MapReduce Online

摘要

MapReduce 是一种流行的架构，用于批处理作业的数据密集型分布式计算。为了简化容错，每个 MapReduce 任务和工作的输出都在其被消耗前实体化到硬盘。本文中，我们提出一种改进的 MapReduce 体系，允许数据在操作之间用管道传送。这样扩展了优于批处理的 MapReduce 编程模型，并能够减少完成时间，以及提高批量工作的系统利用率。我们提出了 Hadoop MapReduce 架构的一种支持联机聚集的改进版本，它允许用户看到工作被计算时的“早期返回”。我们的 Hadoop Online Prototype（HOP）还支持连续查询，这使得 MapReduce 程序可被用于事件监测和流处理等应用。HOP 保留了 Hadoop 的容错特性，并可运行未改进的用户定义 MapReduce 程序。

1. 介绍

MapReduce 已经成为一种控制大型计算机集群能力的流行方式。MapReduce 允许程序员以数据为中心的方式思考：他们注重于应用数据记录集的转换，并允许由 MapReduce 架构处理分布式执行、网络通信、协调和容错等细节。

MapReduce 模型通常应用于主要关注工作完成时间的大型批量化计算。Google MapReduce 架构和开源 Hadoop 系统通过批处理实施策略来加强该使用模型：每个 map 和 reduce 阶段的全部输出在下一阶段被消耗前就被实体化到稳定存储器中，或者产生输出。批量实体化允许一种简单而讲究的检查点/重启容错机制，其对于大型部署十分关键，在 worker 节点拥有怠工或故障的高概率。

我们提出了一种改进 MapReduce 体系，其中间数据在操作间用管道传送，同时保留了之前 MapReduce 架构的编程接口和容错模式。为了验证该设计，我们开发了 Hadoop Online Prototype（HOP）——Hadoop 的一种管道版本。

管道提供了一些对于 MapReduce 架构的重要优势，但也提出了新的设计挑战。我们首先强调潜在优势：

- 下游数据元素可以在 producer 元素完成执行前开始消耗数据，这可以增加并行机会、提高利用率、减少响应时间。在 3.5 节，我们将展示 HOP 相比于未改进 Hadoop 可以显著降低工作完成时间。
- 由于 mappers 一产生数据后 reducers 就开始处理，它们可以在执行工程中生成并改善其最终结果的近似值。这种称为联机聚集的技术可以减少几个数量级的数据分析周转时间。在第 4 节我们将描述我们的管道 MapReduce 体系如何适应联机聚集。
- 管道拓宽了 MapReduce 可被应用的领域。在第 5 节，我们将展示 HOP 如何被用于连续查询：MapReduce 工作连续运行，当新数据到达时便接收并直接分析之。这允许 MapReduce 被应用于系统监测和流处理等领域。

管道提出了一些设计挑战。首先，Google 的简单 MapReduce 容错机制的前提是中间状态的实体化。在 3.3 节，我们将展示其能够与管道并存，通过允许 producers 在实体化的同时周期性发送数据给 consumers。第二个挑战来自于隐含在管道中的贪婪通信，这和由

“combiners”支持的批量优化有差异。map 端代码在通信前通过压缩和预聚集来降低网络利用率。我们将在 3.1 节讨论 HOP 设计如何解决该问题。最后，管道需要 producers 和 consumers 智能地联合调度；我们将在 3.4 节讨论对该问题的初步研究。

1.1 本文结构

为了展开我们的讨论，我们在第 2 节提出 Hadoop MapReduce 体系的概述，然后我们在第 3 节开发 HOP 管道方案设计，它保持对传统批处理任务的专注，并显示管道在该设置后能提供的性能提升。解释了 HOP 执行模式后，我们将在第 4 节展示它如何支持长期工作的联机聚集，并说明该 MapReduce 任务接口的潜在利益。第 5 节我们描述对连续 MapReduce 工作数据流的支持，并展示一个近实时并行系统监测的例子。相关和未来研究包括在第 6 和 7 节。

2. 背景

MapReduce 是一种对大型数据集进行转换的编程模型。在本节，我们将回顾 MapReduce 编程模型，并描述 Hadoop 的主要特点——一种流行的开源 MapReduce 实现。

2.1 编程模型

为了使用 MapReduce，程序员将其期望的计算表示成一系列工作。工作的输入时一列记录（键-值对）。每个工作由两个步骤组成：第一，用户定义的 map 函数被应用于每个记录来产生一系列中间键-值对。第二，用户定义的 reduce 函数被 map 输出的每个不同键调用一次，并通过一列与该键相关的中间值。MapReduce 机构自动并行这些函数的执行，并保证容错。

或者，用户可提供一个 combiner 函数，Combiners 相似于 reduce 函数，但它们不能通过所有给定键的值：相反，combiner 发出总结了通过输入值的输出值。Combiners 通常用于执行 map 端“预聚集”，这降低了 map 和 reduce 步骤间的网络通讯量。

2.2 Hadoop 体系

Hadoop 由 Hadoop MapReduce——用于大型集群的一种 MapReduce 实现，和 Hadoop 分布式文件系统（HDFS）——为 MapReduce 等批量工作负荷优化的一种文件系统。在大多数 Hadoop 工作中，HDFS 用于存储 map 的输入和 reduce 的输出。注意到 HDFS 不是用来存储中间结果（如 map 的输出）：这些被保存在每个节点的本地文件系统。

Hadoop 安装由一个单一 master 节点和许多 worker 节点组成。称为 JobTracker 的 master 负责接收客户端的工作，将这些工作划分为任务，并将这些任务分配给 worker 节点执行。每个 worker 运行一个 TaskTracker 进程来管理当前分配给该节点的任务的执行。每个 TaskTracker 对于执行任务有固定的 slots 数量（默认两个 maps 和两个 reduces）。每个 TaskTracker 和 JobTracker 间的心跳协议用来更新 JobTracker 运行任务状态的簿记，并推动新任务的调度：如果 JobTracker 识别自由的 TaskTracker slots，它将进一步调度 TaskTracker 上的任务。

2.3 Map 任务执行

每个 map 任务被分配输入文件的一部分，称为分块。默认的，一个分块包括一个单一 HDFS 块（默认 64MB），所以输入文件的大小决定 map 任务的数量。

一次 map 任务的执行分为两个阶段。

1. map 阶段从 HDFS 读取任务的分块，将其解析为记录（键/值对），并将 map 函数应用于每个记录。
2. 当 map 函数被应用于每个输入记录后，commit 阶段向 TaskTracker 登记最终输出，然后通知 JobTracker 任务已执行完毕。

