使用内存缓存的迭代应用编程框架①

连文波 [1,2], 汪美玲 [1,2], 陶秋铭 [2], 赵 琛 [2]

[1](中国科学院软件研究所 基础软件国家工程研究中心, 北京 100190)
[2](中国科学院大学, 北京 100190)

摘 要: 迭代式计算是一类重要的大数据分析应用. 在分布式计算框架 MapReduce 上实现迭代计算时, 计算会被分解成多个作业并按作业依存关系顺序运行, 这使得程序与分布式文件系统(DFS)有多次交互而影响程序执行时间. 对这些交互相关数据的缓存会降低与 DFS 的交互时间, 进而提升程序总体的性能. 考虑到集群中的大量内存在多数情况下会处于空闲状态, 提出了一种使用内存缓存的迭代式应用编程框架 MemLoop. 该系统从作业提交 API、调度算法、缓存管理模块实现缓存管理以充分利用内存缓存迭代间可驻留数据与迭代内依存数据. 我们将此框架与已有相关框架进行了比较, 实验结果表明该框架能够提升迭代程序的性能.

关键词: 作业依存; 内存缓存; 迭代程序; 迭代间可驻留数据; 迭代内依存数据

MemLoop: A Programming Framework Using In-Memory Cache for Iterative Application

LIAN Wen-Bo[1,2], WANG Mei-Ling[1,2], TAO Qiu-Ming[2], ZHAO Chen[2]

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Science, Beijing 100190, China)

Abstract: The iterative computation is an important big data analysis application. While implementing iterative computation on the distributed computation framework MapReduce, the iterative program will be divided into more than one jobs which run in the order defined by the dependencies between jobs, which lead to many interactions between the program and distributed file system(DFS) that will affect the program's execution time. Caching these interaction-related data will reduce the time of interactions between the program and DFS and hence improve the overall performance of application. Considering that large amount of memory in cluster nodes is unused at most time, this paper proposes a programming framework called MemLoop using memory cache for iterative application. This system sufficiently uses the free memory in the cluster's nodes to cache data by implementing the memory caching management from three models: job submit API, task scheduling algorithm, cache management. The cached data is classified into two categories: inter-iteration resident data and intra-iteration dependent data. We compare this framework with previous related framework. The result shows that MemLoop can improve the performance of iterative program.

Key words: job dependency; in-memory cache; iterative program; inter-iteration resident data; intra-iteration dependent data

1 引言

随着互联网的发展, 在分布式计算环境中进行大规模数据分析变得更加普遍. 为了提高编程人员进行大数据分析的可扩展性、容错性、可编程性, Google 提出 MapReduce[1]分布式编程框架, 目前已经得到了广泛的应用. 在 MapReduce 中, 一个作业的执行过程分为 map 和 reduce 两个阶段: 在 map 阶段, 对每个输入记录使用 map 函数生成作业的中间结果; 在 reduce 阶段, 对作业中间结果使用 reduce 函数生成作业的最终结果.

当基于 MapReduce 框架进行大数据分析时, 系统作业均需要与分布式文件系统交互以处理数据. 大数

① 基金项目:国家自然科学基金(61100067)
收稿时间:2014-07-04;收到修改稿时间:2014-08-11

据分析的应用有许多不同种类如迭代式应用、数据查询(Hive)等. 迭代式应用是其中重要的一类, 常见的有 PageRank 算法、聚类算法、递归关系查询与反向查询等. 基于 MapReduce 框架的实现大数据分析程序时, 每个程序都由多个在数据上存在依存关系的作业组成. 以数据查询为例, 如图 1, 当查询语句提交时, 编译器会将一个查询转化为一个作业 DAG(有向无环图 Directed Acyclic Graph), DAG 里以边表示作业间关于数据的依存关系(即前一个作业的输出是后一个作业的输入). 此时, 系统与分布式文件系统之间将会发生频繁的交互, 对程序的性能有较大影响. 针对迭代应用中出现这类问题, HaLoop[2]对 Hadoop 进行了扩展, 通过将在迭代间多次使用的数据(本文称作迭代间可驻留数据)缓存至本地磁盘减少系统与文件系统之间的交互时间, 进而减少程序执行时间. 本文扩展了 HaLoop 中对数据进行缓存的思路, 研究编程框架的改进以进一步提高程序的性能.

