可扩展的搜索引擎解决方案——以 BBS 为例

黄一夫

(复旦大学 计算机科学技术学院,上海 200433)

摘 要: 在互联网飞速发展的今天，搜索引擎已经成为人类生活不可或缺的部分。本着动手实践学习搜索引擎背后的实现细节为原则，首先我们以 Apache 基金会下的一些开源项目为基础，设计并完成了可扩展的搜索引擎解决方案。然后以复旦大学日月光华 BBS 为例，进一步完成了部署实现。最后，BBS 搜索引擎成功完成并已投入使用，验证了可扩展的搜索引擎解决方案的可行性。

关键词: 可扩展;搜索引擎;BBS

1 介绍

1.1 项目背景

随着互联网的飞速发展，每时每刻都有大量的文档产生，如何快速并准确地对它们进行检索成为了一个很重要并困难的问题。Google 是该问题的最佳解决者。从 2003 年开始，Google 接连公布了如 GFS[1]，MapReduce[2]和 Bigtable[3]等搜索引擎中一些重要的设计方案，引起了学术界和工业界的轰动。随后，Apache 开源软件基金会在这些论文的基础上进行了探索尝试，集众人之力逐渐累积了 Hadoop[4]和 HBase[8]等开源项目。虽然开源的 Hadoop 生态系统由 Java 开发，从性能上看肯定和 Google 内部 C++开发的原型有一定的差距。但是，由于其扩展性良好，已被学术界和工业界进行疯狂地研究改造和二次开发，并在实际部署中证明了其应有的性能。

1.2 项目动机

首先，我们的目的是为复旦大学的日月光华 BBS 添加全文检索的功能。我们经常需要在 BBS 中寻找某个话题相关的帖子，或者需要把之前的看过帖子重新找出来等等。但是，目前 BBS 的版块设计杂乱无章，搜索功能极其微弱，更本无法满足这些需求。

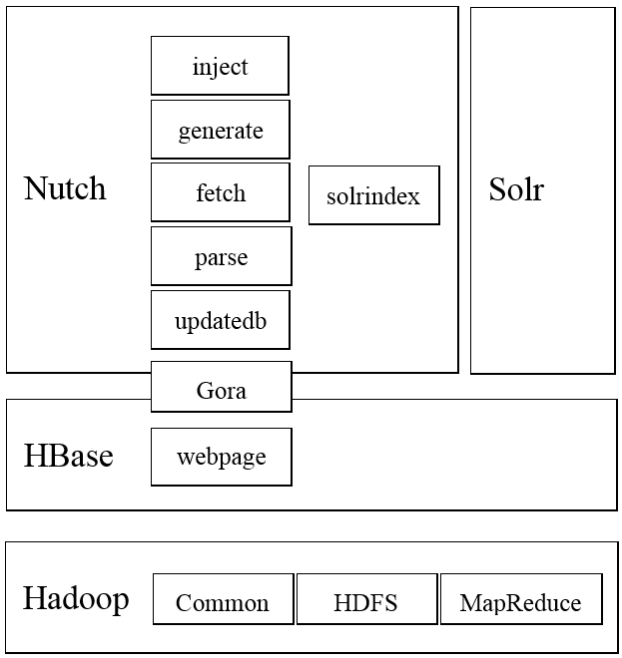
其次，我们在阅读了 Google 的相关论文后，无法把握搜索引擎背后的实现细节。本着动手实践学习搜索引擎背后的实现细节为原则，我们需要进一步对 Hadoop 生态系统的相关开源项目进行探索分析。

最后，考虑到项目工作的可重用性极高，我们把研究高度提升到完成一个可扩展的搜索引擎解决方案。虽然 Hadoop 生态系统的开源项目数量较多，功能强大，但是从整体看来，版本较为混乱，组织较为复杂。可扩展的搜索引擎解决方案可以使他人可以根据自己的需求快速地搭建出搜索引擎。

1.3 项目工作

我们完成了复旦大学日月光华 BBS 搜索引擎，虽然目前功能还很简陋，但是已经可以投入使用。我们完成了可扩展的搜索引擎解决方案，他人可以重用我们的项目成果快速地搭建出自己的搜索引擎。

2 架构



上图是可扩展的搜索引擎解决方案的项目整体架构。最底层是运行在 Linux 集群上的 Hadoop 系统，其主要由 Common, HDFS 和 MapReduce 三部分共同提供基础性的分布式服务，它是 Google 的 GFS 和 MapReduces 的开源实现。上面一层是基于 Hadoop 的分布式存储系统 HBase，搜索引擎中每一个网页对应于该存储系统中 webpage 表的每一行，它是 Google 的 Bigtable 的开源实现。最上面是 Nutch，其实质上就是一系列的 MapReduce 任务，运行在 Hadoop 系统上，并通过 Gora 这样的 ORM 模块将处理之后的数据存放在 HBase 的 webpage 表里，其主要完成了网页的爬取，解析和索引的建立。最后是 Solr，它是基于 Lucene 的搜索服务器，在索引建立完成之后，可以通过 HTTP 请求检索结果，它会以相应的格式返回，再开发相应的用户界面处理逻辑包装即可。