图 1 包含了必须由用户定义 map 函数实现的接口。在 map 函数被应用到每个记录的分裂后，close 方法被调用。

map 方法的第三个参数指定一个 OutputCollector 实例，它积累由 map 函数产生的输出记录。map 的输出被 reduce 函数消耗，所以 OutputCollector 必须格式化存储 map 输出以便 reducer 更易消耗。中间键通过应用分区函数被分配到 reducers，因此 OutputCollector 应用该函数于 map 函数产生的每个键，并存储每个记录和分区编号到内存缓冲区。OutputCollector 负责当缓冲区满时写入到硬盘。

内存缓冲区的溢出涉及到首先按缓冲区记录的分区编号排序，然后按键排序。缓冲区内容作为索引文件和数据文件被写入本地文件系统（图 2）。索引文件指出数据文件中每个分区的偏移。数据文件只包含记录，其按每个分区段的键排序。

在 map 任务将 map 函数应用于每个输入记录后，它进入 commit 阶段。为了生成任务的最终输出，内存缓冲区被刷新到硬盘，并且所有在 map 步骤生成的溢出文件被合并为单一数据和索引文件。这些输出文件在任务完成前向 TaskTracker 登记。TaskTracker 将会在 reduce 任务请求服务时去读这些文件。

```
public interface Mapper<K1, V1, K2, V2> {
  void map(K1 key, V1 value,
           OutputCollector<K2, V2> output);

  void close();
}
```

Figure 1: Map function interface.

```
public interface Reducer<K2, V2, K3, V3> {
  void reduce(K2 key, Iterator<V2> values,
              OutputCollector<K3, V3> output);

  void close();
}
```

Figure 3: Reduce function interface.

2.4 Reduce 任务执行

一次 reduce 任务的执行分为三个阶段。

1. shuffle 阶段获取 reduce 任务的输入数据。每个 reduce 任务分配一个由 map 产生的键范围的分区，所以 reduce 任务必须获取每个 map 任务输出的分区内容。
2. sort 阶段将拥有相同键的记录分组。
3. reduce 阶段将用户定义 reduce 函数应用到每个键和相应值的列表。

在 shuffle 阶段，reduce 任务通过每次向可配置数量的 TaskTrackers（默认 5）发送 HTTP 请求来获取每个 map 任务的数据。JobTracker 将主机 map 输出的每个 TaskTracker 位置传递到执行 reduce 任务的 TaskTracker。在传统的批量 Hadoop 中，reduce 任务直到 map 执行完毕并将其最终输出结果写到硬盘后才能获取 map 任务的输出。

在接收到所有 map 输出的分区后，reduce 任务进入 sort 阶段。每个分区的 map 输出已按照键的大小排序。reduce 任务将这些运行合并在一起来产生一单一按键排序的运行。然后任务进入 reduce 阶段，它调用用户定义的按不同键排序的 reduce 函数，传递给相关值的列。reduce 函数的输出被写入 HDFS 上一临时位置。在 reduce 函数被应用到 reduce 任务分区的每个键后，任务的 HDFS 输出文件自动地从其临时位置重命名到最终位置。

在该设计中，map 和 reduce 任务的输出在其被消耗前被写入硬盘。对于 reduce 任务，由于其输出被写入 HDFS，这特别昂贵。默认的，这需要一同步写操作，其必须在不同节点上存储每个输出块的三分拷贝（确保容错）。

输出实体化简化了容错，因为其减少了单点故障后必须恢复一致性的状态数量。如果任何任务（map 或 reduce）失败，JobTracker 仅仅安排一项新任务来完成和失败任务相同的工作。由于任务除了最终答案外不再输出任何数据，恢复步骤就不再需要了。

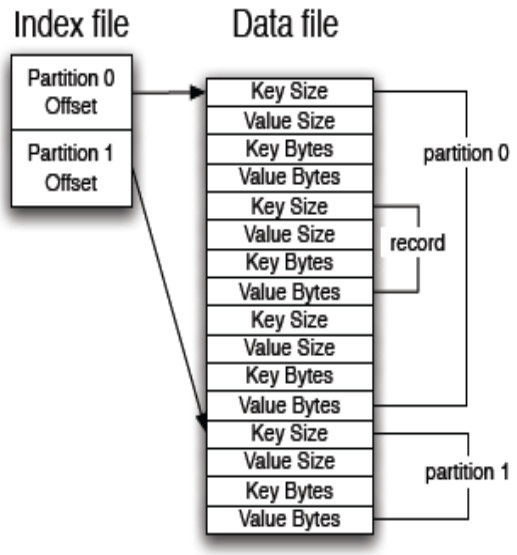

Figure 2: Map task index and data file format (2 partition/reduce case).

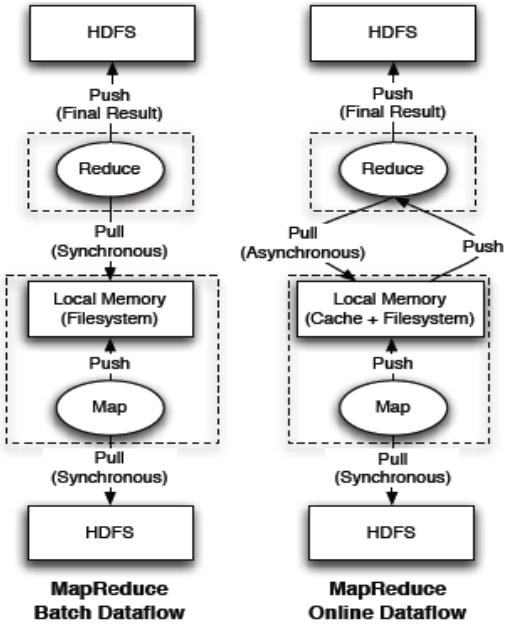

Figure 4: Hadoop dataflow for batch (left) and pipelined (right) processing of MapReduce computations.