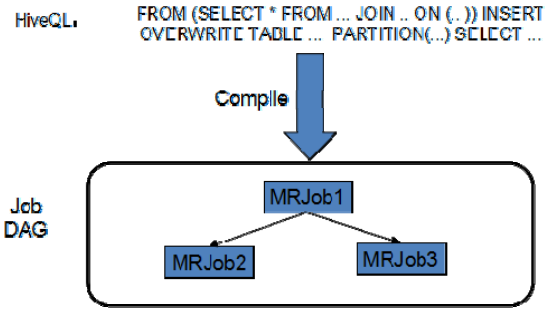


图 1 Hive 到 Hadoop 作业的转换

对 HaLoop 而言, 存在两点是其可以考虑优化而未进行的. 一是使用空闲内存作为一个缓存介质. 如今的集群有大量内存, 这些内存在多数情况下处于空闲状态, 这就为使用内存作为缓存提供可能. 二是迭代体内作业间存在关于数据 IO 的依存关系, 即一个作业生成的数据会用于下一个作业的运算, 这类数据这里称作迭代内依存数据, 对迭代内依存数据进行缓存时, 会加速程序运行. 基于这两点考虑, 本文了提出一种基于 HaLoop 的编程框架 MemLoop. 在该框架中, 我们对迭代间可驻留数据与迭代内依存数据在作业提交 API 上进行描述, 在运行时调整调度算法, 并充分利用本地内存与磁盘缓存迭代间可驻留数据与与迭代内依存数据. 我们在实验中将所提出的 MemLoop 框架与 HaLoop、Hadoop 在性能上以执行时间为标准进行了对比, 实验结果表明 MemLoop 在性能上有一定的提升.

2 MemLoop编程框架

我们在本部分提出 MemLoop 编程框架: 首先通过一个 PageRank 算法的实例介绍迭代应用的基本特征, 之后从作业提交 API、调度算法、缓存框架三个方面描述 MemLoop 对 HaLoop 的总体扩展以及扩展的具体实现.

2.1 迭代式计算用例

迭代式计算经常出现于各种应用当中, 本文以 PageRank 为例对迭代式应用进行说明. PageRank 是一个用于计算各个网站权重的算法,其主要思想: 一个网站当其指向的网站权重比较大或者被许多权重比较大的网站指向时, 其自身权重就比较大. 例如在图 2(a)中, PageRank 的输入是 T_0 表与 L 表, 在 T_0 表中所有网站的初始权重值均为 1.0, L 表保存了由点对(src,dst)构成的集合, 每个点对表示两个网站的链接关系, 即从 src 网站有指向 dst 的链接.

Url	Rank
www. a. com	1. 0
……	……
www. x. com	1. 0

T_0

url-source	url-dest
www.a.com	www.b.com
www.a.com	www.x.com
……	……
www.x.com	www.b.com
www.x.com	www.a.com

