

大规模批处理系统

有太多太多魔力 太少道理
太多太多游戏 只是为了好奇
还有什么值得 歇斯底里
对什么东西 死心塌地

一个一个偶像 都不外如此
沉迷过的偶像 一个个消失
谁曾伤天害理 谁又是上帝
我们在等待 什么奇迹

最后剩下自己 舍不得挑剔
最后对着自己 也不大看得起
谁给我全世界 我都会怀疑
心花怒放 却开到荼蘼

——王菲《开到荼蘼》

大数据计算中一类最常见的计算任务即为批处理，现代批处理计算系统的设计目标一般包括数据的高吞吐量、系统灵活水平扩展、能处理极大规模数据、系统具有极强的容错性、应用表达的便捷性和灵活性等，而非流式计算系统强调的处理的实时性等特性。现代大数据处理系统的发展趋势是特定应用领域设计专用系统，而非追求建立全能而各方面表现平庸的大一统系统，只有这样才能针对领域强调的目标做有针对性的优化与设计上的取舍，在重要特性上追求最优性能。

2004 年 Google 发表了 MapReduce 计算范型及其框架的论文，这是一种最典型的批处理计算范型，随着 Hadoop 的日渐流行，目前这种计算机制已经在很多领域获得了极为广泛的应用。尽管 2009 年左右以 Stone Braker 为首的并行数据库领域专家对 MapReduce 模型提出了质疑并引发了和 Jeff Dean 等人的技术争论，但是最终的结论是 MapReduce 和并行数据库系统（MPP）各有优劣且两者有一定互补和相互学习之处。与传统的 MPP 架构相比，MapReduce 更适合非结构化数据的 ETL 处理类操作，且其可扩展性及容错性明显占优，但是单机处理效率较低。尽管 MapReduce 提供了简洁的编程接口及完善的容错处理机制，使得大规模并发处理海量数据成为可能，但从发展趋势看，相对复杂的任务转换为 MapReduce 任务开发效率还是不够高，所以其有逐步被封装到下层趋势，即在上层系统提供更为简洁方便的应用开发接口，在底层由系统自动转换为大量 MapReduce 任务，这一点值得读者关注。

DAG 计算模型可以认为是对 MapReduce 计算机制的一种拓展（也可以将 MapReduce 看作 DAG 计算的一种特例），MapReduce 尽管提供了简洁的用户接口，应用开发者只须完成 Map 和 Reduce 函数的业务逻辑即可实现大规模数据批处理任务，但是其支持的运算符仅仅限定于 Map 和 Reduce 两类，所以在表达能力上不够丰富。另外，MapReduce 机制本质上是由 Map 和 Reduce 序列两阶段构成的，之所以说是序列的，是因为尽管 Map 阶段和 Reduce 阶段都支持大规模并发，但是在 Map 阶段有个任务同步过程，只有所有 Map 任务执行完成才能开始 Reduce 阶段的任务。从中也可以看出，MapReduce 对于子任务之间复杂的交互和依赖关系缺乏表达能力，DAG 计算模型对此做出了改进，可以表达复杂的并发任务间的依赖关系，有些系统还提供了更加丰富多样的运算符，这样使得整个系统的表达能力更强。当然，DAG 计算模型在带来更强表达能力和灵活性的同时，也需要付出相应的代价：其应用表达结构复杂，使用和学习成本相对较高，所以 DAG 计算系统在设计中关键的一环就是如何简化应用描述，以提高开发效率。

本章首先介绍 MapReduce 的计算模型及整体架构，之后对常见的 MapReduce 应用的计算模式进行了归纳整理，总结了最常见的计算模式并分析其各自特点。最后一部分主要以微软的 Dryad 系统为例，来介绍 DAG 的计算机制和整体框架设计。这里需要说明的是：Spark 本质上也是 DAG 批处理系统，鉴于其最能发挥特长的领域是迭代式机器学习，所以将其放入机器学习相关章节中进行介绍。

11.1 MapReduce 计算模型与架构

MapReduce 分布式计算框架最初是由 Google 公司于 2004 年提出的，这不仅仅是一种分布式计算模型，同时也是一整套构建在大规模普通商业 PC（成千台机器）之上的批处理计算框架，这个计算框架可以处理以 PB 计（1PB=1024TB）的数据，并提供了简易应用接口，将系统容错以及任务调度等设计分布式计算系统时需考虑的复杂实现很好地封装在内，使得应用开发者只需关注业务逻辑本身即可轻松完成相关任务。

11.1.1 计算模型

MapReduce 计算提供了简洁的编程接口，对于某个计算任务来说，其输入是 Key/Value 数据对，输出也以 Key/Value 数据对方式表示。对于应用开发者来说，只需要根据业务逻辑实现 Map 和 Reduce 两个接口函数内的具体操作内容，即可完成大规模数据的并行批处理任务。

Map 函数以 Key/Value 数据对作为输入，将输入数据经过业务逻辑计算产生若干仍旧以 Key/Value 形式表达的中间数据。MapReduce 计算框架会自动将中间结果中具有相同 Key 值的记录聚合在一起，并将数据传送给 Reduce 函数内定义好的处理逻辑作为其输入值。

Reduce 函数接收到 Map 阶段传过来的某个 Key 值及其对应的若干 Value 值等中间数据，函数逻辑对这个 Key 对应的 Value 内容进行处理，一般是对其进行累加、过滤、转换等操作，生成 Key/Value 形式的结果，这就是最终的业务计算结果。