3. 管道 MapReduce

在本节我们讨论为支持管道对 Hadoop 的扩展。我们在此关注批处理任务，将联机聚集和连续查询所需改变的讨论延迟到下一节。

图 4 描述了两种 MapReduce 实现的数据流。左侧数据流相当于普通 Hadoop 使用的输出实体化方法；右侧数据流允许管道。在本节其余部分，我们介绍管道 Hadoop 数据流的设计和实现。我们描述我们的设计如何支持容错（3.3 节），并讨论管道和任务调度间的交互（3.4 节），在 3.5 节我们比较普通 Hadoop 和我们的管道实现的性能。

3.1 工作内的管道

如 2.4 节所述，reduce 任务通常发出 HTTP 请求来从每个 TaskTracker 拉回其输出。这意味着 map 任务执行和 reduce 任务执行是分离的。为了支持管道，我们改进了 map 任务来代替当数据产生时将其推给 reducers。为了对其如何工作有一个直观感觉，我们首先描述一种

简单的管道设计，然后讨论为了达到良好性能所做的改变。

3.1.1 初步管道

在我们的初步实现中，我们希望直接连接 mappers 到 reducers 并在它们间用管道传送数据。当客户端向 Hadoop 提交一项新工作，JobTracker 将可用 TaskTracker slots 分配给该工作相关的 map 和 reduce 任务。为便于讨论，让我们假设有足够的自由 slots 来分配每个工作的所有任务。JobTracker 还将每个 map 任务的位置传达给每个 reduce 任务。我们改进 Hadoop 以便每个 reduce 任务在工作开始后联系每个 map 任务，并打开一个用于管道化 map 函数输出的 socket。当每个 map 输出记录被产生，mapper 决定记录应发送到哪个分区（reduce 任务），并通过适当 socket 直接发送之。

reduce 任务接受从每个 map 任务收到的管道数据并将其存储于内存缓冲区，在需要时将已排序的缓冲区内容写到硬盘。一旦 reduce 任务得知每个 map 任务均已完成，它执行已排序内容的最终合并，并按惯例应用用户定义 reduce 函数，将输出写入到 HDFS。

3.1.2 改进

虽然上述算法很简单，它也有几个实际问题。首先，它可能没有足够可用 slots 来调度新工作的每个任务。打开每个 map 和 reduce 任务间的 socket 也需要大量 TCP 连接。初步设计的一个简单调整解决了这两个问题：如果 reduce 还未被调度，任何产生该分区记录的 map 任务仅仅将其写入硬盘。一旦 reduce 任务被分配一个 slot，它可以像普通 Hadoop 一样从 map 任务获取记录。为了减少并发 TCP 连接数，每个 reducer 可以配置为从有限数量的 mappers 用管道传送数据；reducer 将按传统 Hadoop 方式从剩余 map 任务拉回数据。

我们最初的管道实现遇到了第二个问题：用户定义 map 函数由将输出记录写入管道 sockets 的相同线程所调用。这意味着如果一网络 I/O 堵塞（如因为 reducer 过度使用），则 mapper 无法做有用工作。管道延迟不应阻止 map 任务的进行——特别因为，一旦任务完成，它释放一个 TaskTracker slot 使其可用于其它用途。我们解决了这个问题，通过以单独的线程来运行 map 函数，将其输出存储到内存缓冲区，然后又另一线程定期发送缓冲区内容到管道 reducers。

3.1.3 map 输出的颗粒度

初步设计的另一问题是它急切地发送每个刚产生的记录，这阻止了 map 端 combiners 的使用。想象一下，一项工作其 reduce 键几乎没有不同值（如种类），并且 reduce 应用了聚集函数（如计数）。如 2.1 节的讨论，combiners 允许“map 端预聚集”：通过对 mapper 的每个不同键应用类 reduce 函数，网络流量通常可大大降低。通过在每个记录一产生便用管道传送，map 任务就没机会应用 combiner 函数。

一个相关问题是急切管道将一些排序工作从 mapper 移动到 reducer。回想在堵塞体系中，map 任务产生排序的溢出文件：所有 reduce 任务必须做的是将每个分区的预排序 map 输出合并在一起。在初步管道设计中，map 任务按产生顺序发送输出记录。因此 reducer 必须执行完整的外部排序。由于 map 任务数量通常远远超过 reduces 数量，在我们实验中移动更多工作到 reducer 增加了响应时间。

我们通过如 3.1.2 节所述修改内存缓冲区设计解决了这些问题。不用直接发送缓冲区内容给 reducers，我们等待缓冲区增长到阈值大小。然后 mapper 应用 combiner 函数，按分区和 reduce 键将输出排序，并将缓冲区写入硬盘，使用如 2.3 节所述的溢出文件格式。第二个线程监测溢出文件，并将其发送给管道 reducers。如果 reducers 能够赶上 map 任务并且网络不是瓶颈，溢出文件在产生后马上发送给 reducer（在此情况，溢出文件可能保存在 mapper 机器内核告诉缓冲中）。然而，如果 reducer 开始落后 mapper，未发送溢出文件数将会增加。在该情况中，mapper 定期对溢出文件应用 combiner 函数，将多个溢出文件合并成一个单一大型文件。这样具有将负载从 reducer 适应性移动到 mapper 的效果，反之亦然，这取决于哪个节点是当前瓶颈。

管道和适应性查询技术的联系在别处已被注意到（如[2]）。上述适应性调度相对简单，但我们相信，适应管道的反馈有可能显著提高 MapReduce 集群利用率。

3.2 工作间的管道

许多实际计算不能被表示为单一 MapReduce 工作，如 Pig 等高级语言的输出通常涉及多个工作。在传统 Hadoop 体系中，每个工作的输出在 reduce 步骤被写入 HDFS，然后在下个工作的 map 步骤直接从 HDFS 读回。实际上，JobTracker 直到 producer 工作完成后才能调度 consumer 工作，因为调度 map 任务需要直到 map 输入分块的 HDFS 块位置。

在我们的 Hadoop 改进版本中，一项工作的 reduce 任务可以选择性地将其输出直接用管道传送到下一工作的 map 任务，避免了 HDFS 中对于临时文件数的昂贵容错存储。不幸的，前一工作的 reduce 函数和后一工作的 map 函数的计算不能重叠：直到所有 map 任务完成后 reduce 步骤的最终结果才能产生，这阻止了有效地管道。然而，在下一节我们将描述联机聚集和连续查询管道如何能够发布工作间的真正管道“快照”输出。

3.3 容错

我们的管道 Hadoop 实现对于 map 和 reduce 任务故障都具有强壮性。为了从 map 任务故障中恢复，我们增加了一些 reduce 任务的簿记来记录 map 任务产生的每个管道溢出文件。为了简化容错，直到 JobTracker 通知 reducer map 任务已成功提交后，reducer 才把管道 map 任务的输出看做“试验”。reducer 可将相同未提交 mapper 产生的溢出文件合并在一起，但它直到被通知 map 任务已提交后才将其他 map 任务输出的溢出文件合并。这样，如果一 map 任务失败，每个 reduce 任务能忽略任何由故障 map 尝试所产生的临时溢出文件。JobTracker 会像普通 Hadoop 一样注意调度新 map 任务的尝试。