L:link table

url	rank
www.a.com	2.0
www.b.com	1.9
……	……
www.x.com	3.3

R

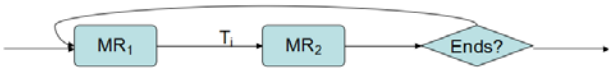
(a) PageRank 输入与最终结果

$$MR_1 \quad T_1 = R_i \triangleright\triangleleft_{url=url\_source} L$$

$$T2 = \gamma_{url,rank,\frac{rank}{COUNT(url\_dest)}\to new\_rank}(T_1)$$

$$T_3 = T_2 \triangleright\triangleleft_{url=url\_source} L$$

$$MR_2 \quad R_{i+1} = \gamma_{url,dest\to url,SUM(new\_rank)->rank}(T_3)$$



(b) PageRank 在 MapReduce 上的实现与流程图

图 2 PageRank 算法

当在MapReduce框架上实现PageRank算法时, 迭代体往往包含多个作业, 例如图 2(b)说明 PageRank 算

法的实现由两个 MapReduce 作业构成, 一个用于计算每个源向外连接的所有的边的权值, 另一个用于在所有的结点上计算指向该点的所有权值之和. 迭代的终止条件断定有两种方法, 一个是通过指定迭代的次数, 另一个是设定两个迭代体输出之间的距离阈值. 图 2(b)中迭代的结果是图 2(a)中 R 表示的形式. 在图 2(b)中可以看到, 某些数据属于迭代间可驻留数据, 如 L 表; 某些数据属于迭代内依存数据, 如图 2(b)中的 T_i. 对迭代间可驻留数据与迭代内依存数据进行缓存能够提升程序的性能.

2.2 MemLoop 主要结构

如图 3 所示, MemLoop 是基于 HaLoop 构造的并针对内存缓存与迭代内依存数据的扩展在设计上进行了改动. 在底层的实现上使用内存作为缓存介质. 在作业提交 API 里除了对于迭代的收敛条件与迭代间可驻留数据的指定之外, 还加入了对于迭代内依存数据的指定. 在作业的调度上, 在原有的 Hadoop 的调度方法之上, 针对迭代间可驻留数据与迭代内依存数据的所在结点信息相应地对 MapReduce 作业调度进行扩展. 在每个结点上, 使用一个内存管理模块将每个读写文件的需求映射到内存或者到磁盘上. MemLoop 需要针对内存缓存可能出现不足的情况处理结点上空闲的内存与待缓存的数据.

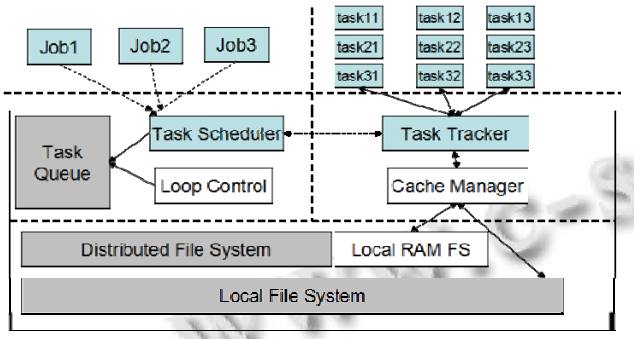


图 3　MemLoop 的主要框架

为构建针对迭代内依存与迭代间可驻留数据使用内存缓存的框架, 下面介绍 MemLoop 的三个重要部分: 作业提交 API、调度与缓存管理.

2.3 作业提交 API

MemLoop 要求程序员在写程序时, 需要对迭代内依存数据有一些相应的指定, 以便在运行时系统上能够针对 API 的这些指定使用调度与缓存模块. 作业提交的 API 主要是基于 Hadoop 的作业提交方法, 这样能够向后保持兼容性. 针对迭代程序使用内存缓存, 扩展了一些 API, 主要包括收敛的条件、迭代体内部作业间的依存关系、迭代间的可驻留数据的描述. 在迭代内作业依存关系上, 使用 addDep(src_job,dst_job)这类接口来描述一个作业与后面作业之间的依存关系. 针对迭代间的可驻留数据用 addInvariant 来描述. 依据 API 决定的作业之间的依存关系, 底层的运行时系统在向分布式文件系统写数据时会决定是否缓存一个作业的输出到本地的磁盘或者内存当中.

上述的 API 在 Hadoop 当中实现时, 主要是在 Hadoop 的常用配置上加入一些配置. 通过在作业提交 API 上关于迭代的一些配置, 底层实现能够更有针对性地进行调度与缓存管理等相关工作.

2.4 调度

传统 Task 调度的主要思路是尽可能地保持将执行的任务与其处理的数据本地性(locality). 针对迭代程序, 因对数据的缓存的影响, 在程序里为充分利用缓存的数据, 本地性的定义就扩大到关于被缓存数据的本地性问题. HaLoop 关于被缓存数据的本地性是针对迭代间可驻留数据. 在调度时, 通常一些不变的数据在执行之后, 记录其任务所执行的结点, 供下次使用. 等于是记录了处理迭代间驻留数据任务的历史以支持后来的本地性. 而本地数据的保存也是同样的原理.

