MapReduce

1 概述

MapReduce 是用于处理和生成大型数据集的编程模型和相关的实现。 用户指定一个处理键/值对以生成一组中间键/值对的 Map 函数，以及一个 Reduce 函数，该 Reduce 函数合并与同一中间键关联的所有中间值。在此模型中可以表达许多现实世界中的任务，例如索引建立，分布式排序，数据统计。

以这种功能风格编写的程序会自动并行化，并在大型企业计算机集群上执行。 运行时系统负责划分输入数据，安排程序在一组机器上的执行，处理机器故障以及管理所需的机器间通信的细节。 这使没有并行和分布式系统经验的程序员可以轻松利用大型分布式系统的资源。

该计算采用一组输入键/值对，并产生一组输出键/值对。 MapReduce 库的用户将计算表示为两个函数：Map 和 Reduce。

由用户编写的 Map 接受一个输入对，并产生一组中间键/值对。MapReduce 库将与同一中间键 I 关联的所有中间值分组在一起，并将它们传递给 Reduce 函数。

同样由用户编写的 Reduce 函数接受中间键 I 和该键的一组值。它将这些值合并在一起以形成可能较小的一组值。 通常，每个 Reduce 调用仅产生零或一个输出值。中间值通过迭代器提供给用户的 reduce 函数。这使我们能够处理超出内存大小的值列表。

2 实现

2.1 执行流程概述

通过将输入数据自动划分为一组 M 个分片（spilt），Map 调用分布在多台计算机上。

输入分片（spilt）可以由不同的机器并行处理。通过使用 partitioning 函数（例如，hash（key）mod R）将中间键空间划分为 R 个片段（piece）。 分区数（R）和 partitioning 函数由用户指定。

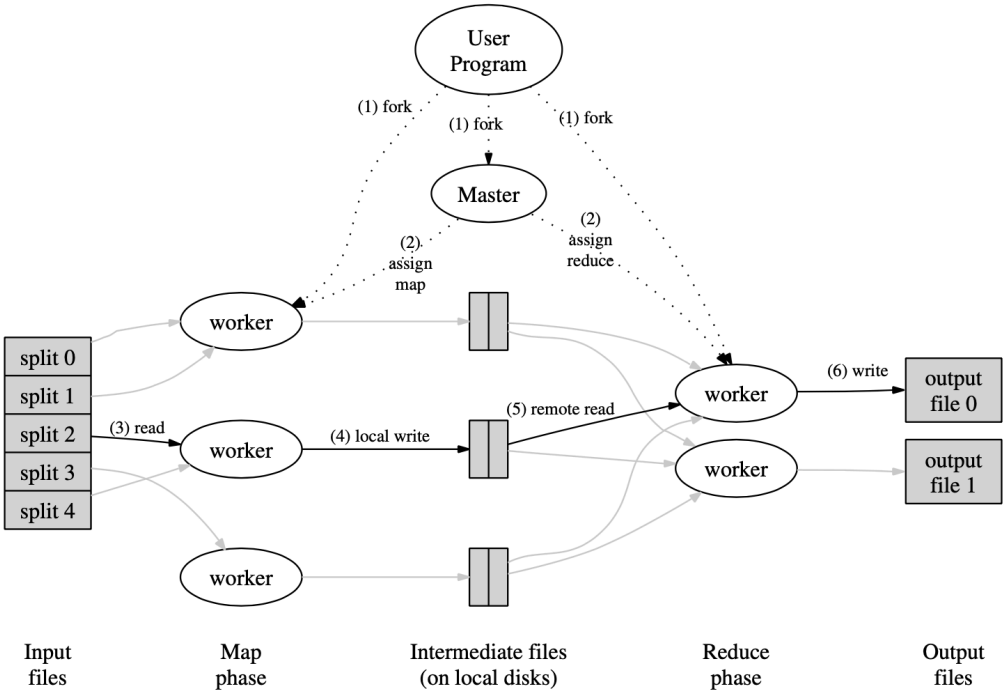

图 1 显示了我们实施中 MapReduce 操作的总体流程。 当用户程序调用 MapReduce 函数时，将发生以下操作序列（图 1 中的编号标签与以下列表中的数字相对应）

