·先进计算与数据处理· 文章编号:1000-3428(2017)02-0063-05 文献标志码:A 中图分类号:TP311

基于数据冗余的 HBase 合并机制研究

熊安萍,王运萍,邹 洋

(重庆邮电大学 计算机科学与技术学院,重庆 400065)

摘 要: HBase 列式数据库的所有操作均以追加数据方式写入,导致其合并机制占用资源过多,影响系统读性能。为解决该问题,提出一种基于数据冗余的合并机制,将列族下文件删除数据占比达到设定阈值的文件进行合并,以减少无用数据在系统中的占用空间。实验结果表明,与 HBase 原有仅考虑文件大小、个数和时间间隔的合并机制相比,改进的合并机制可提高 HBase 系统查询效率以及 Major 合并性能。

关键词: 列式数据库;存储;HBase 合并机制;CPU 利用率;读性能

中文引用格式: 熊安萍,王运萍,邹 洋. 基于数据冗余的 HBase 合并机制研究[J]. 计算机工程,2017,43(2):63-67.

英文引用格式: Xiong Anping, Wang Yunping, Zou Yang. Research on HBase Compaction Mechanism Based on Data Redundancy[J]. Computer Engineering, 2017, 43(2):63-67.

Research on HBase Compaction Mechanism Based on Data Redundancy

XIONG Anping, WANG Yunping, ZOU Yang

(School of Computer Science and Technology, Chongqing University of Posts and Telecommunications, Chongqing 400065, China)

【Abstract】 In HBase, the operations are written to database in the form of appending data. HBase Compaction mechanisms occupy plenty of system resources, which affects read performance. To solve this problem, a mechanism based on data redundancy is proposed. By compacting the column files whose ratio of deleted data equals the threshold, the algorithm can reduce space occupation because it reduces the number of files while cleaning useless data. Experimental result indicates, compared with the original HBase Compaction mechanism, which only considers the size and number of files and time interval, the proposed Compaction mechanism can improve HBase system query efficiency and enhance HBase Major compaction capability.

【Key words】 column database; storage; HBase Compaction mechanism; CPU utilization rate; read performance

DOI: 10.3969/j.issn.1000-3428.2017.02.011

0 概述

HBase 由于其高可靠、高可扩展性,适合存储非结构化数据等特点而得到广泛应用。HBase 存储使用 LSM(Log Structured Merge)[1]树结构实现,合并(Compaction)是该机制中的重要组成部分,其核心在于通过减少磁盘数据文件个数和删除无用数据,从而减少磁盘 I/O,达到提高系统读性能的目的。但该机制在一次合并中,如选取合并文件过多,也会增加磁盘 I/O,影响系统性能。另外,随着系统的运行,文件中无用数据越多,数据的查询效率越低。现有 HBase 中的 Compaction 机制有 2 种类型:Minor 和 Major,Minor Compaction 是对列族下部分文件的合并以及过期版本数据的整理;Major Compaction 是对列族下全部文件的合并和删除以及多版本数据的清理。因此,在 Major Compaction 之前,操作产生的数据删除越多对系统的读性能影响越大;同时在此过程中合并文件数较多,占用系统资源也会增加,对系统性能影响较大。

HBase 在进行删除操作时不会立即删除文件中的数据,而是对需要删除的数据打上一个“墓碑”标记。在此操作之后被删数据已经成为无效数据,但仍然存在于系统中,本文将此类数据视为系统内占用了多余空间的冗余数据。通过分析 HBase 中的 Compaction 机制,本文提出一种基于数据冗余的 Compaction 算法,通过标记文件中的删除数据占比,

