Google MapReduce 论文详解

MapReduce 编程模型

总的来讲，Google MapReduce 所执行的分布式计算会以一组键值对作为输入，输出另一组键值对，用户则通过编写 Map 函数和 Reduce 函数来指定所要进行的计算。

由用户编写的Map 函数将被应用在每一个输入键值对上，并输出若干键值对作为中间结果。之后，MapReduce 框架则会将与同一个键 I 相关联的值都传递到同一次 Reduce 函数调用中。

同样由用户编写的 Reduce 函数以键 I 以及与该键相关联的值的集合作为参数，对传入的值进行合并并输出合并后的值的集合。

形式化地说，由用户提供的 Map 函数和 Reduce 函数应有如下类型：

$$\begin{aligned} \text{map} \quad & (k1, v1) \quad \rightarrow \text{list}(k2, v2) \\ \text{reduce} \quad & (k2, \text{list}(v2)) \quad \rightarrow \text{list}(v_2) \end{aligned}$$

值得注意的是，在实际的实现中 MapReduce 框架使用 Iterator 来代表作为输入的集合，主要是为了避免集合过大，无法被完整地放入到内存中。

作为案例，我们考虑这样一个问题：给定大量的文档，计算其中每个单词出现的次数（Word Count）。用户通常需要提供形如如下伪代码的代码来完成计算：

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, “1”);


reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

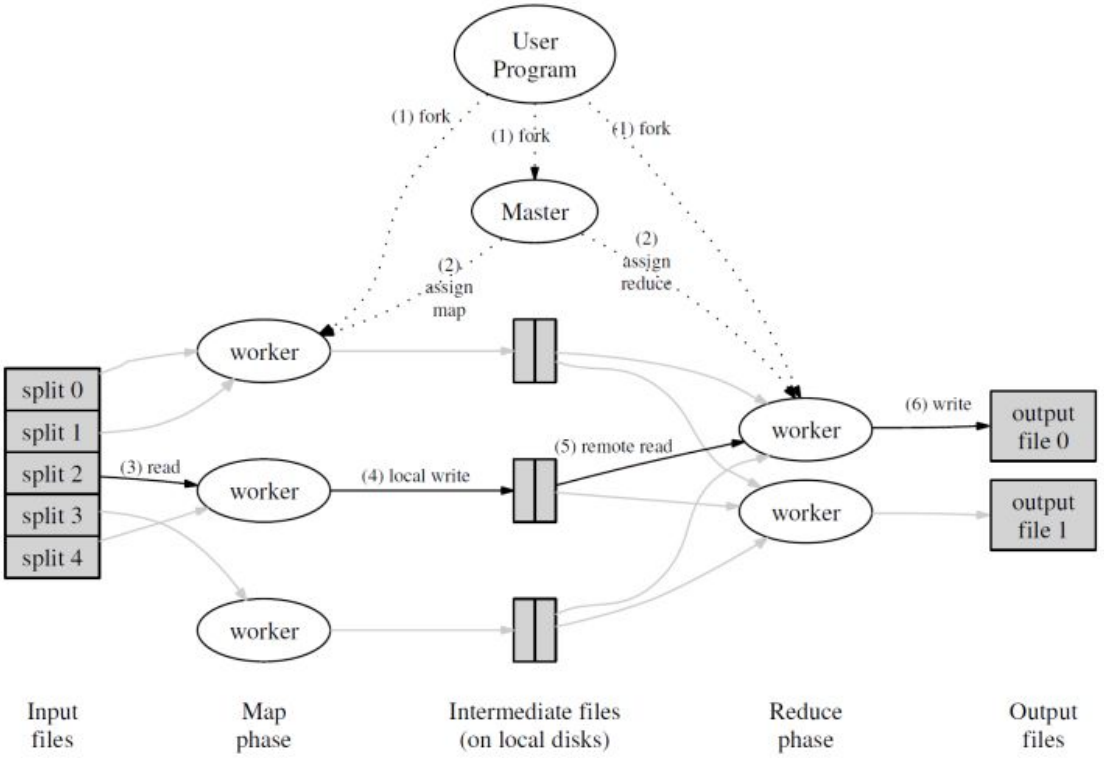
函数式编程模型

了解函数式编程范式的读者不难发现，MapReduce 所采用的编程模型源自于函数式编程里的 Map 函数和 Reduce 函数。后起之秀 Spark 同样采用了类似的编程模型。

使用函数式编程模型的好处在于这种编程模型本身就对并行执行有良好的支持，这使得底层系统能够轻易地将大数据量的计算并行化，同时由用户函数所提供的确定性也使得底层系统能够将函数重新执行作为提供容错性的主要手段。

MapReduce 计算执行过程

每一轮 MapReduce 的大致过程如下图所示：



首先，用户通过 MapReduce 客户端指定 Map 函数和 Reduce 函数，以及此次 MapReduce 计算的配置，包括中间结果键值对的 Partition 数量 R 以及用于切分中间结果的哈希函数 $hash$。
用户开始 MapReduce 计算后，整个 MapReduce 计算的流程可总结如下：