为了更好地理解其计算模型，下面我们用 3 个具体的简单实例来对 MapReduce 计算模型进行讲解。

● 实例一：单词统计

假设给定 10 亿个互联网网页内容，如何统计每个单词的总共出现次数？这个任务看似简单，但在单机环境下快速完成还是需要实现技巧的，主要原因在于数据规模巨大。MapReduce 计算框架下实现这个功能很简单直观，只要完成 Map 和 Reduce 操作的业务逻辑即可。这个任务对应的 Map 操作和 Reduce 操作如下所示：

```
map(String key, String value):
    // key: 文档名
    // value: 文档内容
    for each word w in value:
        Emit Intermediate(w, "1");

reduce(String key, Iterator values):
    // key: 单词
    // values: 出现次数列表
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

1 累加values值

Map 操作的 Key 值是某个网页的网页 ID，对应的 value 是网页内容，即由相继出现在网页中的一序列单词构成，Map 操作对于每一个页面内容中依次出现的单词，产生中间数据<w,1>，代表了单词 w 在页面内容某个位置出现了一次；而 Reduce 操作的 Key 值为某个单词，对应的 Value 为出现次数列表，通过遍历相同 Key 的次数列表并累加其出现次数，即可获得某个单词在网页集合中总共出现的次数。可以看出，Map 阶段的 Key 值并未在后续操作中出现，因为其对于统计单词这个任务

而言不是必需的信息。

● 实例二：链接反转

在搜索引擎对网页进行链接分析时，有些算法需要获知网页链接的反转链接，比如 BadRank 反作弊算法即是如此。所谓“链接反转”，就是将链接指向倒置，比如网页 A 包含指向网页 B 和网页 C 的链接，链接反转将(A,B)和(A,C)两个链接反置为(B,A)和(C,A)两个反转链接。

在 MapReduce 计算模式下，链接反转的相应 Map 操作和 Reduce 操作如下：

```
map(String source_url, Iterator outlinks):
    // key:    网页url
    // value:  出链列表
    for each outlink o in outlinks:
        Emit Intermediate(o, source_url);

reduce(String target_url, Iterator source_urls):
    // key:    target网页url
    // values: source网页url列表
    list result = [];
    for each v in source_urls:
        Result.append(v);
    Emit(AsString(result));
```



Map 操作的 Key 是网页 URL，Value 是这个网页内包含的出链列表，其操作逻辑为将链接的指向者和被指向者反转输出。Reduce 操作的 Key 即为某个被指向网页的 URL，Value 内容为有链接指向 Key 这个 URL 的网页 URL 列表，其操作逻辑为将这个列表内容累加后输出。这样即可得到所有网页的入链列表。

● 实例三：页面点击统计

对于互联网网站来说，记录用户在网站的操作行为是后续数据挖掘和网站结构优化的必备步骤之一。通常的手段是通过 Log 形式将用户行为记载下来，Log 里每一行会记录相关信息，比如用户 ID、点击页面 URL、点击时间、用户所处地理位置等。图 11-1 是某个 Log 的片段信息。

Clicks(time	url	user_id	geo_info)
2012-06-05 19:34:25	http://www.baidu.com	1451	bj
2012-06-05 19:35:13	http://tieba.baidu.com	2365	sh
2012-06-05 19:36:45	http://mp3.baidu.com	1578	gz
2012-06-05 19:37:26	http://www.baidu.com	1987	cs
.....			

图 11-1 Log 片段信息

最常见的 Log 挖掘项目之一是统计页面点击数（PV），即网站各个页面在一定时间段内各自有多少访问量。在 MapReduce 计算框架下，相应的 Map 操作和 Reduce 操作如下：

```
map(String tuple_id, String tuple):
    Emit Intermediate(url, "1");