项目中用到的节点的 CPU 为 Intel(R) Core(TM)2 Duo CPU E7500 @ 2.93GHz，4GB 内存，一共 25 个节点。Hadoop 版本为 1.2.1，其中 namenode/jobtracker 节点 1 个，datanode/tasktracker 节点 24 个。HBase 版本为 0.90.4，其中 master 节点 1 个，zookeeper 节点 3 个，regionserver 节点 24 个。Nutch 版本为 2.2.1。Solr 版本为 4.4.0。数据来源于复旦大学日月光华 BBS，从 http://bbs.fudan.edu.cn/bbs/all 进入，经过两层的目录遍历便可获得所有帖子的 url。BBS 版块数目为 376，帖子数目为 3111945。

3 实现

3.1 Hadoop

3.1.1 Common

Common 为 Hadoop 的其他项目提供了一些常用工具，主要包括系统配置工具 Configuration、远程过程调用 RPC、序列化机制和 Hadoop 抽象文件系统 FileSystem 等。它们为在通用硬件上搭建云计算环境提供基本的服务，并为运行在该平台上的软件开发提供了所需的 API。

3.1.2 HDFS

HDFS(Hadoop Distributed File System)是 Hadoop 体系中数据存储管理的基础。它是一个高度容错的系统，能检测和应对硬件故障，用于在低成本的通用硬件上运行。HDFS 简化了文件的一致性模型，通过流式数据

访问，提供高吞吐量应用程序数据访问功能，适合带有大型数据集的应用程序。

设计预期:

①系统由许多廉价的普通组件组成，组件失效是一种常态。系统必须持续监控自身的状态，它必须将组件失效作为一种常态，能够迅速地侦测、冗余并恢复失效的组件。

②系统存储一定数量的大文件。我们预期会有几百万文件，文件的大小通常在 100MB 或者以上。数个 GB 大小的文件也是普遍存在，并且要能够被有效的管理。系统也必须支持小文件，但是不需要针对小文件做专门的优化。

③系统的工作负载主要由两种读操作组成：大规模的流式读取和小规模的随机读取。大规模的流式读取通常一次读取数百 KB 的数据，更常见的是一次读取 1MB 甚至更多的数据。来自同一个客户机的连续操作通常是读取同一个文件中连续的一个区域。小规模的随机读取通常是在文件某个随机的位置读取几个 KB 数 据。如果应用程序对性能非常关注，通常的做法是把小规模的随机读取操作合并并排序，之后按顺序批量读取，这样就避免了在文件中前后来回的移动读取位置。

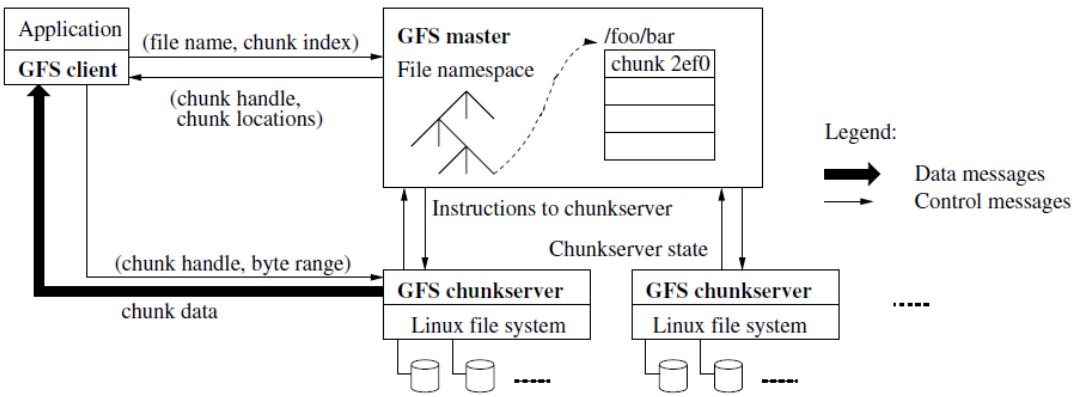
④系统的工作负载还包括许多大规模的、顺序的、数据追加方式的写操作。一般情况下，每次写入的数据的大小和大规模读类似。数据一旦被写入后，文件就很少会被修改了。系统支持小规模的随机位置写入操作，但是可能效率不彰。