如果一 reduce 任务失败并且该任务的新副本已启动，所有发送给失败 reduce 尝试的输入数据必须再发送给新 reduce 实例。如果 map 任务以纯粹管道方式操作并且在输出发送给 reducer 后放弃之，这将十分困难。因此，map 任务保存其输出数据，并在以普通 Hadoop 方式提交前将完整输出文件写入硬盘。这允许 map 的输出在任何 reduce 任务失败的情况下重新生成。对于批作业，我们体系的主要优势是，reducer 不会因等待任务的完整输出被写入硬盘而堵塞。

我们的 map 任务故障恢复技术很简单，但设定了一个 reducer 合并溢出文件能力的较小限制。为了避免如此，我们设想引入“检查点”概念：当 map 任务运行时，它会定期通知 JobTracker 其在输入分裂中已达到 x 位移。JobTracker 将通知任何连接的 reducers；然后在 x 位移前产生的 map 任务输出像其他 map 任务输出一样被 reducers 合并。为了避免重复结果，

如果 map 任务失败新 map 任务尝试在 x 位移处重新读取其输入。该技术也具有减少 map 任务失败后多余工作数量的优势。

3.4 任务调度

Hadoop JobTracker 不得不被改造成了解工作的管道执行。内部工作管道由基本 Hadoop 版本部分处理，其联合调度的 map 和 reduce 任务是相同工作的一部分。然而，JobTracker 不清楚内部工作的依赖性。在普通 Hadoop 中，客户端需要提交一系列工作（可能由大量查询组成）必须按数据流依赖性的次序来做。也就是说，直到生产者完成执行后，消耗一个或多个其他工作输出的工作才能被提交。

Hadoop 输出一个接口到客户端，应用于提交作业。我们扩展了该接口来接受一列（管道）工作，列中的每个工作依赖于前一工作。客户端接口遍历此列并用其依赖的工作标识符来标注每个工作。JobTracker 调度者寻找该标注和联合调度工作的依赖性，给予“上游”工作比其提供的“下游”工作 slot 优先权。正如我们在第 7 节注意的，对于调度管道甚至我们计划未来研究的这些工作的 DAGs，都有很多有趣的选择。

3.5 性能评估

我们在 Amazon EC2 上使用 60 节点集群执行了一系列性能实验。一个节点执行 Hadoop 的 JobTracker 和 HDFS 的 NameNode，而余下的 59 节点担任运行 Tasktracker 的 slaves 和 HDFS 的 DataNodes。所有节点在拥有 1.7GB 内存和 2 个虚拟内核的“高 CPU 介质”EC2 实例上执行。每个虚拟内核相当于 2007 年 2.5Ghz Intel Xeon 处理器。

我们开始时测量单一 MapReduce 工作的性能，其不使用 combiner。排序通常作为基本 MapReduce 性能的基准，因为隐含排序在 reduce 阶段完成。我们将从维基百科提取的 5.5GB 文章排序；文章的每个单词被作为单独记录来分析。图 5 描述了在 EC2 集群上使用 128MB 大小的 HDFS 块（产生 40 个 map 任务）的排序性能。我们配置系统使用 59 个 reducers。在每个图中，我们给出 map 和 reduce 任务实现的 CDF。左图和右图分别面熟堵塞和管道的性能。

在该配置中管道比堵塞占优势，部分原因是它有更高的集群利用率：堵塞工作的 reduce 任务在试验前 192 秒是闲置的，然而在管道工作中，reducers 在 20 秒内就开始做有用工作。注意到在高利用集群中，增长的管道并行性不一定导致总吞吐量的改进。然而，这些结果表明管道可以大大减少个体工作的响应时间，这通常是重要的（如快速执行高优先级工作）。

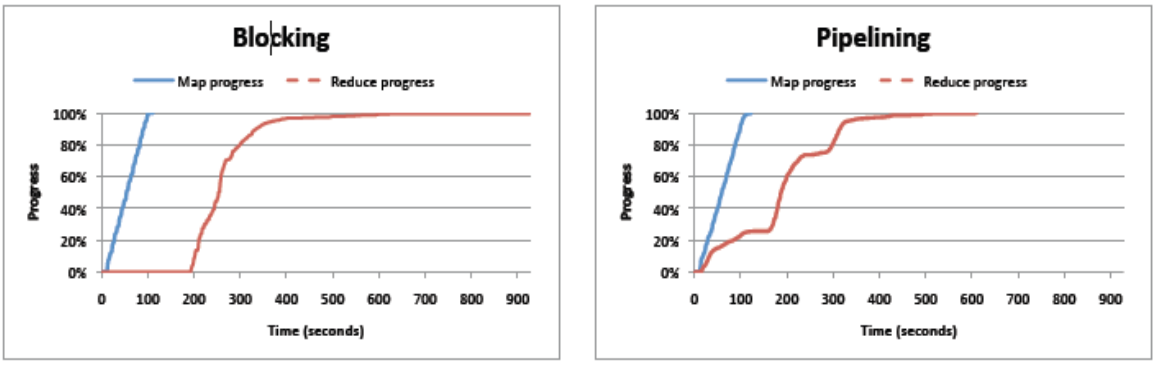

Figure 5: CDF of map and reduce task completion times for a sort job on 5.5GB of text extracted from Wikipedia. The total job runtimes were 927 seconds for blocking, and 610 seconds for pipelining.

4. 联机聚集

虽然 MapReduce 最初被设计为一个面向批处理的系统，现在往往用于交互式数据分析：用户提交一项工作来提取数据集中的信息，然后等待查看数据分析过程下一步骤前的结果。这种趋势将随着以 MapReduce 工作执行的高级语言（如 Hive、Pig、Sawzall）的发展而加速。

传统 MapReduce 实现提供了简单的交互式数据分析接口，因为知道工作执行完毕后他们才输出。然而，在许多情况下，交互式用户更希望有一个“临时应急”的近似值来代替需要更长时间计算的准确答案。在数据库文献中，联机聚集已被提出来解决该问题，但传统 MapReduce 实现的批处理特性使这些技术难以适用。在本节，我们展示我们如何扩展管道 Hadoop 实现来支持单一工作（4.1 节）和多个工作间（4.2 节）的联机聚集。我们说明联机聚集对于工作完成时间有最小影响，并通常可远远早于在工作执行完毕便产生一个准确的近似答案。

4.1 单一工作的联机聚集

在我们的管道 Hadoop 版本，每个 map 任务产生的数据记录在其产生不久便递增地发送给 reduce 任务。为了产生工作的最终输出，直到每个 map 任务的输出都产生后 reduce 函数才能被调用。通过对 reduce 任务迄今收到的数据简单应用 reduce 函数，我们能够支持联机聚集。我们将这种中间 reduce 操作的输出称为快照。