reduce(String url, Iterator list_tuples):
    int result = 0;
    for each v in list_tuples:
        result += ParseInt(v);
    Emit(AsString(result));
```

其逻辑非常简单，Map 操作将 Log 中每条记录的 URL 计数设定为 1，而 Reduce 操作则对相同 URL 的计数进行累加并输出。这样即可获得页面 PV 值。

11.1.2 系统架构

MapReduce 是一种分布式批处理计算模型，可以开发不同的具体系统来实现这种计算思路，事实上有很多支持 MapReduce 计算模型的框架，其中最有名的当属 Google 的 MapReduce 计算框架和 Hadoop 的 MapReduce 计算框架。

Google 的 MapReduce 计算框架架构如图 11-2 所示。

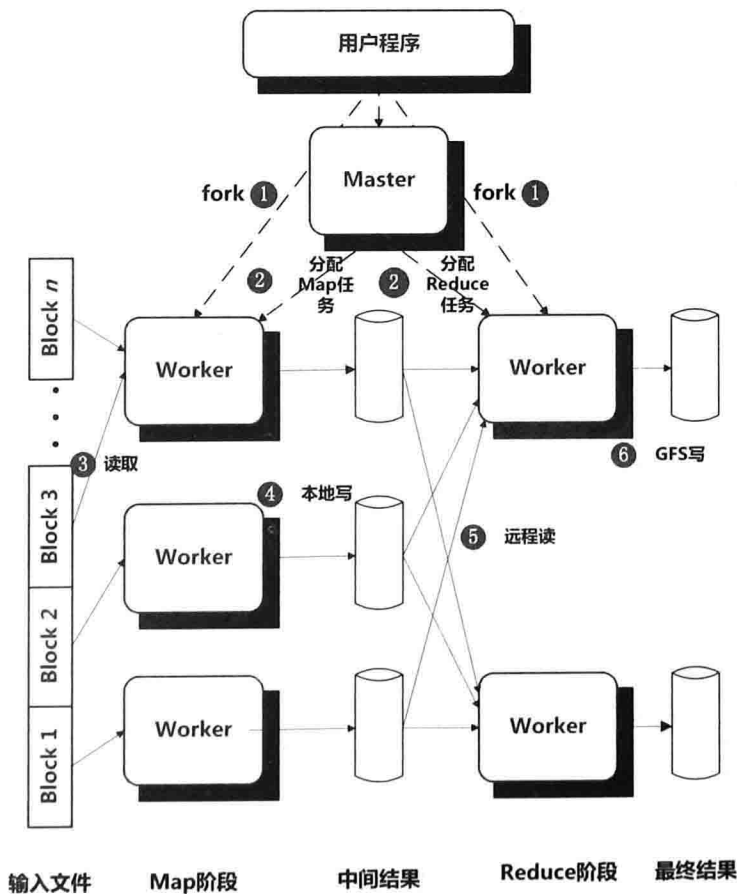


图 11-2 Google 的 MapReduce 计算框架架构

由图 11-2 可见，当用户程序执行 MapReduce 提供的调用函数时，其处理流程如下。

1. MapReduce 框架将应用的输入数据切分成 M 个数据块，典型的数据块大小为 64MB，然后可以启动位于集群中不同机器上的若干程序。

2. 这些程序中有一个全局唯一的主控 Master 程序以及若干工作程序（Worker），Master 负责为 Worker 分配具体的 Map 任务或者 Reduce 任务并做一些全局管理功能。整个应用有 M 个 Map 任务和 R 个 Reduce 任务，具体的 M 和 R 个数可以由应用开发者指定。Master 将任务分配给目前处于空闲状态的 Worker 程序。

3. 被分配到 Map 任务的 Worker 读取对应的数据块内容，从数据块中解析出一个个 Key/Value 记录数据并将其传给用户自定义的 Map 函数，Map 函数输出的中间结果 Key/Value 数据在内存中进行缓存。

4. 缓存的 Map 函数产生的中间结果周期性地被写入本地磁盘，每个 Map 函数的中间结果在写入磁盘前被分割函数（Partitioner）切割成 R 份， R 是 Reduce 的个数。这里的分割函数一般是用 Key 对 R 进行哈希取模，这样就将 Map 函数的中间数据分割成 R 份对应每个 Reduce 函数所需的数据分片临时文件。Map 函数完成对应数据块的处理后将其 R 个临时文件位置通知 Master，再由 Master 将其转交给 Reduce 任务的 Worker。

5. 当某个 Reduce 任务 Worker 接收到 Master 的通知时，其通过 RPC 远程调用将 Map 任务产生的 M 份属于自己的数据文件（即 Map 分割函数取模后与自己编号相同的那份分割数据文件）远程拉取（Pull）到本地。从这里可以看出，只有所有 Map 任务都完成时 Reduce 任务才能启动，也即 MapReduce 计算模型中在 Map 阶段有一个所有 Map 任务同步的过程，只有同步完成才能进入 Reduce 阶段。当所有中间数据都拉取成功，则 Reduce 任务根据中间数据的 Key 对所有记录进行排序，这样就可以将具有相同 Key 的记录顺序聚合在一起。这里需要强调的是：Reduce 任务从 Map 任务获取中间数据时采用拉取方式而非由 Map 任务将中间数据推送（Push）给 Reduce 任务，这样做的好处是可以支持细粒度容错。假设在计算过程中某个 Reduce 任务失效，那么对于 Pull 方式来说，只需要重新运行这个 Reduce 任务即可，无须重新执行全部所有的 Map 任务。而如果是 Push 方式，这种情形下只有所有 Map 任务都全部重新执行才行。因为 Push 是接收方被动接收数据的过程，而 Pull 则是接收方主动接收数据的过程。