算法 1.　Iterative Job Task Scheduling in MemLoop

```
Input: TaskTrackerStatus: status
//The current iteration's schedule;initially empty
Global variable:Map<Node,List<Partition>>current
//The Previous iteration's schedule
Global variable:Map<Node,List<Partition>>previous
If job process non invariant data then //如果是关于迭代内依存数据
  For reduce task: //作业
    hadoopSchedule(status.node);
  For Map task:
    List<Splits> cachedSplits=status.getCacheMapSplits()
    scheduleCacheMap(status.node,cachedSplits)
Else
  if iteration == 0 then
    Partition part = hadoopSchedule(node);
    current.get(node).add(part);
  Else
    if node.hasFullLoad() then
      Node substitution = findNearestIdleNode(node);
```

```
previous.get(substitution).addAll(previous.remove(node));
        return;
    end if
    if previous.get(node).size()>0 then
        Partition part = previous.get(node).get(0);
        schedule(part, node);
        current.get(node).add(part);
        previous.remove(part);
    end if
  end if
```

MemLoop 还需要考虑迭代内依存数据. 对迭代内依存数据使用缓存有较大的性能提升空间. 因此在调度上, MemLoop 若一个程序在迭代体内, 一个使用迭代内依存数据作业会根据其依存的数据所缓存的结点分配 map 任务. 产生迭代内依存数据的 reduce 任务会将其输出缓存到本地的内存上. 本文假设输入输出的文件格式为 FileInputFormat 或者 FileOutputFormat. 在 reduce 任务输出被缓存后, 计算这个文件对应的分片, 并由 TaskTracker 记录分片列表, 通过 InterTrackerProtocol 与 master 结点交互.

MemLoop 关于迭代的调度算法主要如算法 1 所示. MemLoop 将迭代内的作业分成两种类型, 一种是涉及了迭代间可驻留数据的作业, 一种作业是与迭代内依存数据相关的作业. MemLoop 针对定义的两种不同的作业类型分别采取不同的调度方法. 针对迭代内可驻留数据相关的作业, 按照 HaLoop 处理迭代式作业的方法进行. 针对与迭代内依存数据相关的作业, 则采取新的方法. 由于依存关系与 MapReduce 编程模型的特点. 调度要求负责生成迭代内依存数据的作业在生成结果时, 将数据缓存到任务运行的结点的内存或磁盘上. 对使用这些数据的作业, 则只需要将 Map 任务分配到其所处理文件被缓存的结点上启动. TaskTracker 需要跟踪在每个结点上被 Map/Reduce 任务处理的数据的相关信息. 在 TaskTracker 与 TaskScheduler 之间需要在 InterTrackerProtocol 上扩展相应的心跳信息用于获取 TaskTracker 中关于迭代间数据缓存的状态以支持算法 1 当中的关于作业之间依存关系的调度. 算法 1 当中的调度策略需要每个任务在缓存输出结果上有一个针对每个结点内存与磁盘的管理.

2.5 缓存管理模块

这个部分主要涉及 MemLoop 的缓存管理模块. HaLoop 中的缓存介质是每个结点的本地磁盘. 然而作为一个普通的外存设备, 磁盘在读写的延迟上与内存相比有一定差距. 在有空闲的内存情况下, 根据一定原则使用内存作为缓存介质对于整个迭代式应用的性能的改善更大.

在每个结点上, 有一个 CacheManager 模块用于在本地内存与本地磁盘上进行缓存管理. 内存的优势是在于其读取的性能, 磁盘的优势则在于其稳定性与容量大. 以内存作为读写的介质, 以磁盘作为换出介质则是将两种优势结合. 在任务运行比较多的情况下, 内存容量是一个瓶颈. 所以针对 CacheManager 的设计既充分利用内存以实现系统更高的读写性能的同时, 也要考虑到内存不足时的情况, 充分利用磁盘进行备份与换出.

如图 4 所示, CacheManager 分布三个部分: 缓存(Caching)模块、索引(Indexing)模块、替换(Replacement)模块.