基金项目: 重庆市教委科学技术研究项目(KJ1400414);重庆邮电大学博士启动基金(A2015-17);重庆邮电大学自然科学基金(A2011-29)。

作者简介: 熊安萍(1970—),女,教授、博士,主研方向为高性能计算、信息安全;王运萍,硕士研究生;邹 洋,讲师、硕士。

收稿日期: 2015-12-29 **修回日期:** 2016-02-18 **E-mail:** 970454492@qq.com

在适当时机清理删除数据,提升数据查询效率和 Major Compaction 性能。

1 相关工作

1.1 HBase Compaction 机制介绍

HBase 自推出以来,得到了业界的广泛应用及研究。针对 HBase 存储方面,文献[2]评估了 HBase 数据存储位置信息的随机读取和随机写入以及检索、存储数据到 HDFS 的性能;文献[3]提出一种基于备份日志的持久性、可用性方案,使得 HBase 能够在不同的缓存规模下获得更好的写性能;文献[4]通过分析主流分布式 NoSQL 在写操作方面存在的问题,提出 2 个优化批处理算法来解决服务端批处理低效问题。在 HBase Compaction 机制研究方面,文献[5]提出一种将 Major Compaction 操作从 RegionSever 转移到 CompactionSever(专注处理 Compaction 任务)上去处理的方法。文献[6]将 Compaction 形式化地定义为最小化磁盘 I/O 优化问题并证实其是一个 NP 难问题,并用实验证明基于平衡树的 Compaction 算法性能较优。文献[7]通过分析 Compaction 程序,针对其中存在的系统瓶颈提出流水线 Compaction 程序(PCP),从理论分析上证实了 PCP 可以提高 Compaction 带宽。

1.2 HBase Compaction 机制分析

HBase 是 Google BigTable 的开源实现,利用 Hadoop 中 HDFS 作为其文件存储系统[8]。

1.2.1 HBase 存储机制

HBase 基于 LSM 存储实现,在高负载环境下的更新和删除方面有很大优势。LSM 将数据的写入、修改和删除都以追加的方式保持在内存中。当内存达到指定的容量限制时,其中的内容会被批量写入磁盘中。因此,在写入、更新和删除操作方面性能较高。

HBase 每个列族对应一个 Store[9]。Store 是 HBase 存储的核心,由两部分组成:MemStore,StoreFile(底层实现是 HFile)。用户在使用 HBase 时,数据首先会放入 MemStore,当 MemStore 累计达到阈值时,就会创建一个新的 MemStore,并将存满数据的 MemStore 添加到 Flush 队列中,由单独的线程 Flush 到磁盘中,形成一个 StoreFile[10]。

1.2.2 HBase Compaction 机制

HBase 为了防止小文件(被 Flush 到磁盘的 MemStore)过多,保证查询数据的效率,因此,在必要时会将小的 StoreFile 合并成相对较大的 StoreFile,这个过程称作 Compaction[11]。

HBase 在 StoreFile 文件数量增长达到阈值时,会触发 Compact 操作,将多个 StoreFile 合并成一个 StoreFile,合并过程中会进行版本合并和数据删除。因此,HBase 中所有的更新和删除操作都是在后续的 Compaction 过程中进行[12]。但是列族下的 StoreFile 文件是只读的,不可修改,在 Compaction 过程中,首先创建一个新文件,然后将待合并文件中的数据按照 Rowkey 的字典序追加写入到新文件中。图 1 所示为 HBase Compaction 机制的具体流程。

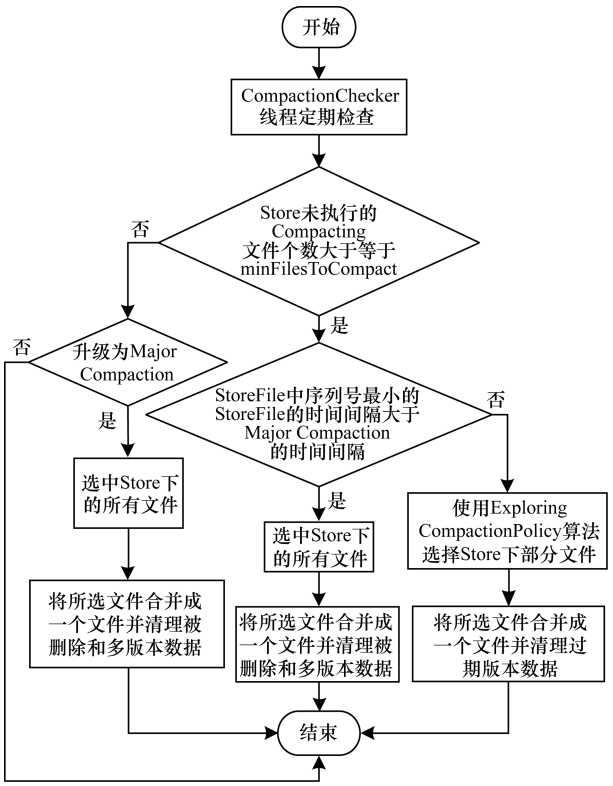

图 1 HBase Compaction 机制流程

从图 1 可知,HBase Compaction 机制根据合并时选择文件个数分为 Minor Compaction 与 Major Compaction 2 种策略。在 Store 下未合并的文件个数大于等于最小合并文件数的情况下,若 StoreFile 中序列号最小的 StoreFile 的时间间隔大于 Major Compaction 的时间间隔或系统设定为 Major Compaction,则采用 Major Compaction 策略;否则采用 Minor Compaction 策略[13]。

