smart Computing (BigComp). Shanghai, China, 2018: 495-498

[17] Kiehn A, Aggarwal D. A study of mutable checkpointing and related algorithms. Science of Computer Programming, 2018, 160(1): 78-92

[18] Wen Mei, Li Hong-Liang. Research and practice of the roll-forward recovery technique in distributed and real-time systems. Computer Engineering & Science, 1999, 21(5): 28-31(in Chinese)
(文梅，李宏亮. 分布式实时系统中前向恢复技术的研究与实践. 计算机工程与科学，1999，21(5)：28-31)

[19] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. In OSDI, 2004: 137-150

[20] Zhou Jiang, Wang Wei-Ping, Meng Dan, et al. Key Technology in distributed file system towards big data analysis. Journal of Computer Research and Development, 2014, 51(2): 382-394(in Chinese)
(周江，王伟平，孟丹等. 面向大数据分析的分布式文件系统关键技术. 计算机研究与发展，2014，51(2)：382-394)

[21] Ewen S, Tzoumas K, Kaufmann M, et al. Spinning fast iterative data flows. Proceedings of the VLDB Endowment, 2012, 5(11): 1268-1279

[22] Dudoladov S, Xu C, Schelter S, et al. Optimistic recovery for iterative dataflows in action//Proceedings of the ACM SIGMOD International Conference on Management of Data. Melbourne, Victoria, Australia, 2015: 1439-1443

[23] Ekanayake J, Li H, Zhang B, et al. Twister: A runtime for iterative MapReduce//Proceedings of the ACM International Symposium on High Performance Distributed Computing. Chicago, Illinois, 2010: 810-818

[24] Bu Y, Howe B, Balazinska M, et al. HaLoop: Efficient iterative data processing on large clusters. Proceedings of the VLDB Endowment, 2010, 3(1-2): 285-296

[25] Upadhyaya P, Kwon Y C, Balazinska M. A latency and fault-tolerance optimizer for online parallel query plans//Proceedings of the ACM SIGMOD International Conference on Management of Data. 2011: 241-252

[26] Alexandrov A, Bergmann R, Ewen S, et al. The stratosphere platform for big data analytics. The International Journal on Very Large Data Bases, 2014, 23(6): 939-964

[27] Benjamin A S, Diaz M M, Laura E, et al. Tests of the DRYAD theory of the age-related deficit in memory for context: Not about context, and not about aging. Psychology & Aging, 2012, 27(2): 418-428

[28] Li Z, Nie F, Chang X, et al. Rank-constrained spectral clustering with flexible embedding. IEEE Transactions on Neural Networks and Learning Systems, 2018, 29(12): 6073-6082

[29] Yin H, Benson A R, Leskovec J, et al. Local higher-order graph clustering//Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Halifax, Canada, 2017: 555-564

[30] Diefenderfer C L, Doan R A, Salowey C. The quantitative reasoning program at Hollins University. Peer Review, 2004, 6(4): 13

[31] Leskovec J, Kleinberg J, Faloutsos C. Graphs over time: Densification laws, shrinking diameters and possible explanations//Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining. Chicago, USA, 2005: 177-187

[32] Leskovec J, Lang K J, Dasgupta A, et al. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Internet Mathematics, 2009, 6(1): 29-123

[33] Centola D. The social origins of networks and diffusion. American Journal of Sociology, 2015, 120(5): 1295-1338

GUO Wen-Peng, M. S. candidate. His major research interest is big data.

ZHAO Yu-Hai, Ph. D., professor. His major research interest is data mining.

WANG Guo-Ren, Ph. D., professor. His major research interest is database.

WEI Liu-Guo, M. S. candidate. His main research interest is big data.

Background

Distributed iterative computing is one of the mainstream technologies in the field of big data processing and analysis. The fault tolerance mechanism is a necessary guarantee for high availability of distributed systems. Although the fault tolerance mechanism of existing distributed systems performs well in high availability, it ignores the problem of fault tolerance efficiency for iterative computing. The new generation of big data computing system Apache Flink mainly uses the "distributed snapshot" checkpoint mechanism to perform fault tolerance when performing stream processing tasks. For iterative analysis of massive data, checkpoints add unnecessary delay. When executing a batch processing task, when the computing node fails and the task fails, the task is executed from the beginning to complete the fault tolerance. This fault tolerance method brings a lot of overhead.

Based on the characteristics of Flink's architecture and iterative processing, this paper proposes an optimistic iterative fault tolerance mechanism based on compensation functions and a pessimistic iterative fault tolerance mechanism based on head-to-end checkpoints, which reduces fault recovery time and fault tolerance overhead and improves iterative calculation effectiveness. The experimental results prove the efficiency of the iterative fault-tolerant optimization technique proposed in this paper.

This work was supported by the National Key Research and Development Program of China (2018YFB1004402) and the General Program of the National Natural Science Foundation of China (61772124).