1. 用户程序中的 MapReduce 库首先将输入文件拆分为 M 个文件，每个文件通常为 16 兆字节至 64 兆字节（MB）（可由用户通过可选参数进行控制）。 然后，它会在多个计算机上启动该程序的许多副本。
2. 该程序的副本之一是特殊的-Master。 其余的是由 Master 分配工作的 Worker。 要分配 M 个 Map 任务和 R 个 Reduce 任务。 Master 选择空闲的 Worker，并为每个 Worker 分配一个 Map 任务或一个 Reduce 任务。
3. 分配了 Map 任务的 Worker 将读取相应输入拆分的内容。它从输入数据中解析键/值对，并将每对传递给用户定义的 Map 函数。由 Map 函数产生的中间键/值对被缓存在内存中。

4. 缓冲会定期地写入本地磁盘，并通过 partitioning 函数划分为 R 个区域。这些缓冲对在本地磁盘上的位置被传递回 Master，该 Master 负责将这些位置转发给 reduce worker。
5. 当 Master 通知 reduce worker 这些位置时，Reduce Worker 将使用 RPC 从 Map Worker 的本地磁盘读取缓冲的数据。 当 reduce worker 读取了所有中间数据时，它将按中间键对其进行排序，以便将同一键的所有出现都分组在一起。 需要排序是因为通常许多不同的键映射到相同的 reduce 任务。 如果中间数据量太大而无法容纳在内存中，则使用外部排序。
6. Reduce Worker 迭代排序后的中间数据，并针对遇到的每个唯一的中间键，将键和相应的中间值集传递给用户的 Reduce 函数。 Reduce 函数的输出将附加到此 reduce 分区的最终输出文件中。
7. 完成所有 Map 任务和 Reduce 任务后，Master 将唤醒用户程序。 此时，用户程序中的 MapReduce 调用将返回到用户代码。

任务成功完成后，映射执行的输出在 R 输出文件中可用（每个 Reduce 任务一个，其文件名由用户指定）。 通常，用户不需要将这些 R 输出文件合并为一个文件–他们通常将这些文件作为输入传递给另一个 MapReduce 调用，或者从另一个能够处理被划分为多个文件的分布式应用程序中使用它们 。

2.2 Master 数据结构

Master 保留几个数据结构。对于每个 map 任务和 reduce 任务，它存储状态（空闲，进行中或已完成）和 Worker 机器的标识（对于非空闲任务）。

Maste 是将 Map 任务中间文件位置传递给 Reduce 任务的通道。因此，对于每个完成的 Map 任务，Master 会存储由 Map 任务生成的 R 个中间文件区域的位置和大小。Map 任务完成后，将接收到此位置和大小信息的更新。信息将逐步推送给正在进行 Reduce 任务的 Worker。

2.3 容错

由于 MapReduce 库旨在使用数百或数千台计算机处理大量数据，因此该库必须能够容忍机器故障。

Worker 故障

Master 定期对每个工人执行 ping 操作。如果在一定时间内未收到 Worker 的任何响应，则 Master 将 Worker 标记为失败。由该 Worker 完成的所有 Map 任务都将重置为初始的空闲状态，因此有资格安排其他 Worker。同样，在失败的工作程序上进行的任何 Map 任务或 Reduce 任务也将重置为空闲，并有资格进行重新调度。

完成的 Map 任务会在失败时重新执行，因为它们的输出存储在故障机器的本地磁盘上，因此无法访问。 而已完成的 Reduce 任务的输出文件存储在全局文件系统中，因此无需重新执行。

当首先由 Worker A 执行 Map 任务，然后再由 Worker B 执行 Map 任务（因为 A 失败）时，将向所有执行 reduce 任务的工作程序重新执行通知。任何尚未从 Worker A 读取数据的 reduce 任务都将从 Worker B 读取数据。

MapReduce 可以抵抗大规模的 Worker 故障。例如，在一次 MapReduce 操作期间，正在运行的集群上的网络维护导致一次由 80 台计算机组成的组在数分钟内无法访问。 MapReduce Master 只是简单地重新执行了无法访问的 Worker 所完成的工作，并继续取得进步，最终完成了 MapReduce 操作。