Minor Compaction[14]是对 Store 下部分文件进行合并,利用 ExploringCompactionPolicy 算法选择文件,该算法从头到尾遍历 StoreFile 列表的同时记录下当前最优,然后从中选择一个全局最优列表。Major Compaction[15]是将列族下的所有文件合并成一个文件,并清理被删除和多版本的数据。

2 基于数据冗余的 Compaction 算法

通过分析 HBase Compaction 机制,本节提出基于数据冗余的 Compaction 算法,首先对 Flush 算法

进行改进,将 HFile 文件分为 2 类,目的是把“墓碑”标记数据和被删除数据分开存储;然后在满足触发条件的情况下,选择删除数据占比大的文件进行合并。

2.1　问题分析

通过 1.2.2 节分析可知,在 HBase Compaction 机制中,Minor Compaction 策略主要目的是减少磁盘上的小文件个数,同时满足 HDFS 在处理大文件方面的优势条件。Minor 选取的文件个数少,且是列族下的小文件,因此,此过程产生的磁盘 I/O 比较少,对系统的负载和资源消耗影响不大,耗时短。Major Compaction 策略涉及的文件数多,文件比较大,会产生较多的磁盘 I/O,占用系统资源较多,此过程对系统的负载和性能方面造成较大的影响。系统默认每隔 24 h 进行一次 Major Compaction,但是,在实际生产环境中,一般会将这个触发开关关掉,通过手工进行触发。

HBase Compaction 机制合并文件时仅考虑了文件的大小、个数以及合并的时间间隔,因此,在进行 Major Compaction 之前,存在的删除数据越多,占用系统空间越多,对系统使用的影响就越大。

2.2　相关定义

表 1 是基于数据冗余的 Compaction 算法中涉及到的相关名词与符号的定义。

表 1　相关名词及符号定义

相关名词及符号	含义
Delete MemStore	存储删除操作写入数据的 MemStore
Add MemStore	存储除删除操作之外的操作产生数据的 MemStore
Tombstone HFile	Delete MemStore 刷入磁盘生成的文件
Add HFile	Add MemStore 刷入磁盘之后形成的文件
删除数据占比 R	被删除数据在 HFile 文件中所占的比例
高峰期	在 HBase 配置文件中通过设置高峰期开始时间和结束时间(取值范围为[0,23])来确定高峰期时段
非高峰期	一天中高峰期时段之外的时段属于非高峰期时段
α	高峰期删除数据占比阈值
β	非高峰期删除数据占比阈值
Φ	待合并 Add HFile 文件集合
H_{min}	在合并过程中,选取最小合并文件个数 minFilesToCompact,可在 HBase 配置文件中进行设置
H_{max}	在合并过程中,选取最大合并文件个数 maxFilesToCompact,可在 HBase 配置文件中进行设置

2.3　Flush 算法改进

对于删除操作,HBase 不会立即将文件中的数据删除,而是对需要删除的数据打上一个“墓碑”标记,这个“墓碑”标记记录是以追加的方式写入数据库中,这样会使得“墓碑”标记记录和被删除数据的记录存在于不同的 HFile 文件中。为了准确计算 HFile 文件中删除数据所占的比例,本节将“墓碑”标记数据和其余操作产生的数据分开存储在不同的 HFile 文件中。分开存储的实现如下:在每个 Store 下设置 2 个 MemStore:一个是 Delete MemStore;另一个是 Add MemStore。在客户端操作时,根据操作类型的不同,将数据写入不同的 MemStore 中。当 MemStore 大小达到指定的大小之后,将其中的数据刷到磁盘中,在此记录 Delete MemStore 刷入生成的文件为 Tombstone HFile,Add MemStore 刷入之后形成的文件为 Add HFile。Store 结构如图 2 所示。