6. Reduce 任务 Worker 遍历已经按照中间结果 Key 有序的数据，将同一个 Key 及其对应的多个 Value 传递给用户定义的 Reduce 函数，Reduce 函数执行业务逻辑后将结果追加到这个 Reduce 任务对应的结果文件末尾。

7. 当所有 Map 和 Reduce 任务都成功执行完成时，Master 便唤醒用户的应用程序，此时，

MapReduce 调用结束，进入用户代码执行空间。

为了优化执行效率，MapReduce 计算框架在 Map 阶段还可以执行可选的 Combiner 操作。所谓“Combiner 操作”，即是在 Map 阶段执行的将中间数据中具有相同 Key 的 Value 值合并的过程，其业务逻辑一般和 Reduce 阶段的逻辑是相似的，和 Reduce 的区别无非是其在 Map 任务本地产生的局部数据上操作，而非像 Reduce 任务一样是在全局数据上操作而已。这样做的好处是可以大大地减少中间数据数量，于是就减少了网络传输量，提高了系统效率。比如上面的单词计数例子中，如果 Map 阶段的中间结果数据中对单词进行了 Combiner 操作，则对某个单词来说网络只须传输一个 <key,value> 数值即可，而无须传输 Value 个 <key,1>，这明显大大减少了网络传输量。Combiner 一般也作为与 Map 和 Reduce 并列的用户自定义函数接口的方式存在。

Hadoop 的 MapReduce 运行机制基本上与 Google 的 MapReduce 机制类似，图 11-3 是其运行机制示意图。Mapper 任务调用用户自定义 Map 函数后对中间结果进行局部排序，然后运行 Combiner 对数据进行合并。Reducer 任务的 Shuffle 过程就是上述 MapReduce 运行流程中的步骤 5，不过 Hadoop 是采用 HTTPS 协议来进行数据传输的，并采用归并排序（Merge-Sort）来对中间结果 Key 进行排序，然后调用用户自定义 Reduce 函数进行业务逻辑处理并输出最终结果的。

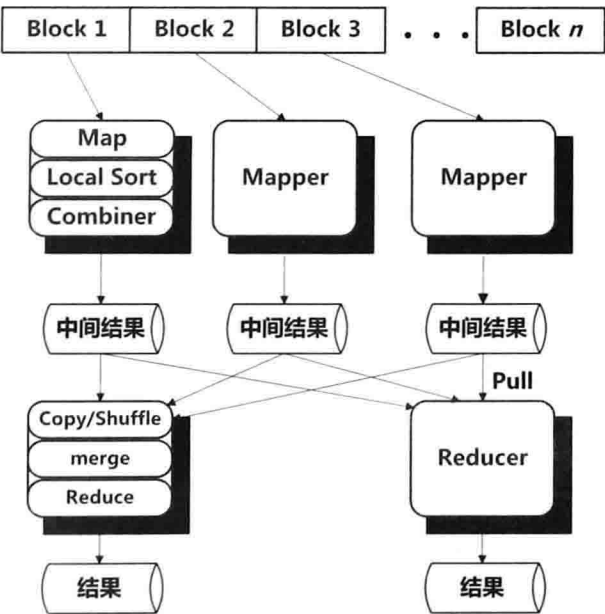


图 11-3 Hadoop 的 MapReduce 运行机制

Google 的 MapReduce 框架支持细粒度的容错机制。Master 周期性地 Ping 各个 Worker，如果在一定时间内 Worker 没有响应，则可以认为其已经发生故障。此时将由这个 Worker 已经完成的和正在进行的所有 Map 任务重新设置为 Idle 状态，这些任务将由其他 Worker 重新执行。至于为何将已经完成的任務也重新执行，是因为 Map 阶段将中间结果保存在执行 Map 任务的 Worker 机器本地磁盘上，Map 任务的 Worker 发生故障意味着机器不可用，所以无法获取中间结果，此时只能重新执行

来获得这部分中间数据。对于已经完成的 Reduce 任务来说,即使 Worker 发生故障也无须重新执行,因为其结果数据是保存在 GFS 中的,数据可用性已经由 GFS 获得了保证。但因为 Master 是单点的,所以如果 Master 失败,则整个 MapReduce 任务失败,应用可以通过反复提交来完成任务。

11.1.3 MapReduce 计算的特点及不足

MapReduce 计算模型和框架具有很多优点。首先,其具有极强的可扩展性,可以在数千台机器上并发执行。其次,其具有很好的容错性,即使集群机器发生故障,一般情况下也不会影响任务的正常执行。另外,其具有简单性,用户只需要完成 Map 和 Reduce 函数即可完成大规模数据的并行处理。

一般认为 MapReduce 的缺点包括:无高层抽象数据操作语言、数据无 Schema 及索引、单节点效率低下、任务流描述方法单一等。其中前两个缺点其实并不能认为是其真正的缺点,因为其设计初衷就是高吞吐、高容错的批处理系统,所以不包含这两个特性是很正常的。在目前的大数据处理架构范型下,试图构造满足所有类型应用各种不同特性要求的处理系统是不现实、也不明智的,最好的发展思路是对特定领域制定特定系统,这样可以在系统设计时充分强调领域特色的专用设计以使其效率达到最优,而不是去追求大而全但是各方面表现都平庸的系统。至于后两个缺点确实是客观存在的,对于任务流描述单一这个缺点,可以考虑将其拓展成更为通用的 DAG 计算模型来解决。