用户想知道快照的精确程度：就是说，快照和工作的最终输出有多接近。精确估计甚至对于简单 SQL 查询都是很难的问题，对于那些 map 和 reduce 函数是不透明用户定义代码的工作更是特别困难。因此，我们报告工作进度，而不是准确度：我们将其留给用户（或他们的 MapReduce 代码）来将进度和准确的正式概念相关联。我们举一个简单的进度指标如下。

在新数据到达每个 reducer 时，快照周期性计算。用户使用进度指标作为计算单位来指定快照计算的频率。例如，用户可以要求快照在输入显示为 25%、50%和 75%时被计算。用户还可以指定是否包含暂时（未完成）map 任务的数据。该选择不影响 3.3 节所述的容错设计。在当前原型，每个快照被保存在 HDFS 的一个目录中。目录的名字包含和此快照相关的进度值。每个 reduce 任务按不同速度独立运行；一旦 reduce 任务取得足够的进度，它将快照写入 HDFS 临时目录中，然后准确地将其重命名到适当的快照目录。

应用可以通过在固定位置轮询 HDFS 来消耗快照。当每个 reduce 任务将文件写入快照目录时应用就知道给定的快照已完成。准确重命名用来避免应用错误地读取不完整的快照文件（相同技术被普通 Hadoop 使用在写每个 reduce 任务的最终输出）。

注意到，如果没有足够的自由 slots 允许工作中的所有 reduce 任务被调度，快照对于仍然等待执行的 reduce 任务不可用。用户可检测到这种情况（如通过检查 HDFS 快照目录的预计文件数），因此不存在错误数据，但联机聚集的作用将被损害。在当前原型，我们手动配置集群来避免该情况。该系统还可被加强来完全避免此缺陷，比如通过等到足够 reduce slots 可用后才执行联机聚集工作。

4.1.1 进度指标

Hadoop 提供对监测任务执行进度的支持。当每个 map 任务执行时，它被分配一个在[0，1]范围内的进度值，基于其输入 map 任务被消耗的程度。我们重用此特性来确定当前 reduce

任务输入的多少，从而决定新的快照何时开始。

首先，我们改进了图 2 所述的溢出文件格式来包含 map 的当前进度值。当溢出文件的一个分区发送给 reducer，溢出文件的进度值也包括在内。为了计算快照的进度值，我们采用和每个用来产生快照溢出文件有关的平均进度值。

注意到，map 任务可能没有用管道传送任何输出到 reduce 任务，或者由于 map 任务还没有被调度（没有足够自由 TaskTracker slots），或者由于 reduce 被配置为只能用管道同时传送最多 k 个 map 任务的数据。为了说明这点，我们需要测量进度指标来反映用管道传送数据给 reduce 任务的 map 任务分区：如果一个 reducer 连接上工作总 map 任务数的 1/n，我们就用 n 来划分平均进度值。

该进度指标可以很容易变得更精确：例如，一改进指标可能包含每个 map 任务的选择度（|输出|/|输入|）、map 任务输出的统计分类和每个 map 任务 combine 函数的效果。虽然我们找到了满足如下所述的实验的简单进度指标，这显然代表了以后研究的机会。

4.1.2 评估

我们发现我们的联机聚集实现能迅速产生准确估计，并且对工作完成时间只产生相对少的开销。我们实验的准确指标是有因果关系的——我们注意到快照的前 K 个单词就是最终结果的前 K 个单词。

图 6 包含两个图形，其报告了在 Gutenberg 项目的 550MB 数据集中搜索前 5 个单词的工作结果。正确的前 5 个答案在总运行时间的 21%内可用。用来加载数据的 128MB 块大小在此工作的长期执行中存在差异。该文件被分为 5 块，其中之一（尾部）是小块。被分配到该小块的任务几乎立即完成。左图的堵塞工作显示了这个效果，因为 reduce 任务很早就收到该快速 map 的输出。其余三个 map 任务几乎同时完成，一 map 稍微落后。右图描述了管道的联机聚集。尽管联机聚集导致了性能损失，但其相对温和。此外，准确的近似答案在每个工作的最终完成前就被计算。

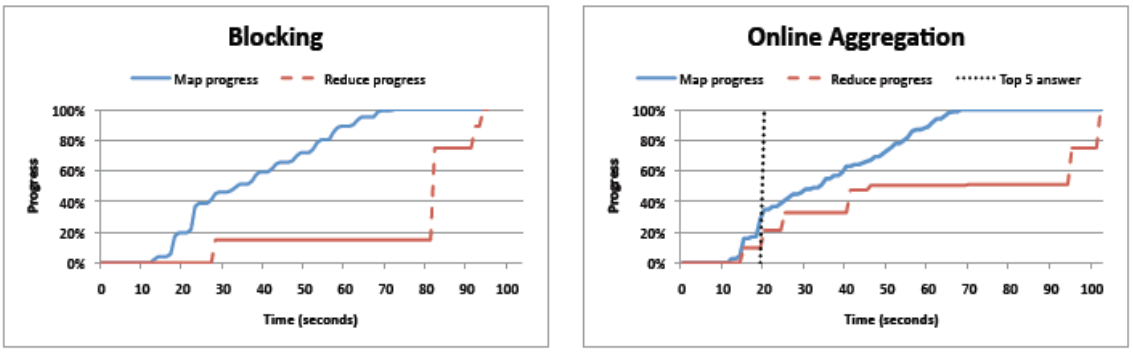

Figure 6: Single job that finds the top 5 words in the 550MB Gutenberg data set. In the graph on the right, the vertical dashed line indicates when online aggregation produces an accurate result.