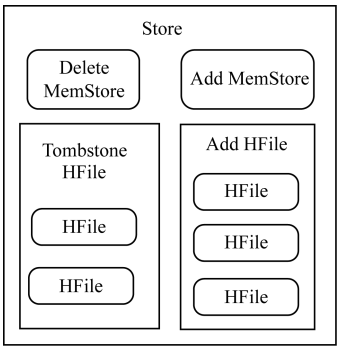

图 2　改进 Flush 算法后的 Store 结构

在每次 Delete MemStore 刷入磁盘之后,计算每个 Add HFile 文件的删除数据占比,具体计算过程如下:

遍历之前未遍历过的 Tombstone HFile,对于其中的 RowKey,找到被删除数据对应的 Add HFile,将被删除数据记录的条数加 1;当遍历完 Tombstone HFile 文件时,可以得出每个 Add HFile 文件的删除数据占比,计算公式如式(1)所示。

$$R = \frac{delCount}{totalCount} \times 100\% \tag{1}$$

其中,$delCount$ 为 Add HFile 中被删除数据记录的条数;$totalCount$ 为 Add HFile 中记录的总条数。

另外,在 HBase 中有一个 CompactionChecker 类,用来定时检查和判断系统是否需要进行合并。在本节提出的算法中需要针对此类添加 HFile 文件删除数据占比检查。在完成上述改进后,HBase 将对 Add HFile 和 Tombstone HFile 这 2 类文件分别做 Minor Compaction。

2.4　基于数据冗余的 Compaction 算法描述

本文基于数据冗余的 Compaction 算法核心在于选择删除数据占比较高的文件进行合并。具体的算法步骤如下:

1)初始化。设置删除数据占比阈值(分别对高

峰期和非高峰期设置不同的删除数据占比阈值 α,β)。

2)通过 CompactionChecker 线程对 Add HFile 删除数据占比做定期检查。

(1)如果该时段处于高峰期,则将删除数据占比大于 α 的文件添加到 Φ 中。

(2)如果该时段处于非高峰期,则将删除数据占比大于 β 的文件添加到 Φ 中。

3)检查 Φ 的元素个数 n:

(1)如果 $n < H_{\min}$,那么本次定期检查处理结束。

(2)如果 $H_{\min} \leqslant n \leqslant H_{\max}$,则选取 Φ 中的所有文件,转步骤 4)。

(3)如果 $n > H_{\max}$,则将待合并文件按照删除数据占比大小进行降序排序,选取前 $H_{\max}$ 个待合并文件,转步骤 4)。

4)将步骤 3)选取的待合并文件和所有的 Tombstone HFile 进行合并,其中所有 Add HFile 合并成一个文件 Add HFile,所有 Tombstone HFile 合并成一个 Tombstone HFile,合并过程中排除在 Add HFile 和 Tombstone HFile 的 RowKey 中所对应的记录。

5)算法结束。

由于原有 Major Compaction 过程带来的磁盘 I/O多,对系统的性能影响大、耗时长,因此在实际生产环境中关闭系统中自动触发的开关,通过手动触发。本文将删除数据提前进行合并,提高系统在 Major Compaction 之前的数据查询效率,尤其在删除操作比较频繁的情况下,同时,提高 Major Compaction 的合并性能。

3 实验结果及分析

本文将移动互联用户通话记录数据作为测试数据,将本文提出的基于数据冗余的 Compaction 机制与 HBase 原有 Compaction 机制(仅考虑文件大小、个数以及时间间隔,其中包括 Minor Compaction 和 Major Compaction)进行实验对比。由于 Compaction 是 HBase 存储的核心机制,对系统的读性能和 CPU 利用率起到重要的影响作用,因此实验通过比较两者读效率和 CPU 利用率来验证本文机制的有效性。

3.1 实验环境

本文的实验环境搭建在 Hadoop2.4.0 分布式集群上。Zookeeper, RegionSever 集群服务器配置如表 2 所示,HBase 版本为 0.98.3。实验数据是存储在 HDFS 上的移动互联用户通话记录数据,数据有 10 亿条左右。

表 2 集群服务器配置

集群各组件	服务器配置
Zookeeper	CPU:Intel 2.40 GHz 24 核,内存:24 GB,硬盘:300 GB
Master	CPU:Intel 2.40 GHz 24 核,内存:24 GB,硬盘:300 GB
RegionSever	CPU:Intel 2.20 GHz 24 核,内存:64 GB,硬盘:30 TB