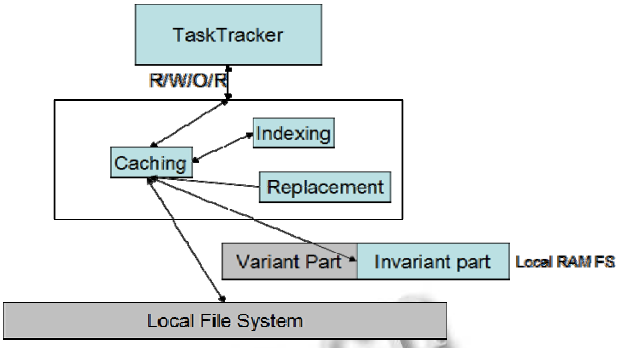


图 4 CacheManager 主要内部结构

缓存模块需要与 TaskTracker 进行交互, 实时报告内存缓存状态. 同时与 CacheManager 当中的其他功能进行交互. 向底层的两个文件管理系统: 内存的文件系统与本地磁盘文件系统缓存与删除数据, 读取数据等操作. 在一个作业完成时, 会撤销其使用的迭代内依存数据的缓存. 在一个迭代完成时, 才会撤销迭代间可驻留数据的缓存.

替换模块指的是数据缓存与换出模块. 这个替换模块针对两种不同类型的数据(即迭代间可驻留数据、迭代内依存数据)分别有两种缓存与替换的模块. 迭代计算处理了两种不同类型的数据, 所以 Replacement 的方法也针对两个底层文件缓存: Variant 部分与 Invariant 部分. 一个是迭代内依存数据的, 这些数据的缓存在内存上对应于内存文件当中的 Variant 部分. 这些数据特征是: 只与少数作业相关, 作业间存在读写关系, 所以往往是由上一个作业的 Reduce 完成写,

由下一个作业的 Map 任务读. 所以在将数据缓存到本地内存时, 其格式是同 Reduce 任务在 HDFS 上写的文件是一样的. 另一个是在迭代间可驻留数据, 即 Invariant 部分. 对迭代间可驻留数据缓存的处理, 包括 Reduce 端的输入、最后一个迭代的输出等, 这些数据由于计算的要求, 其缓存的格式与常规的 Reduce 输入输出并不完全相同. 这两个部分由于数据本身的性质不同, 其缓存的驻留时间与其处理特性是不同的. 因此将内存文件系统作为两个独立的部分, 分别使用不同的替换策略与缓存格式是必要的. 两个不同部分, 在内存容量不足时, 采取不同的替换策略. 在 Invariant 部分, 使用常见的缓存替换策略 LRU. 在 Variant 部分, 则不对现有驻留在内存的数据替换, 只把数据缓存到本地文件系统上, 因为这些数据在整个文件的驻留时间不长. 数据创建与删除比较频繁.

Indexing 模块是作为 HDFS 文件名到本地缓存的一个映射而存在的. 从作业提交到最终结束, 都需要相应地更新一下索引的状态. 在每个结点上, 当一个任务需要获得相应的数据时, 需要通过 Indexing 模块判断需要的文件是否在本地的缓存或者本地的磁盘上. Indexing 模块的变化要根据每次缓存与释放的操作进行.

三个模块通过 TaskTracker 将缓存的相关消息传送到 master 结点, 以获取全局状态. 根据迭代间可驻留数据与迭代内依存数据的状态, 基于 HaLoop 的 MemLoop 缓存管理在照顾性能提升与缓存容量上实现了一个较好的平衡.

3 实验对比

本文在一个初步的原型系统上验证上述基本结构在实际应用中的优化. 在实验中, PageRank 运行在一个 livejournal 数据集上, 通过对迭代次数的设置比较 MemLoop、Hadoop 与 HaLoop 的运行时间. 在硬件上, 实验运行在一个 3 结点的集群上, 其中一个结点作为主结点, 集群结点之间以 100Mb 的以太网进行连接.