4.2 多工作的联机聚集

联机聚集特别适用于由多 MapReduce 工作组成的长时间分析任务。如 3.2 节所述，我们的 Hadoop 版本允许 reduce 任务的输出直接发送给 map 任务。该特性可用来支持一系列工作的联机聚集。

假设 j_1 和 j_2 是两个 MapReduce 工作，并且 j_2 消耗 j_1 的输出。当 j_1 的 reducer 计算出快照

来完成联机聚集时，快照被写入 HDFS，并直接发送给 j_2 的 map 任务。然后 j_2 的 map 和 reduce 步骤如常计算，来产生 j_2 输出的快照。该过程也可以连续支持任意长系列工作的联机聚集。

不幸的，中间工作联机聚集有一些缺点。首先，reduce 函数的输出不“单调”：前 50% 输入数据的 reduce 函数的输出可能和前 25%的 reduce 函数输出没有明显关系。因此作为由 j_1 产生的新快照，j_2 必须使用该新快照重头开始重算。正如工作间的管道（3.2 节），这可能是声明为分布或代数聚集的 reduce 函数的最优化。

为了使工作间联机聚集能够容错，我们必须考虑三种情况。如 3.3 节所述在 j_1 上失败的任务恢复。如果 j_2 任务失败，系统仅仅重启失败的任务。由于由 j_1 产生的快照是单调的，由 j_2 上重启任务接收的下一快照将有更高的进度值。为了处理 j_1 的故障，j_2 任务隐藏最新的由 j_1 接收的快照，并当其收到更高进度的快照时取代它。如果两个工作的任务都失败了，j_2 的新任务恢复 j_1 上存在 HDFS 重的最新快照，并等待更高进度的快照。

图 7 描述了批量模式（左）的工作间数据流和支持联机聚集（右）的数据流。批量模式推动工作读取其他 HDFS 上工作的最终输出。

除了读取 HDFS 的快照，我们支持用管道直接传送快照到所需的工作。这是通过一个由每个 reduce 任务输出的异步请求调用接口来支持的，通过它其它工作的 map 任务可以请求快照甚至最终输出。除非在工作配置中另外指定，最终输出为了容错会同时写入 HDFS。

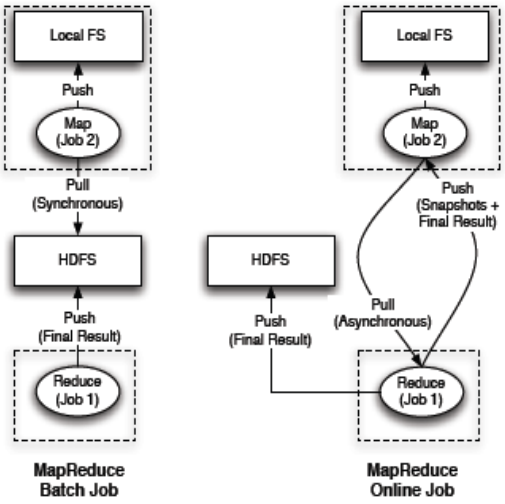

Figure 7: Hadoop dataflow for batch (left) and snapshot (right) processing of MapReduce jobs.

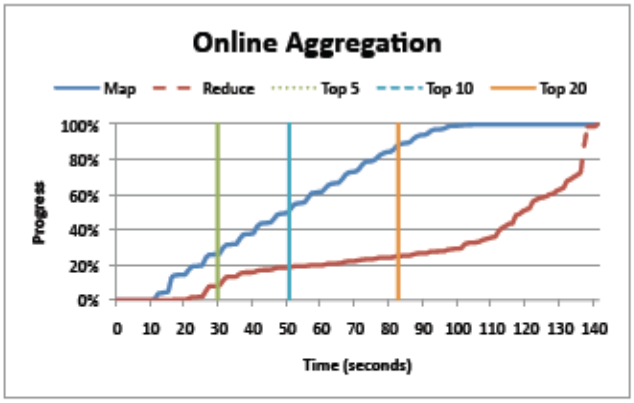

Figure 9: Multi-job Top-100 query over 5.5GB of Wikipedia article text. The vertical lines describe the increasing accuracy of the intermediate results produced by online aggregation.

4.2.1 评估

为了评估联机聚集的工作间数据流，我们使用两个 MapReduce 工作来写前 K 个查询并且在 5.5GB 维基百科文章中执行之。第一个工作完成了对每个文章中单词的计数。第一个工作的 reducer 将输出在其分区中观测到的前 K 个单词列。该输出的键是单词，值是单词计数。随后工作的每个 map 任务被分配一个单一 reduce 任务的输出。map 函数反转了键-值顺序，并将结果（按降序计数排列）发送给单一 reduce 任务。单一 reduce 任务合并每个 mapper 的排序列并返回前 K 个单词。

图 8 报告了在 5.5GB 维基百科文章上的前 100 个查询的结果。左图代表了 reduce 任务进度的空闲周期所指出的堵塞情况。首个 map 任务在进入工作的 100 秒时完成，这就是 reduce 任务开始取得进展之处。右图展示了一个更均衡的负载。接收 mapper 输出的 reduce

任务几乎紧跟着工作的执行，有助于工作提前完成。

图 9 包含在相同维基百科数据集上用联机聚集执行工作的前 100 个查询的图表。虽然该工作的最终结果直到将近结束才出现，但我们在图中所示的时间能观察到前 5、10 和 20 的值。该工作尽可能频繁地完成联机聚集，这导致了减少最终工作完成时间的性能。

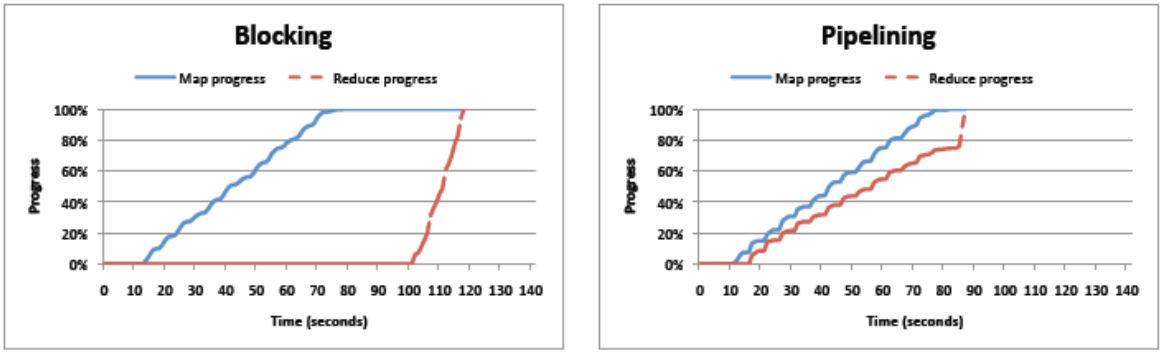

Figure 8: Multi-job Top-100 query over 5.5GB of Wikipedia article text.