3.2 结果分析

3.2.1 改进前后系统读性能和 CPU 利用率对比

实验首先从 HDFS 上将 3 亿条数据拉取存储到 HBase 上,之后不断添加剩余的数据到 HBase 中并不断删除 HBase 表中的数据。在此过程中每隔 20 min 查询表中的 1 000 条数据,同时监控 RegionSever 的 CPU 利用率。从上述实验设计中得出读性能、系统 CPU 利用率在 HBase 合并机制改进前后的对比,如图 3、图 4 所示。

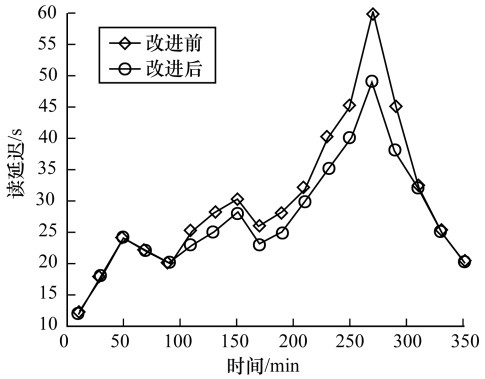

图 3 改进前后系统读性能对比

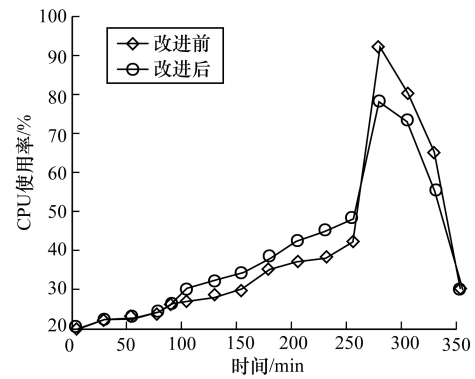

图 4 改进前后 RegionSever CPU 利用率对比

从图 3 可以看出,改进后系统读性能比改进前降低了约 2 s 左右。随着时间的增长,HBase 中数据增多,在 80 min 左右时,表中文件删除数据占比达到设置的阈值,触发基于数据冗余的 Compaction 操作,清理文件中的删除数据,这样表中无用数据减少使得系统的读性能提升。在 280 min 左右时,触发 Major Compaction 操作,所有文件合并成一个文件,系统查询效率降低。从图 4 中看出,改进后 RegionSever CPU 利用率的变化曲线比改进前变化

平缓,系统性能波动比改进前小。但是在 280 min 之前,改进后 RegionSever CPU 利用率高于改进前,在 280 min 左右,改进后的 RegionSever CPU 利用率比改进前低,因此,改进后的算法提高了 HBase Major Compaction 的合并性能。

通过上述实验可知,在 HBase 中引入基于数据冗余的 Compaction 算法,系统读性能有一定提升。同时,该算法将删除数据进行清理,文件中无用数据的减少为 Major Compaction 的性能提升起到了一定的作用。

3.2.2　删除数据占比阈值对 CPU 利用率的影响

为了考虑删除数据占比阈值对 RegionSever CPU 利用率的影响,通过设计实验进行分析。实验将 HDFS 上 5 亿条左右的移动互联用户通话记录数据全部存入 HBase 中,然后编写程序,在 5 h 时间内每分钟删除 10 000 条数据,此过程分别在删除数据占比阈值为 10% ,20% ,30% ,40% 的情况下进行。在上述实验中,实时监控 RegionSever 的 CPU 利用率,得出不同阈值情况下的 CPU 利用率,如图 5 所示。

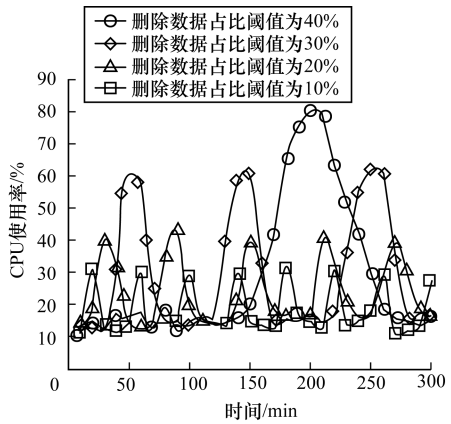

图 5　不同删除数据占比阈值情况下的 CPU 利用率

由图 5 的实验结果可知,删除数据占比阈值设置得越小,CPU 利用率变化越频繁,但 CPU 利用率越低。这是由于在实验过程中,删除数据占比阈值设置得越小,文件删除数据占比能越快达到设置的阈值,从而使得越频繁地触发基于数据冗余的 Compaction 机制,因此 CPU 利用率变化越频繁。但基于冗余数据的 Compaction 机制有利于删除数据对 I/O 带宽的充分利用,一定程度上降低了 CPU 负载。