在软件上, Hadoop 0.20.2 作为基础. 考虑到集群中只有 3 个结点, 实际用于存储数据的结点只有 2 个, 在 HDFS 的配置上, 将复本的备份数设置成 1. 关于底层用于缓存文件的内存, 使用 tmpfs 在每个结点上管理文件系统与本地的磁盘作为基本的介质, 以此为基础, 在调度、接口提交依据 HaLoop 进行一些扩展. 关于数据集, 从 soc-Livejournal 上获得了一个点对结构. 这个结构比较粗糙, 且容量不大只有 1G, 因此使用了一些替换的方法将其放大到 10G. 在作业提交上, 加入了对于作业间关于迭代内依存数据的依存关系描述的配置.

如图 5 所示, 同一个集群上, 分别针对 PageRank 的 Hadoop、HaLoop、MemLoop 的三种不同实现在不同的迭代次数下进行了比较. 从图中可以看到, MemLoop 在性能上随着迭代次数的变化, 相比 HaLoop 有了一定的优化. 相比 HaLoop 对 Hadoop 的优化, MemLoop 在性能上增加有限. 在未考虑对作业间依存关系的缓存时, 时间的减少几乎可以忽略. 在缓存作业间的缓存关系时, 则会对性能有一定的提升, 但是提升有限, 主要是由于本地文件系统均使用内存缓存作为优化, 从读写来看与内存文件系统的差距并不是特别大. 所以内存在这里的增长有限.

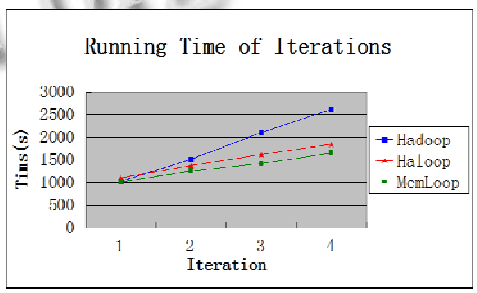


图 5　MemLoop/Haloop/Hadoop 执行 PageRank 时间

关于 HaLoop 的性能优化主要体现在其定义的三种缓存上: Reduce 输入缓存、Reduce 输出缓存、Map 输入缓存. MemLoop 则另外加入了迭代内依存数据. 关于作业间依存数据的缓存, 在其他类型的作业当中也可以考虑进行实现. 这样 MemLoop 当中的关于作业间的依赖的优化可以更广泛地适用到如 Hive 这样的结构当中.

4 相关工作

MemLoop 重点在通过已知整个迭代程序内部作业的依存关系实现数据缓存的优化. 利用作业间的依存关系进行的数据共享的优化已经有一些研究. 内存上的缓存与优化已有一些研究如 Tachyon[3]、Hadoop2.3[4]、GraphX[5]等. 我们从四个方面介绍一下与 MemLoop 相关的工作.

(1) 编程模型: 当前流行的分布式计算系统主要数据并行模型. 数据并行模型除了 MapReduce 之外, 还有 Dryad[6]、Pregel[7]、Spark[8]等擅长不同类型应用的计算框架. 这些应用框架提供的编程接口是操作分布式计算系统的底层原语. 为提高编程效率, 特别是兼容传统编程人员的习惯, 在这些分布式编程框架之上又有一些高层的编程框架, 如 Hive[9]、Pig Latin[10]、

Shark[11]等.

(2) 缓存的方式: 缓存作为计算当中提升性能的重要手段, 通常有不同的替换方法. 在分布式计算环境下, 各种不同的计算环境针对内存缓存有不同的方案. Spark[8]主要使用不同于 Hadoop 的一组 API, 这组 API 用于构造一个完整作业的操作 DAG. 中间结果自动缓存到内存上. HaLoop[2]基于 Hadoop 的结构, 引出了一些专门针对迭代的 API, 更主要的是由于其本身考虑到在运行过程当中, 关于数据的缓存主要是以在迭代间多次被引用的为主要处理, 在 API 上也显式指定需要缓存的文件. 在 Hadoop 2.3 当中, 又有了关于 HDFS 上的内存缓存管理. 以上缓存的方式或者由API直接指定, 或者在常用的 MR 接口之外另使用一组 API. MemLoop 的主要结构是基于 Haloop 的 API 进行缓存并对于作业间的依存关系有更明确的描述用于指导底层运行时在调度与缓存时考虑到作业的依存关系.