1. 作为输入的文件会被分为 M 个 Split，每个 Split 的大小通常在 16~64 MB 之间
2. 如此，整个 MapReduce 计算包含 M 个Map 任务和 R 个 Reduce 任务。Master 结点会从空闲的 Worker 结点中进行选取并为其分配 Map 任务和 Reduce 任务
3. 收到 Map 任务的 Worker 们（又称 Mapper）开始读入自己对应的 Split，将读入的内容解析为输入键值对并调用由用户定义的 Map 函数。由 Map 函数产生的中间结果键值对会被暂时存放在缓冲内存区中
4. 在 Map 阶段进行的同时，Mapper 们周期性地将放置在缓冲区中的中间结果存入到自己的本地磁盘中，同时根据用户指定的 Partition 函数（默认为 $hash(\text{key}) \bmod R$）将产生的中间结果分为 R 个部分。任务完成时，Mapper 便会将中间结果在其本地磁盘上的存放位置报告给 Master
5. Mapper 上报的中间结果存放位置会被 Master 转发给 Reducer。当 Reducer 接收到这些信息后便会通过 RPC 读取存储在 Mapper 本地磁盘上属于对应 Partition 的中间结果。在读取完毕后，Reducer 会对读取到的数据进行排序以令拥有相同键的键值对能够连续分布
6. 之后，Reducer 会为每个键收集与其关联的值的集合，并以之调用用户定义的 Reduce 函数。Reduce 函数的结果会被放入到对应的 Reduce Partition 结果文件

实际上，在一个 MapReduce 集群中，Master 会记录每一个 Map 和 Reduce 任务的当前完成状态，以及所分配的 Worker。除此之外，Master 还负责将 Mapper 产生的中间结果文件的位置和大小转发给 Reducer。

值得注意的是，每次 MapReduce 任务执行时， M 和 R 的值都应比集群中的 Worker 数量要高得多，以达成集群内负载均衡的效果。

MapReduce 容错机制

由于 Google MapReduce 很大程度上利用了由 Google File System 提供的分布式原子文件读写操作，所以 MapReduce 集群的容错机制实现相比之下便简洁很多，也主要集中在任务意外中断的恢复上。

Worker 失效

在 MapReduce 集群中，Master 会周期地向每一个 Worker 发送 Ping 信号。如果某个 Worker 在一段时间内没有响应，Master 就会认为这个 Worker 已经不可用。

任何分配给该 Worker 的 Map 任务，无论是正在运行还是已经完成，都需要由 Master 重新分配给其他 Worker，因为该 Worker 不可用也意味着存储在该 Worker 本地磁盘上的中间结果也不可用了。Master 也会将这次重试通知给所有 Reducer，没能从原本的 Mapper 上完整获取中间结果的 Reducer 便会开始从新的 Mapper 上获取数据。

如果有 Reduce 任务分配给该 Worker，Master 则会选取其中尚未完成的 Reduce 任务分配给其他 Worker。鉴于 Google MapReduce 的结果是存储在 Google File System 上的，已完成的 Reduce 任务的结果的可用性由 Google File System 提供，因此 MapReduce Master 只需要处理未完成的 Reduce 任务即可。

Master 失效

整个 MapReduce 集群中只会有一个 Master 结点，因此 Master 失效的情况并不多见。

Master 结点在运行时会周期性地将集群的当前状态作为保存点（Checkpoint）写入到磁盘中。Master 进程终止后，重新启动的 Master 进程即可利用存储在磁盘中的数据恢复到上一次保存点的状态。

落后的 Worker

如果集群中有某个 Worker 花了特别长的时间来完成最后的几个 Map 或 Reduce 任务，整个 MapReduce 计算任务的耗时就会因此被拖长，这样的 Worker 也就成了落后者（Straggler）。

MapReduce 在整个计算完成到一定程度时就会将剩余的任务进行备份，即同时将其分配给其他空闲 Worker 来执行，并在其中一个 Worker 完成后将该任务视作已完成。

其他优化

在高可用的基础上，Google MapReduce 系统现有的实现同样采取了一些优化方式来提高系统运行的整体效率。

数据本地性

在 Google 内部所使用的计算环境中，机器间的网络带宽是比较稀缺的资源，需要尽量减少在机器间过多地进行不必要的数据传输。

Google MapReduce 采用 Google File System 来保存输入和结果数据，因此 Master 在分配 Map 任务时会从 Google File System 中读取各个 Block 的位置信息，并尽量将对应的 Map 任务分配到持有该 Block 的 Replica 的机器上；如果无法将任务分配至该机器，Master 也会利用 Google File System 提供的机架拓扑信息将任务分配到较近的机器上。

Combiner

在某些情形下，用户所定义的 Map 任务可能会产生大量重复的中间结果键，同时用户所定义的 Reduce 函数本身也是满足交换律和结合律的。

在这种情况下，Google MapReduce 系统允许用户声明在 Mapper 上执行的 Combiner 函数：Mapper 会使用由自己输出的 R 个中间结果 Partition 调用 Combiner 函数以对中间结果进行局部合并，减少 Mapper 和 Reducer 间需要传输的数据量。

分类： 大数据技术