从上述 MapReduce 架构及其运行流程描述中也可以看出,为什么将其作为典型的批处理计算模型。MapReduce 运算机制的优势是数据的高吞吐量、支持海量数据处理的大规模并行处理、细粒度的容错,但是并不适合对时效性要求较高的应用场景,比如交互式查询或者流式计算,也不适合迭代运算类的机器学习及数据挖掘类应用,主要原因有以下两点。

第一,其 Map 和 Reduce 任务启动时间较长。因为对于批处理任务来说,其任务启动时间相对后续任务执行时间来说所占比例并不大,所以这不是个问题,但是对于时效性要求高的应用,其启动时间与任务处理时间相比就太高,明显很不合算。

第二,在一次应用任务执行过程中,MapReduce 计算模型存在多处的磁盘读/写及网络传输过程。比如初始的数据块读取、Map 任务的中间结果输出到本地磁盘、Shuffle 阶段网络传输、Reduce 阶段的磁盘读及 GFS 写入等。对于迭代类机器学习应用来说,往往需要同一个 MapReduce 任务反复迭代进行,此时磁盘读/写及网络传输开销需要反复进行多次,这便是导致其处理这种任务效率低下的重要原因。

11.2 MapReduce 计算模式

鉴于 MapReduce 已经在各种应用领域获得了广泛的使用,本节归纳了使用 MapReduce 机制来解

决批处理任务中若干最常见的计算模式，在实际应用中大部分 ETL 任务都可以归结为这些计算模式或者其变体。

11.2.1 求和模式（Summarization Pattern）

对于海量数据来说，通过对相似数据进行简单求和、统计计算或者相似内容归并是非常常见的应用场景，求和模式即描述这类应用场景及其对应的 MapReduce 解决方案，根据求和对象的类型，可以细分为数值求和以及记录求和两种情况。

1. 数值求和

如果计算对象是数值类型，那么对其进行统计计算是最常见的应用，统计计算包括简单计数、求最小值/最大值、求平均值/中位数等各种情况。11.1.1 节所举的例子中，实例一计算单词的出现频次，实例三计算网页 PV 数，就是简单计数的典型应用。对于此类应用，典型的 Hadoop MapReduce 解决方案的结构如图 11-4 所示。

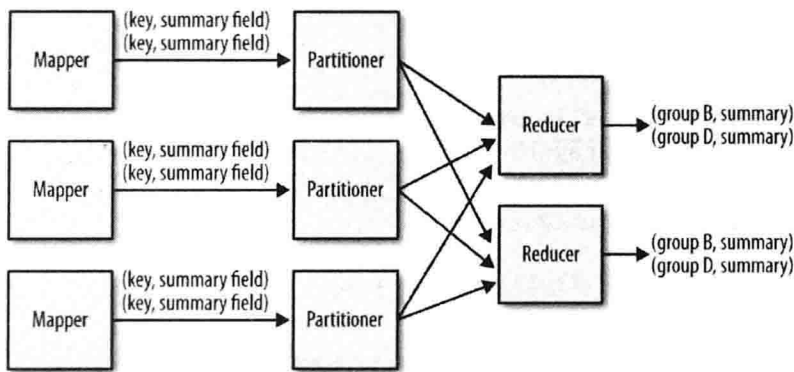


图 11-4 数值求和 MapReduce 结构

Mapper 以需要统计对象的 ID 作为 Key，其对应的数值作为 Value，比如单词计数中 Key 为单词本身，Value 为 1。在此种应用中如果使用 Combiner 会极大地减少 Shuffle（拖曳）阶段的网络传输量。另外，Partitioner 在这种应用中如何设计也很重要，一般的策略是对 Reducer 个数哈希取模，但是这可能会导致数据分布倾斜（Skewed），即有些 Reducer 需要处理大量的信息，如果能够合理选择 Partitioner 策略会优化此种情形。通过 Shuffle 阶段，MapReduce 将相同对象传递给同一个 Reducer，Reducer 则对相同对象的若干 Value 进行数学统计计算，得到最终结果。比如单词计数中这个数学计算就是简单求和，对于其他类型的应用，在这里也可以采取求均值或者中位数等各种统计操作。

2. 记录求和

对于非数值的情况，往往需要将非数值内容进行累加形成队列，一般应用中累加内容是对象的 ID。比如 11.1.1 节的实例二中对链接关系进行反转就是一种典型的记录求和计算模式，其累加的对象是网页 ID，即 URL。

对于此类应用，典型的 Hadoop MapReduce 解决方案的结构如图 11-5 所示。由图可以看出，其与数值求和流程基本类似，区别主要是在 Reducer 阶段采用累加对象 ID 形成信息队列。

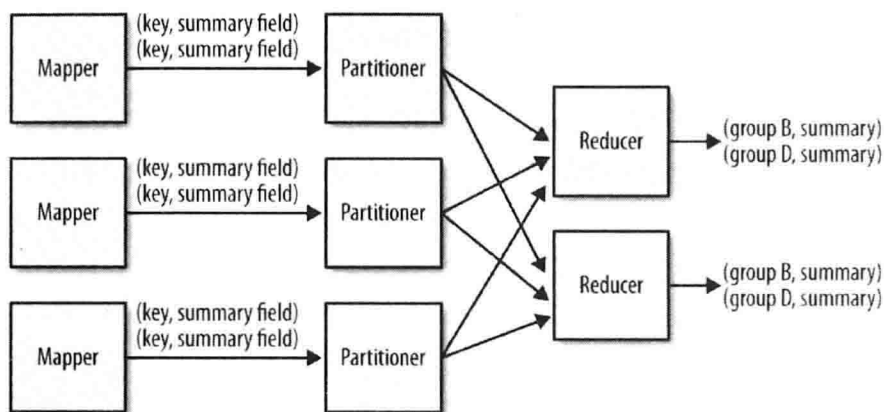


图 11-5 记录求和 MapReduce 结构

此类应用中最典型的的就是搜索引擎中建立倒排索引，其 MapReduce 算法如下所示。