(3) 缓存介质: 在 CPU 层面, 通常是使用 SRAM 进行缓存的. 在分布式计算系统当中, 不同的计算系统使用不同的缓存介质. 在 Haloop 下, 使用本地磁盘作为缓存. 在 Spark[8]等系统当中, 主要使用内存作为缓存介质.

(4) 基于内存的文件系统: 基于内存的分布式缓存系统有 MemCached[12]、Redis[13]等, 这些系统是一些基于内存的、键值对存储数据库. Tachyon[3]是一个运行在 HDFS 之上的分布式内存文件系统. Pacman[14]是一个用于协调分布式系统上缓存的一个程序.

5 结语

MemLoop 编程框架针对分布式迭代计算的一些特征及集群环境特征提出的. 本文主要思路是通过以内存缓存降低作业读写的延迟从而提升应用的性能. 迭代缓存的数据分为迭代间可驻留数据与迭代内依存数据. 迭代内依存数据相关的作业在时序上离得较近, 使用分布式文件系统存在读写性能问题. 因此针对两种数据 MemLoop 用相应的 API 与调度算法确定缓存的数据并将任务分配到缓存所在的结点上. 针对服务器程序当中多数情况下比较多的空闲内存现象. 将内存作为缓存的一个介质在底层缓存相应的数据. 与 HaLoop 不同, MemLoop 因为使用内存, 需要专门处理内存不足的情况. 通过上述的设计与实验表明, MemLoop 对于迭代式应用的性能有一定的提升.

参考文献

1 Dean J, Ghemawat S. Mapreduce: Simplified data processing on large clusters. Communications of the ACM, 2008, 51(1): 107–113.

2 Bu Y, Howe B, Balazinska M, Ernst MD. Haloop: Efficient iterative data processing on large clusters. Proc. of the VLDB Endowment, 2010, 3(1-2): 285–296.

3 Li H, Ghodsi A, Zaharia M, Baldeschwieler E, Shenker S, Stoica I. Tachyon: Memory throughput I/O for cluster computing frameworks. Memory, 2013, 18: 1.

4 Apache. Hadoop 2.3.0. http://hadoop.apache.org/docs/r2.3.0/. 2014.

5 Xin RS, Gonzalez JE, Franklin MJ, Stoica I. Graphx: A resilient distributed graph system on spark. First International Workshop on Graph Data Management Experiences and Systems. ACM. 2013. 2.

6 Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: Distributed data-parallel programs from sequential building blocks. ACM SIGOPS Operating Systems Review, 2007, 41(3): 59–72.

7 Malewicz G, Austern MH, Bik AJ, Dehnert JC, Horn I, Leiser N, Czajkowski G. Pregel: A system for large-scale graph processing. Proc. of the 2010 ACM SIGMOD International Conference on Management of Data. ACM. 2010. 135–146.

8 Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Proc. of the 9th USENIX Conference on Networked Systems Design and Implementation. USENIX Association. 2012. 2–2.

9 Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R. Hive: A warehousing solution over a map-reduce framework. Proc. of the VLDB Endowment, 2009, 2(2): 1626–1629.

10 Olston C, Reed B, Srivastava U, Kumar R, Tomkins A. Pig latin: A not-so-foreign language for data processing. Proc. of the 2008 ACM SIGMOD International Conference on Management of Data. ACM. 2008. 1099–1110.

11 Xin R. Shark makes hive faster and more powerful. http://shark.cs.berkeley.edu/.

12 Fitzpatrick B. Distributed caching with memcached. Linux Journal, 2004, (124): 5.

13 Sanfilippo S, Noordhuis P. Redis: The Definitive Guide, 2010.

14 Ananthanarayanan G, Ghodsi A, Wang A, Borthakur D, Kandula S, Shenker S, Stoica I. Pacman: Coordinated memory caching for parallel jobs. USENIX NSDI. 2012.