5. 连续查询

MapReduce 通常用来分析持续到达的数据流，比如 URL 访问日志和系统控制日志。由于 MapReduce 的传统限制。大批量如此执行只能提供周期性的活动视图。对于理想上应该近实时运行的数据分析，这带来了显著的等待时间。它也可能是无效的：每个新 MapReduce 还没有进入最后分析的计算状态，因此该状态必须重头重新计算。程序员可以手动保存每个工作的状态，然后重载它来操作下一分析，但这是劳动密集的。

我们的 Hadoop 管道版本允许可选择体系：持续运行的 MapReduce 工作在其变为可用时接受新数据并直接分析之。这允许近实时的数据流分析，从而使 MapReduce 编程模式能适用于环境监测和实时欺诈监测等领域。

在本节，我们描述 HOP 如何支持 MapReduce 工作，以及我们如何使用该特性来实现一个基本集群监测工具。

5.1 连续 MapReduce 工作

连续 MapReduce 工作的基础实现很容易用管道实施。实现连续 map 不需要改变：map 输出在产生后迅速传递给适当 reduce 任务。我们增加了一可选“刷新”API，其允许 map 函数将其当前输出推到 reduce 任务。当 reduce 任务无法接受这样的数据时，mapper 架构将其存在本地并随后发送。随着 reducers 的正确调度，该 API 允许 map 任务确保输出记录及时被送到适当 reducer。

为了支持连续 reduce 任务，用户定义 reduce 函数必须被 reducer 上可用的 map 输出周期性调用。对于 reduce 函数被调用的频率，应用有不同的要求：可能的选择包括基于挂钟时间、逻辑时间（如 map 任务输出的字段值）和传给 reducer 的输入行数等的周期。如同我们的联机聚集实现，reduce 函数的输出被写入 HDFS。然而，其他选择是可能的：我们的原型系统监测应用（如下所述）一旦发现异常情况就同多 email 发送警报。

在我们当前的实现，map 和 reduce 任务的数量是固定的，并且必须由用户配置。这显然是有问题的：手动配置容易出错，并且流处理应用展示了“突发”流量模式，其最大负荷远远超过平均负荷。今后，为了响应负载变化，我们计划增加对 map 和 reduce 任务的弹性

按比例增加/缩减的支持。

5.1.1 容错

在 Hadoop 所使用的检查点/重启容错模式中，mappers 保留其输出到工作结束以便从 reducer 故障快速恢复。就连续查询而言，这是不可行的，因为 mapper 历史原则上是无限的。然而，许多连续 reduce 函数（如 30 秒移动平均数）只取决于 map 流历史的后缀。这种常见情况很容易被支持，通过扩展 JobTracker 接口来俘获 reducer 消耗的滚动概念。map 端溢出文件以其唯一 ID 保存在一个环形缓冲区中。当 reducer 想 HDFS 提交输出时，它通知 JobTracker 其不再需要的 map 运行输出记录，通过溢出文件 IDs 和这些文件内的偏移来识别此运行。然后 JobTracker 通知 mappers 删除相应数据。

原则上，复杂 reducers 可能取决于很长（或无限）map 记录历史来准确重建其内部状态。在该情况下，删除 map 端环形缓冲区的溢出文件将导致故障后潜在的错误恢复。这种情况使用 HDFS 的 reducer 检查点内部状态来处理，同时 mapper 的标记和被检查的内部状态相抵消。MapReduce 架构可以用 APIs 来扩展，以帮之状态序列化和偏移管理，但它仍提出了用户编程负担来正确识别敏感的内部状态。这种负担可通过更多重量级容错进程对技术来避免，但这些十分复杂并且使用大量资源。在我们至今的工作，我们集中研究 reducer 可以从 mappers 的合理规模历史中的恢复情况，支持用于 Hadoop 的简单容错方法的有限扩展。

5.2 原型监测系统

我们的监测系统由运行在监测机器上的代理和兴趣记录查询（如平均负载，每秒 I/O 操作等）组成。每个代理连续 map 任务所实现：不是从 HDFS 中读取，map 任务从各种连续本地系统数据流（如/proc）读取。

每个代理提出由连续 reduce 任务实现的 aggregator 统计。aggregator 记录了本地代理统计如何随时间进展（如通过计算窗口平均值），并比较代理间的统计来检测异常行为。每个 aggregator 仅仅监测向它报告的代理，但也可能报告统计摘要给另一“上游”aggregator。例如，系统可能配置为每个机架有一个 aggregator，然后 aggregator 得第二级比较机架间的统计来分析数据中心范围的行为。

5.3 评估

为了验证我们的原型系统监测工具，我们构造了一种情况，当工作执行时 MapReduce 集群之一开始超负荷。我们的目标是检测我们的监测系统多快能检测到该行为。其基本机制类似于某一作者在网络搜索公司实现的警报系统。

我们使用了一简单负载指标（一种 CPU 利用率、分页和交换活动的线性组合）。连续 reduce 函数维持该指标样本窗口：每隔一段时间，它比较每个主机 20 秒移动平均负载指标和集群中除此主机外所有主机 120 秒移动平均值。如果给定主机的负载指标超过总体平均值的两倍标准差，它被任务是异常值和发出的初步警报。为了抑制“突发”负载情况的误报，直到一个时间窗口内接收到 10 个初步警报我们才发出一个警报。

我们在由 7 个“大型”节点（大型节点被选中是因为 EC2 将整个物理主机分配给它们）组成的 EC2 集群上部署该系统。我们在 5.5GB 维基百科数据集上运行单词计数工作，使用 5

个 map 任务和 2 个 reduce 任务（每个主机 1 个）。在工作大约运行 10 秒后，我们选中一个运行任务的节点，并启动一个导致超负荷的程序。

我们在图 10 报告了检测延迟。在超负荷主机上，我们运行 vmstat 来监测随时间交换页面数。垂直线表明监测工具发出警报（非初步）的时间。超负荷主机很快被检测到——比普通 Hadoop 上用了检测滞后任务的 5 秒 TaskTracker-JobTracker 心跳周期快很多。我们设想使用这些警报来作为 MapReduce 工作中滞后者的早期检测。这种情况是 HOP 通过运行辅助连续 MapReduce 检测查询来对 MapReduce 工作的调度决策。相比于带外监测工具，这种对于反射监测重用 MapReduce 基础设施的机制有益于软件维护和系统管理。

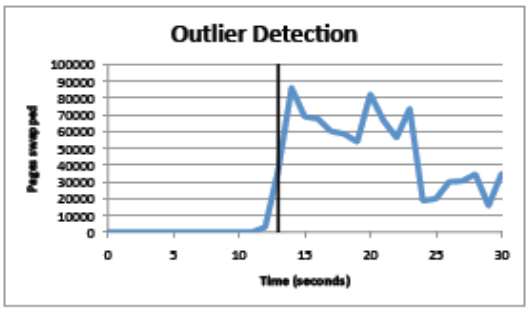

Figure 10: Pages swapped in and out over time on the thrashing host. The vertical line indicates the time at which the alert was sent by the monitoring system.