```

map(String pageID, String pageContent):
    // key: 网页 ID
    // value: 网页内容
    for each distinct word w in pageContent:
        Emit Intermediate(w, pageID);
        ① 单词所在文档

reduce(String word, Iterator pageID_List):
    // key: 单词
    // values: 包含单词的网页 ID 列表
    list result = [];
    for each v in pageID_List:
        Result.append(v);
        ① 累计倒排文档列表
    Emit(AsString(result));
    
```

Map 函数中的 Key 为网页 ID，Value 为网页内容，Map 函数输出网页内不同单词及其对应的文档编号。通过 Shuffle 阶段，相同单词信息传输给同一个 Reducer，Reduce 任务累加同一个单词的不同文档 ID 列表并输出，这个列表就代表了包含某个单词的所有文档，如此即可建立倒排索引。当然这个例子是简化版本，真实情形要复杂些，但是基本思路是一致的。

11.2.2 过滤模式（Filtering Pattern）

数据过滤也是非常常见的应用场景，很多情形下，需要从海量数据中筛选出满足一定条件的数据子集，这就是典型的数据过滤应用场景。这种场景的一个特点是并不对数据进行任何转换，只是从大量数据中筛选。

1. 简单过滤

简单过滤即根据一定条件从海量数据中筛选出满足条件的记录。我们假设存在一个函数 f 对记

录内容进行判断，并以返回 True 或者 False 作为判断结果，这个函数即是记录满足条件与否的判断标准，不同应用只是这个函数内容不同而已，其整体处理逻辑是一致的。

对于此类应用，典型的 Hadoop MapReduce 解决方案的结构如图 11-6 所示。因为这类应用不需要对数据进行聚合等操作，所以无须 Reduce 阶段，是一个 Map-Only 类型的 MapReduce 方案。Mapper 从数据块中依次读入记录，并根据条件判断函数 f 判断该记录是否满足指定条件，如果满足则输出结果。

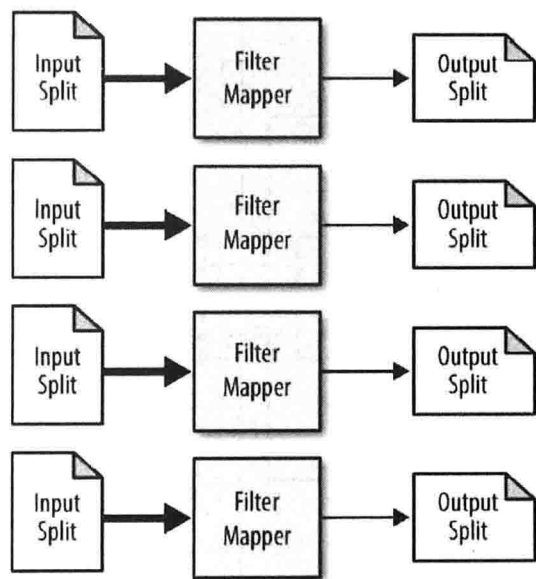


图 11-6 简单过滤 MapReduce 结构

这类应用适用场景非常广，比如对数据进行清理将无关数据清除、从大量数据中追踪感兴趣的记录、分布式 Grep 操作、记录随机抽样等。下面我们以分布式 Grep 作为例子，其 Map 函数如下：