4　结束语

HBase 依靠 Compaction 机制来合并文件并清理无用数据以提高系统读性能。本文在分析 HBase Compaction 机制的基础上,提出一种基于数据冗余的 Compaction 算法。该算法通过选取列族下删除数据占比较大的文件,并对其进行合并以减少无用数据在系统中的占用空间。通过搭建实验平台来进行对比,结果表明,在 HBase 引入该算法能够提高系统查询效率和 Major Compaction 性能。下一步考虑将机器负载因素引入 Compaction 机制中,进一步提升系统性能。

参考文献

[1] O'Neil P, Cheng E, Gawlick D, et al. The Log-structured Merge-tree(LSM-tree)[J]. Acta Informatica, 1996,33(4):351-385.

[2] Vora M N. Hadoop-HBase for Large-scale Data[C]// Proceedings of 2011 International Conference on Computer Science and Network Technology. Washington D. C.,USA: IEEE Press,2011:601-605.

[3] 唐长城,杨　峰,代　栋,等. 一种基于 HBase 的数据持久性和可用性研究[J]. 计算机系统应用,2013(10):175-180.

[4] 周　跃,臧斌宇. 分布式 NoSQL 系统写操作性能优化设计与实现[J]. 计算机应用与软件,2014,31(11):25-28.

[5] Ahmad M Y, Kemme B. Compaction Management in Distributed Key-value Datastores[J]. Proceedings of the VLDB Endowment,2015,8(8):850-861.

[6] Ghosh M, Gupta I, Gupta S, et al. Fast Compaction Algorithms for NoSQL Databases[C]//Proceedings of the 35th International Conference on Distributed Computing Systems. Washington D. C.,USA:IEEE Computer Society, 2015:452-461.

[7] Zhang Zigang, Yue Yinliang, He Bingsheng, et al. Pipelined Compaction for the LSM-Tree[C]// Proceedings of the 28th International Parallel and Distributed Processing Symposium. Washington D. C., USA:IEEE Press,2014:777-786.

[8] Harter T,Borthakur D,Dong Siying,et al. Analysis of HDFS Under HBase: A Facebook Messages Case Study[C]// Proceedings of the 12th USENIX Conference on File and Storage Technologies. Berkeley,USA:USENIX Association, 2014:199-212.

[9] 张　智,龚　宇. 分布式存储系统 HBase 关键技术研究[J]. 现代计算机,2014(11):33-37.

[10] 冯晓普. HBase 存储的研究与应用[D]. 北京:北京邮电大学,2014.

[11] 竹叶青. hbase 权威指南: store file 合并(compaction)[EB/OL]. [2015-11-08]. http://blog. csdn. net/azhao _ dn/article/details/8867036.

[12] Dumon B. Visualizing Hbase Flushes and Compaction [EB/OL]. [2015-11-08]. http://outerthought. org/blog/465-ot. html.

[13] Bhupathiraju V, Ravuri R P. The Dawn of Big Data-Hbase[C]//Proceedings of CSIBIG'14. Washington D. C., USA:IEEE Press,2014:1-4.

[14] Bao Xianqiang, Liu Ling, Xiao Nong, et al. HConfig: Resource Adaptive Fast Bulk Loading in HBase[C]// Proceedings of 2014 International Conference on Collaborative Computing: Networking, Applications and Worksharing. Washington D. C., USA: IEEE Press, 2014:215-224.

[15] Saloustros G, Magoutis K. Rethinking HBase: Design and Implementation of an Elastic Key-value Store over Log-structured Local Volumes[C]//Proceedings of the 14th International Symposium on Parallel and Distributed Computing. Washington D. C.,USA:IEEE Press,2015:225-234.

编辑　陆燕菲