6. 相关工作

本文的工作与并行数据流架构、联机聚集和连续查询处理的文献有关。

6.1 并行数据流

Dean 和 Ghemwat 关于 Google 的 MapReduce 论文已经成为一个标准参考，并成为了开源 Hadoop 实现的基础。如第 1 节所述，Google MapReduce 设计将大型集群作为目标，其工作故障和怠工的概率非常高。这导致了它们对于容错很讲究的检查点/重启方法和（必然结果）它们对管道的缺乏。我们的研究扩展了 Google 的设计来适应管道，并对其核心编程模式和容错机制没有很大的修改。

Dryad 是一种经常和 MapReduce 相比较的并行数据编程模式，其支持更普遍的无环数据流图模型。和 MapReduce 一样 Dryad 默认在数据流节点间进行硬盘实体化步骤，破坏了管道。论文描述的 Dryad 支持可选地将特殊异步多阶段“封装”到单一进程中，使其可以使用管道。但这需要一个更复杂的编程接口。Dryad 论文明确提到，该系统以批处理为目标，并不会出现连续查询这样的情况。

有人指出，并行数据库系统早已提供分区数据流架构，并且最近的商业数据库已开始在这些架构的顶层提供 MapReduce 编程模式。大多数并行数据库系统可以提供和我们研究类似的管道执行，但它们使用更紧密耦合的迭代和交换模式来通过队列保持生产者和消费者速度匹配。将每个数据流阶段工作传播到集群所有节点中。这提供了少于 Hadoop 的 MapReduce 模式调度灵活性，并通常不提供中间查询工作的容错。Aster 数据广告支持中间 查询容错，尽管它们报告在 MapReduce 是实现的技术没有解释容错是如何实现的。

Logothetis 和 Yocum 描述了称为 Mortar 的连续查询系统 MapReduce 接口，在某些方面和我们的研究很像。和 HOP 一样，它们的 mappers 通过管道方式将数据推到 reducers。它们关注高效流处理的特殊问题，包括通过特殊 reduce APIs 聚集重叠窗口的最低限度。它们不是建立在 Hadoop 上，并能准确回避容错问题。

Hadoop 流是一个接口，允许 Map 和 Reduce 函数被表示为 UNIX 命令行。它不会以管道方式通过 Map 和 Reduce 阶段流出数据。

6.2 联机聚集

联机聚集最初在涉及“分组语句”聚集的简单单表 SQL 查询情况下被提出，非常类似于 MapReduce。最初的重点不仅在于这些 SQL 查询的“早期返回”，还在于基于随机采样的统计估计值和置信区间指标的最终结果。这些统计事项不能概括任意的 MapReduce 工作，尽管我们的架构可以支持那些已被开发的。随后联机聚集被扩展来处理连接查询（通过 Ripple Join 方法），并且 CONTROL 项目概括了联机查询处理的概念来提供数据清理、数据挖掘和数据可视化任务的交互性。该研究以单处理器系统为目标。Luo 等人开发了一种分区并行 Ripple Join 变种，没有近似答案的统计保障。

近年来，这个话题出现了新兴趣，开始于 Jermaine 等人关于 DBO 系统的研究。这种努力包括更多硬盘意识联机连接算法，以及维持随机文件来删除扫描中的任何统计偏移可能性。Wu 等人描述了在分布式哈希表环境中的对等网络联机聚集系统。对于联机聚集，MapReduce 的开放编程性和容错在之前的研究中海没有显著解决。

6.3 连续查询

在过去十年，在数据库研究界关于数据流连续查询的话题有大量研究，包括 Borealis、STREAM 和 Telegraph 等系统。其中，Borealis 和 Telegraph 研究了机器间的容错和负载均衡。在 Borealis 中是用管道数据流完成的，但没有分区并行：管道的每个阶段（“操作”）在广域不同机器中连续运行，并且容错处理整个操作的故障。SBON 是一个可与 Borealis 结合的覆盖网络，负责处理这些广域管道数据流的“操作配置”优化。

Telegraph 的 FLuX 操作是唯一对我们知识的研究，其解决了以 HOP 方式管道化和分区的中间数据流容错。FLuX（“容错、负载均衡交换”）是封装了比如 map 和 reduce 阶段间混排动作的数据流操作。它提供了负载均衡接口，在处理调度方针和更改路由策略时可以移动节点间的操作状态（如 reducer 状态）。对于容错，FLuX 开发了基于处理对的解决方法，其冗余工作确保操作状态在多节点上保持活跃。这消除了如第 5 节所述的连续查询排序程序的负担。另一方面，相比我们的管道 Google 检查点/重启容错改进模式，FLuX 协议更加复杂和消耗资源。

7. 结论和未来研究

MapReduce 已被证明是一种对于大规模并行编程的流行模式。我们的 Hadoop 联机原型扩展了该模型来适应管道行为，并保持了简单编程模式和完整 MapReduce 架构的容错。这提供了重要的新功能，包含通过联机聚集在长期工作上的“早期返回”，和流数据的连续查询。我们还证明了批处理的优点：通过工作内和工作间的管道，HOP 可缩短工作完成时间。

考虑未来研究时，调度是一个直接出现的话题。初步 Hadoop 已有机器和时间上调度皮任务的自由度，而且 HOP 中管道的引入仅仅增加了设计空间。首先，管道并行是提高 MapReduce 工作性能的新选择，但需要智能地结合任务内分区并行和“落后者”处理的投机冗余执行。其次，reduces 和 maps 间直接通信（绕过分布式文件系统）来调度深度管道的能力开启了不同工作间联合定位任务的新机会和挑战，来避免可能时的通信。

Olston 和 colleagues 已注意到不同于传统数据库的 MapReduce 系统需要“轻模式”最优化方法来聚集和反应运行完成信息。HOP 的联系查询工具对此确保强大的自省编程接口：完整 MapReduce 接口可被用来完成收集系统级近实时信息的性能监测任务，确保调度和数据流优化的紧密反馈循环。这是一个我们计划去探索的话题，包括监测工作中未完成工作最低接口的投机方法，以及如 Eddies 和 Flux 等早期研究的联系优化动态方法。

作为一较长期议程，我们希望使用 MapReduce 编程探索更多交互式应用。作为第一步，我们希望重访 CONTROL 工作的交互式数据处理，着眼于通过并行来改进扩展性。更进一步，我们正在考虑消除 MapReduce 数据流程序和如 SEDA 的轻量事件流程序模型间的差别。起源于 Hadoop 的 HOP 实现使其不可能完成原始性能如 SEDA 等事件。但如果有兴趣转变两个传统单独编程模型的观念，或许对于云计算和通用框架的编程将着眼于一个新的更通用目的架构。