```
map(Int lineNum, String lineContent):  
    // key: 行标号  
    // value: 行内容  
    If(lineContent.Match(RegexRules)): 1 判断条件  
        Emit(lineContent);
```

2. Top 10

从大量数据中，根据记录某个字段内容的大小取出其值最大的 k 个记录，这也是非常常见的数据过滤应用场景（不失一般性，这里以 $k=10$ 作为例子）。这和简单过滤的差异是：简单过滤的条件判断只涉及当前记录，而 Top k 计算模式则需要在记录之间进行比较，并获得全局最大的数据子集。

典型的例子比如搜索引擎统计当日热搜榜，即是从用户搜索日志中找出最常搜索的 Top k 个热门查询。对于热搜榜的应用，可以首先使用数值求和计算模式的 MapReduce 任务统计出当日搜索日志中每个查询的频次，再串接一个 Top 10 数据过滤计算模式的 MapReduce 任务即可得出所需的数据。

对于此类应用，典型的 Hadoop MapReduce 解决方案的结构如图 11-7 所示。其基本思路很简单：Mapper 首先统计出数据块内所有记录中某个字段满足 Top 10 条件的记录子集，不过这只是局部 Top 10 记录，然后通过 Reducer 对这些局部 Top 10 记录进一步筛选，获得最终的全局最大的 10 条记录。在这里 Mapper 和 Reducer 的处理逻辑是类似的，即找到数据集合中指定字段最大的若干记录，在实际使用中可以使用排序算法来实现，比如堆排序。

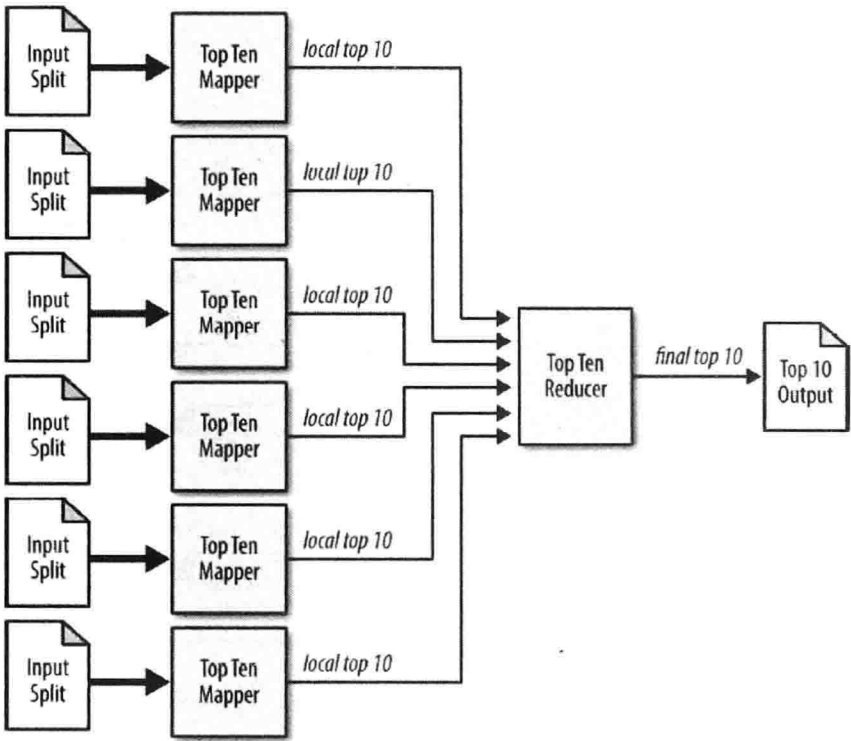


图 11-7 Top 10 的 MapReduce 结构

11.2.3 组织数据模式（Data Organization Pattern）

很多应用需要对数据进行整理工作，比如转换数据格式、对数据进行分组归类、对数据进行全局排序等，这是组织数据模式发挥作用的应用场景。

1. 数据分片

有些应用场景需要对数据记录进行分类，比如可以将所有记录按照日期进行分类，将同一天的数据放到一起以进一步做后续数据分析；再比如可以将相同地区的记录分类到一起等。因为 MapReduce 计算流程中天然具有 Partition 过程，所以对于这类应用，MapReduce 方案的工作重心在 Partition 策略设计上。

对于此类应用，典型的 Hadoop MapReduce 解决方案的结构如图 11-8 所示。一般情况下，Mapper 和 Reducer 非常简单，只需要将原始 KV 输入数据原样输出即可（即 Identity Mapper 和 Identity Reducer），其重点在 Partitioner 策略的设计，通过改变 Partition 策略来将相同标准的数据经过 Shuffle

过程放到一起，由同一个 Reducer 来输出，这样即可达到按需数据分片的目的。比如对于日期类型的数据，可以通过将同一天数据分配到一起实现。

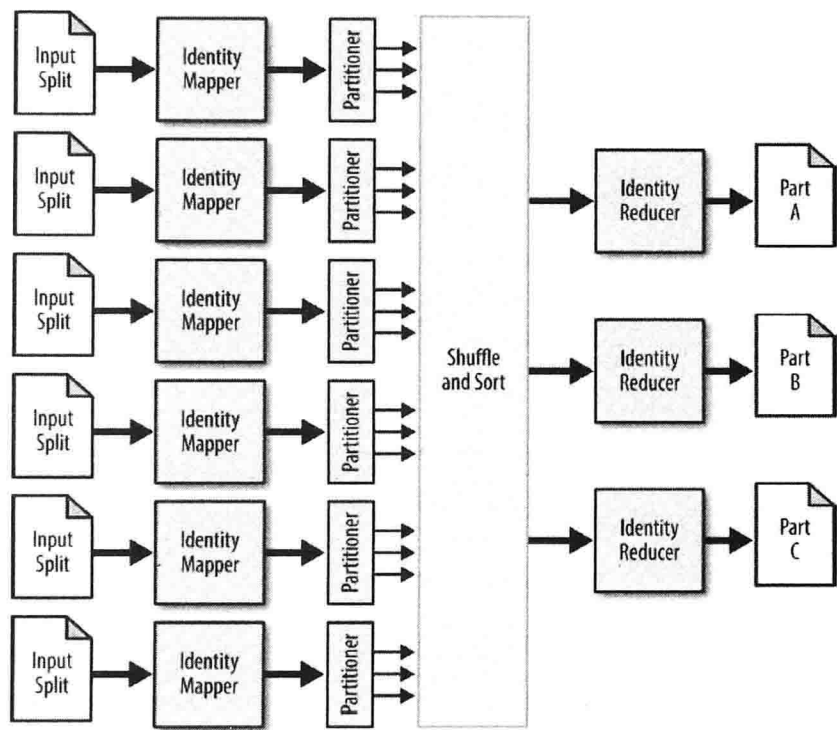


图 11-8 数据分片 MapReduce 结构

2. 全局排序

对于海量数据全局排序应用场景来说，可以充分利用 MapReduce 本身自带的排序特性来实现。从前面所述的 MapReduce 架构可知，在 Reduce 阶段需要首先将中间数据按照其 Key 大小进行排序，目的是将相同 Key 的记录聚合到一起，所以对于全局排序类应用可以直接利用这个内置排序过程。

Mapper 逻辑很简单，只需要将记录中要排序的字段作为 Key，记录内容作为 Value 输出即可。如果设定一个 Reducer，那么 Reduce 过程不需要做额外工作，只需以原样输出即可，因为 Reduce 过程已经对所有数据进行了全局排序。但是如果设定多个 Reducer，那么需要在 Partition 策略上做些研究，因为尽管每个 Reducer 负责的部分数据是有序的，但是多个 Reducer 产生了多份部分有序结果，仍然没有得到所需的全局排序结果。此时，可以通过 Partition 策略，在将数据分发到不同 Reducer 的时候，保证不同 Reducer 处理一个范围区间的记录，比如 Key 的范围是 1~10000，那么可以将 1~1000 的 Key 记录交给第 1 个 Reducer 处理，1001~2000 交给第 2 个 Reducer 处理……依次类推，这样将所有结果顺序拼接即可得到全局有序的记录。一种 Partition 策略列举如下。