Master 故障

使 Master 很容易将上述主数据结构写入定期检查点（Checkpoint）。如果 Master 死亡，则可以从最后一个检查点状态开始新的副本。但是，由于只有一个 Master，因此发生故障的可能性不大。 因此，如果 Master 失败，则当前的实现会中止 MapReduce 计算。 客户可以检查这种情况，并根据需要重试 MapReduce 操作。

语义中出现的失败

当用户提供的 map 和 reduce 运算符是其输入值的确定性函数时，我们的分布式实现将产生与整个程序的无故障顺序执行相同的输出。

我们依靠 map 和 reduce 认为的原子提交以实现此属性。每个正在进行的任务都会将其输出写入私有临时文件。Reduce 任务产生一个这样的文件，而 Map 任务产生 R 个这样的文件（每个精简任务一个）。Map 任务完成后，Worker 会向主服务器发送一条消息，并在消息中包含 R 个临时文件的名称。 如果 Master

收到有关已完成的 Map 任务的完成消息，它将忽略该消息。否则，它将在 Master 数据结构中记录 R 文件的名称。

当 reduce 任务完成时，reduce worker 自动将其临时输出文件重命名为最终输出文件。如果在多台计算机上执行相同的 reduce 任务，则将对同一最终输出文件执行多个重命名调用。我们依靠基础文件系统提供的原子重命名操作来确保最终文件系统状态仅包含一次执行 reduce 任务所产生的数据。

我们的 map 和 reduce 运算符绝大多数是确定性的，在这种情况下我们的语义等同于顺序执行的事实使程序员很容易就其程序行为进行推理。当 map 和/或 reduce 运算符不确定时，我们将提供较弱但仍然合理的语义。在存在不确定性运算符的情况下，特定 Reduce 任务 R1 的输出等效于由不确定性程序的顺序执行产生的 R1 的输出。然而，用于不同 Reduce 任务 R2 的输出可以对应于由不确定性程序的不同顺序执行所产生的用于 R2 的输出。

考虑 Map 任务 M 和 Reduce 任务 R1 和 R2。令 e（Ri）为已落实的 Ri 的执行（恰好有一个这样的执行）。之所以出现较弱的语义，是因为 e（R1）可能已经读取了由 M 的一次执行所产生的输出，而 e（R2）可能已经读取了由 M 的不同执行所产生的输出。

2.4 数据局部性

在我们的计算环境中，网络带宽是相对稀缺的资源。 通过利用输入数据（由 GFS [8]管理）存储在组成集群的计算机的本地磁盘上这一事实，可以节省网络带宽。 GFS 将每个文件划分为 64 MB 的块，并将每个块的多个副本（通常为 3 个副本）存储在不同的计算机上。 MapReduce Master 会考虑输入文件的位置信息，并尝试在包含相应输入数据副本的计算机上安排 Map 任务。 如果失败，它尝试将地图任务安排在该任务输入数据的副本附近（例如，在与包含数据的计算机位于同一网络交换机的工作机上）。在集群中很大一部分的工作线程上运行大型 MapReduce 操作时，大多数输入数据都在本地读取，并且不占用网络带宽。

2.5 任务粒度

如上所述，我们将映射阶段细分为 M 个片段（piece），将 Reduce 阶段细分为 R 个片段。 理想情况下，M 和 R 应该比工作计算机的数量大得多。让每个工作程序执行许多不同的任务可以改善动态负载平衡，并且还可以在工作程序失败时加快恢复速度：它完成的许多映射任务可以分布在所有其他工作程序计算机上。

如上所述，由于主机必须制定 O（M+R）调度决策并将 O（M∗R）状态保持在内存中，因此在我们的实现中可以有多大的 M 和 R 有实际的界限。（但是，用于内存使用的常量因子很小：Map/Reduce 任务对的状态的部分包含大约一个字节的数据。）

此外，R 通常受用户约束，因为每个 reduce 任务的输出最终都存储在单独的输出文件中。 在实践中，我们倾向于选择 M，以便每个单独的任务大约是 16 MB 到 64 MB 的输入数据（这样，上述的局部优化是最有效的，并且我们将 R 设为 Worker 数量的一小部分。 我们通常使用 2,000 个工作机，执行 M = 200,000 和 R = 5,000 的 MapReduce 计算.