```
Partition(key) {  
    range = (KEY_MAX - KEY_MIN) / Number_of_Reducers;
```

```
    reducer_no = (key - KEY_MIN) / range;
    return reducer_no;
}
```

11.2.4 Join 模式（Join Pattern）

两个数据集合进行 Join 操作也较常见，所谓“Join”，就是将两个不同数据集合内容根据相同外键进行信息融合的过程，图 11-9 和图 11-10 展示了论坛应用中用户信息表和帖子信息表根据用户 ID 进行 Join 的过程。

User ID	Reputation	Location
3	3738	New York, NY
4	12946	New York, NY
5	17556	San Diego, CA
9	3443	Oakland, CA

User ID	Post ID	Text
3	35314	Not sure why this is getting downvoted.
3	48002	Hehe, of course, it's all true!
5	44921	Please see my post below.
5	44920	Thank you very much for your reply.
8	48675	HTML is not a subset of XML!

图 11-9 用户信息表和帖子信息表

A.User ID	A.Reputation	A.Location	B.User ID	B.Post ID	B.Text
3	3738	New York, NY	3	35314	Not sure why this is getting downvoted.
3	3738	New York, NY	3	48002	Hehe, of course, it's all true!
5	17556	San Diego, CA	5	44921	Please see my post below.
5	17556	San Diego, CA	5	44920	Thank you very much for your reply.

图 11-10 Join 操作

常见的 Join 包括 Reduce-Side Join 和 Map-Side Join，下面我们分述之。

1. Reduce-Side Join

Reduce-Side Join 是解决数据集合 Join 操作的一种比较通用的方法，很多其他类型的 Join 操作也可以通过 Reduce-Side Join 来完成，所以其具有实现简单以及具备通用性的优点，但是缺点是因为其没有根据不同 Join 类型的特点做出特定优化，所以计算效率较低。

对于此类应用，典型的 Hadoop MapReduce 解决方案的结构如图 11-11 所示。这里需要注意的一点是：Reduce-Side Join 和一般 MapReduce 任务不同的地方在于其需要两个输入数据，而常见的 MapReduce 任务只需要一个输入数据。如何解决这个问题？

参照图 11-11，其运行流程如下。

首先，Mapper 将两个数据集合 A 和 B 的记录进行处理，抽取出需要 Join 的外键作为 Key，记录的其他内容作为 Value 输出，为了解决在 Reduce 阶段进行实际 Join 操作的时候判断数据来源的问题，

可以增加一个标志信息，表明这条记录属于数据集 *A* 还是属于数据集 *B*，实际实现时可将这个标志信息存储在 Value 中。

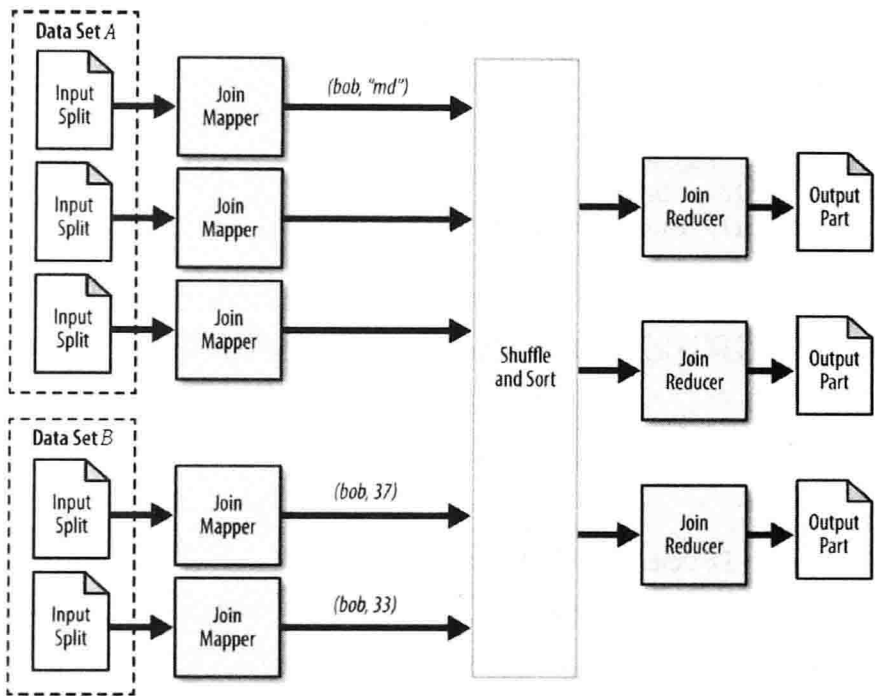


图 11-11 Reduce-Side Join 的 MapReduce 架构

然后，通过正常的 Partition 策略并经过 Shuffle 过程，两个数据集中具有相同外键的记录都被分配到同一个 Reducer，Reducer 根据外键排序后可以将同一个外键的所有记录聚合在一起。之后，Reducer 根据标识信息区分数据来源，并维护两个列表（或哈希表），分别存储来自于数据集 *A* 以及数据集 *B* 的记录内容，然后即可对数据进行 Join 操作并输出结果。


从上述计算流程可以看出，之所以将其称为“Reduce-Side”，是因为真正的 Join 操作是在 Reduce 阶段完成的。另外，从其流程也可以看出 Reduce-Side Join 具备很强的通用性，一般的 Join 操作都可以通过上述方式完成，但是所有的数据需要经过若干轮中间数据的磁盘读/写、Shuffle 阶段网络传输以及 Reduce 阶段的排序等耗时的操作，所以其计算效率比较低。

2. Map-Side Join

另外一类常见的 Join 操作是 Map-Side Join，其含义是：有些场景下，两个需要 Join 的数据集合 *L* 和 *R*，一个大一个小（假设 *L* 大 *R* 小），而且小的数据集合完全可以在内存中放入，此时，只需要采用一个 Map-Only MapReduce 任务即可完成 Join 操作。Mapper 的输入数据块是 *L* 进行拆分后的内容，而由于 *R* 足够小，所以将其分发给每个 Mapper 并在初始化时将其加载到内存存储，一般比较高效的方法是将 *R* 存入内存哈希表中，以外键作为哈希表的 Key，这样即可依次读入 *L* 的记录并查找哈希表来进行 Join 操作，其算法如下。

```
class Mapper
    initialize():
        H = new Associative Array : join_key -> tuple from R
        R = loadR();
        for all [ join_key k, tuple [r1, r2,...] ] in R
            H{k} = H{k}.append( [r1, r2,...] );

    map(join_key k, tuple l):
        for all tuple r in H{k}:
            Emit(null, tuple [k r l]) ;
```



与 Reduce-Side Join 相比，Map-Side Join 处理效率要高很多，因为其避免了 Shuffle 网络传输过程以及 Reduce 中的排序过程，但是其必须满足 R 小到可以在内存存储这一前提条件。

11.3 DAG 计算模型

DAG 是有向无环图（Directed Acyclic Graph）的简称，在数据结构领域里我们很早就接触过这个概念。在大数据处理领域，DAG 计算模型往往是指将计算任务在内部分解成为若干个子任务，这些子任务之间由逻辑关系或运行先后顺序等因素被构建成为有向无环图结构。

本节主要以微软的 Dryad 系统来讲解在批处理领域是如何构建 DAG 架构的。DAG 是在分布式计算中非常常见的一种结构，在各个细分领域都可见其身影，比如批处理中的 Dryad、FlumeJava 和 Tez，都是明确构建 DAG 计算模型的典型系统，再比如流式计算领域的 Storm 等系统抑或机器学习框架 Spark 等，其计算任务大多数也是以 DAG 的形式出现的，除此外还有很多场景都可见其踪影。DAG 之所以如此常见，是因为其通用性强，所以表达能力自然也强。比如前面介绍的 MapReduce 计算模型，在本质上是 DAG 的一种特例，在下文介绍 Dryad 的时候可以体会到这一点。

11.3.1 DAG 计算系统的三层结构

在大数据处理领域，很容易归纳出 DAG 计算系统比较通用的由上到下三层结构。

最上层是应用表达层，即通过一定手段将计算任务分解成由若干子任务形成的 DAG 结构，这层的核心是表达的便捷性，主要目的是方便应用开发者快速描述或者构建应用。因为 DAG 作为一种任务描述方式尽管表达能力足够强，但正是因其灵活性高，导致开发者适应这种表达方式有较大难度，也即学习和使用成本较高，所以如何降低使用成本是非常重要的，同时这也是应用表达层应该重点解决的问题。

最下层是物理机集群，即由大量物理机器搭建的分布式计算环境，这是计算任务最终执行的场所。