2.6 备份任务

延长 MapReduce 操作总时间的常见原因之一是“落后”（straggle）现象：一台机器花费异常长的时间才能完成最后几个 Map 之一或 Reduce 计算任务。落后之所以出现，可能是由于多种原因。例如，磁盘损坏的计算机可能会经常发生可纠正的错误，从而将其读取性能从 30 MB/s 降低到 1 MB/s。 集群调度系统可能已经在计算机上调度了其他任务，由于竞争 CPU，内存，本地磁盘或网络带宽，导致其执行 MapReduce 代码的速度较慢。 我们最近遇到的一个问题是机器初始化代码中的一个错误，该错误导致了处理器缓存的禁用：受影响机器的计算速度降低了一百倍。

我们有一个通用的机制来缓解落后者的问题。当 MapReduce 操作接近完成时，主服务器会调度其余正在进行的任务的备份执行。每当原始执行任务或备份执行任务完成时，该任务就会标记为已完成。 我们已经对该机制进行了调整，以使其通常将操作使用的计算资源增加不超过百分之几。 我们发现这大大减少

了完成大型 MapReduce 操作的时间。 例如，禁用备份任务机制后，第 5.3 节中描述的排序程序将花费 44％的时间才能完成。

3 扩展

尽管只需编写 Map 和 Reduce 函数即可满足大多数需求，但我们发现一些扩展很有用。这些将在本节中介绍。

3.1 Partitioning 函数

MapReduce 的用户指定他们想要的 Reduce 任务/输出文件的数量 R。通过在中间结果熵使用 Partitioning 函数可将数据划分到这些任务中。MapReduce 提供了使用散列的默认 Partitioning 函数（例如“ hash（key）mod R”）。这往往会导致分区非常均衡。但是，在某些情况下，通过键的其他功能对数据进行分区很有用。 例如，有时输出键是 URL，我们希望单个主机的所有条目都以同一输出文件结尾。为了支持这种情况，MapReduce 库的用户可以提供特殊的分区功能。例如，将“ hash（Hostname（urlkey））mod R”用作分区函数会使来自同一主机的所有 URL 最终都位于同一输出文件中。

3.2 排序保证

我们保证在给定的分区内，中间键/值对以递增的键顺序进行处理。这种排序保证使您可以轻松地为每个分区生成排序的输出文件，当输出文件格式需要通过键支持有效的随机访问查找，或者输出的用户发现对数据进行排序很方便时，这很有用。

3.3 合并（Combiner）函数

在某些情况下，每个映射任务产生的中间键会大量重复，并且用户指定的 Reduce 函数具有可交换性和关联性。一个很好的例子是 2.1 节中的单词计数示例。由于单词频率倾向于遵循 Zipf 分布，因此每个 Map 任务都会生成成百上千的<

the，1>形式的记录。 所有这些计数将通过网络发送到单个 Reduce 任务，然后通过 Reduce 函数加在一起以产生一个数字。 我们允许用户指定一个可选的 Combiner 函数，该函数会在通过网络发送数据之前进行部分合并。

在执行 Map 任务的每台计算机上执行合并器功能。通常，使用相同的代码来实现 Combiner 和 Reduce 功能。Reduce 函数和组合函数之间的唯一区别是 MapReduce 库如何处理函数的输出。reduce 函数的输出将写入最终输出文件。Combiner 功能的输出将写入中间文件，该文件将发送到 reduce 任务。部分组合可显着加快某些类的 MapReduce 操作。

3.4 输入输出类型

MapReduce 库提供了几种不同格式的读取输入数据。例如，“text”格式的输入将每一行看作 key/value 对，Key 是每一行数据在文件中的偏移，Value 是每一行的内容。另一种常用的受支持格式存储按键排序的键/值对序列，每个输入类型的实现都知道如何将自己拆分为有意义的范围，以作为单独的 Map 任务进行处理（例如，“text”模式的范围拆分可确保范围拆分仅在行边界处发生）。用户可以通过提供简单阅读器界面的实现来增加对新输入类型的支持，尽管大多数用户仅使用少量预定义输入类型中的一种。

reader 不一定需要提供从文件读取的数据。例如，定义一个 reader 很容易，它可以从数据库或从内存中映射的数据结构读取记录。

以类似的方式，我们支持一组输出类型，用于生成不同格式的数据，并且用户代码很容易添加对新输出类型的支持。

3.5 副作用

在某些情况下，MapReduce 的用户发现可以方便地生成辅助文件作为其 Map 和/或 reduce 运算符的附加输出。我们依靠应用程序编写者来使此类副作用成为原子和幂等的。通常，应用程序将写入临时文件，并在文件完全生成后以原子方式重命名该文件。

我们不支持由单个任务生成的多个输出文件的原子两阶段提交。 因此，产生具有跨文件一致性要求的多个输出文件的任务应该是确定的。这种限制在实践中从未成为问题。

3.6 跳过 Bad 记录

有时，用户代码中的错误会导致 Map 或 Reduce 函数在某些记录上确定性地崩溃。此类错误会阻止 MapReduce 操作完成。通常的做法是修复该错误，但是有时这是不可行的。 该错误可能存在于第三方库中，该库的源代码不可用。 此外，有时可以忽略一些记录，例如在对大型数据集进行统计分析时。 我们提供了一种可选的执行模式，其中 MapReduce 库检测导致确定性崩溃的记录，并按顺序跳过这些记录以取得进展。

每个工作进程都安装一个信号处理程序，以捕获分段违规和总线错误。 在调用用户 Map 或 Reduce 操作之前，MapReduce 库将参数的序列号存储在全局变量中。如果用户代码生成一个信号，信号处理程序将包含序列号的“最后喘息（last gasp）” UDP 数据包发送到 MapReduce Master。当 Master 在特定记录上看到多个故障时，表明该记录在下一次重新执行相应的 Map 或 Reduce 任务时应跳过该记录。

3.7 本地执行（测试）

Map 或 Reduce 函数中的调试问题可能很棘手，因为实际的计算是在分布式系统中进行的，通常是在几千台机器上，而工作分配决定是由主服务器动态地做出的。 为了帮助简化调试，性能分析和小规模测试，我们开发了 MapReduce 库的另一种实现，该库可在本地计算机上顺序执行 MapReduce 操作的所有工作。向用户提供控件，以便可以将计算限制在特定的地图任务中。 用户使用特殊标志调用程序，然后可以轻松使用他们认为有用的任何调试或测试工具（例如 gdb）

3.8 状态信息

Master 运行一个内部 HTTP 服务器，并导出一组状态页面供人类使用。 状态页面显示了计算进度，例如完成了多少任务，正在进行多少任务，输入字节，中间数据字节，输出字节，处理速率等。这些页面还包含链接到每个任务生成的标准错误和标准输出文件。用户可以使用此数据来预测计算将花费多长时间，以及是否应将更多资源添加到计算中。这些页面还可用于确定何时计算速度比预期的慢得多。

此外，顶层状态页面显示哪些工作人员发生了故障，以及在工作人员发生故障时映射并减少了他们正在处理的任务。尝试诊断用户代码中的错误时，此信息很有用。

3.9 计数器

MapReduce 库提供了一种计数器功能，可以对各种事件的发生进行计数。例如，用户代码可能想要计算处理的单词总数或索引的德语文档的数量等。

要使用此功能，用户代码将创建一个命名计数器对象，然后在 Map 和/或 Reduce 函数中适当地增加计数器。 例如：

```
1.  Counter* uppercase;
2.  uppercase = GetCounter("uppercase");
3.  map(String name, String contents):
4.    for each word w in contents:
5.      if (IsCapitalized(w)):
6.        uppercase->Increment();
7.      EmitIntermediate(w, "1");
```

来自各个工作计算机的计数器值会定期传播到 Master（在 ping 响应上）。 主服务器从成功的 Map 和 Reduce 任务中聚合计数器值，并在 MapReduce 操作完成时将其返回给用户代码。当前的计数器值也显示在主状态页面上，以便人们可以观察实时计算的进度。在汇总计数器值时，Master 消除了重复执行同一 Map 或 Reduce 任务以避免重复计数的影响。重复执行可能是由于我们使用备份任务以及由于失败而重新执行任务引起的。）

一些计数器值由 MapReduce 库自动维护，例如处理的输入键/值对的数量和产生的输出键/值对的数量。

用户发现计数器功能对于完整性检查MapReduce操作的行为很有用。例如，在某些 MapReduce 操作中，用户代码可能想要确保所生成的输出对的数量完全等于所处理的输入对的数量，或者所处理的德语文档在已处理文件总数的比例。