⑤系统必须高效的、行为定义明确的实现多客户端并行追加数据到同一个文件里的语意。我们的文件通常被用于“生产者-消费者”队列，或者其它多路文件合并操作。通常会有数百个生产者，每个生产者进程运行在一台机器上，同时对一个文件进行追加操作。使用最小的同步开销来实现的原子的多路追加数据操作是必不可少的。文件可以在稍后读取，或者是消费者在追加的操作的同时读取文件。

⑥高性能的稳定网络带宽远比低延迟重要。我们的目标程序绝大部分要求能够高速率的、大批量的处理数据，极少有程序对单一的读写操作有严格的响应时间要求。

GFS 架构:

下图是 GFS(HDFS)的架构图，其中 master(namenode)主要管理整个分布式文件系统的命名服务，以及和客户端之间的交互，在客户端看来，整个分布式文件的根目录为 hdfs://namenode:9000/；chunkserver(datanode)将实际的数据进行多次备份存储，以提高系统的容错性和鲁棒性，它们从 master(namenode)处接收处理指令，然后根据指令直接和客户端进行数据传输。



GFS 存储的文件都被分割成固定大小的 Chunk。在 Chunk 创建的时候，Master 服务器会给每个 Chunk 分配一个不变的、全球唯一的 64 位的 Chunk 标识。Chunk 服务器把 Chunk 以 linux 文件的形式保存在本地硬盘上，并且根据指定的 Chunk 标识和字节范围来读写块数据。出于可靠性的考虑，每个块都会复制到多个块服务器上。默认情况下，使用 3 个存储复制节点，不过用户可以为不同的文件命名空间设定不同的复制级别。

Master 节点管理所有的文件系统元数据。这些元数据包括名字空间、访问控制信息、文件和 Chunk 的映射信息、以及当前 Chunk 的位置信息。Master 节点还管理着系统范围内的活动，比如，Chunk 租用管理、孤

儿 Chunk 的回收、以及 Chunk 在 Chunk 服务器之间的迁移。Master 节点使用心跳信息周期地和每个 Chunk 服务器通讯，发送指令到各个 Chunk 服务器并接收 Chunk 服务器的状态信息。

GFS 客户端代码以库的形式被链接到客户程序里。客户端代码实现了 GFS 文件系统的 API 接口函数、应用程序与 Master 节点和 Chunk 服务器通讯、以及对数据进行读写操作。客户端和 Master 节点的通信只获取元数据，所有的数据操作都是由客户端直接和 Chunk 服务器进行交互的。我们不提供 POSIX 标准的 API 的功能，因此，GFS API 调用不需要深入到 Linux vnode 级别。

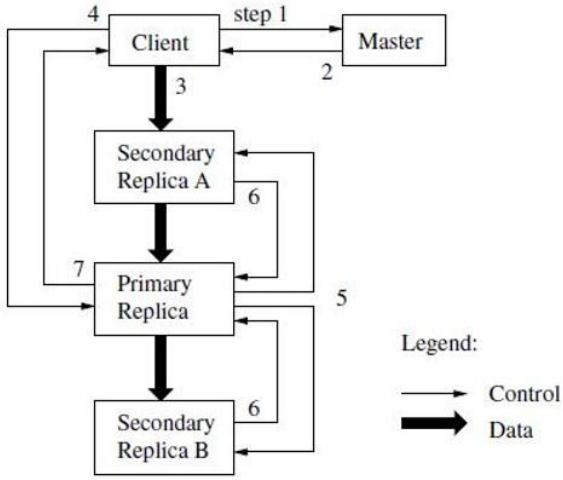
无论是客户端还是 Chunk 服务器都不需要缓存文件数据。客户端缓存数据几乎没有什么用处，因为大部分程序要么以流的方式读取一个巨大文件，要么工作集太大根本无法被缓存。无需考虑缓存相关的问题也简化了客户端和整个系统的设计和实现。（不过，客户端会缓存元数据。）Chunk 服务器不需要缓存文件数据的原因是，Chunk 以本地文件的方式保存，Linux 操作系统的文件系统缓存会把经常访问的数据缓存在内存中。

上图的简单流程是，首先，客户端把文件名和程序指定的字节偏移，根据固定的 Chunk 大小，转换成文件的 Chunk 索引。然后，它把文件名和 Chunk 索引发送给 Master 节点。Master 节点将相应的 Chunk 标识和副本的位置信息发还给客户端。客户端用文件名和 Chunk 索引作为 key 缓存这些信息。

之后客户端发送请求到其中的一个副本处，一般会选择最近的。请求信息包含了 Chunk 的标识和字节范围。在对这个 Chunk 的后续读取操作中，客户端不必再和 Master 节点通讯了，除非缓存的元数据信息过期或者文件被重新打开。实际上，客户端通常会在一次请求中查询多个 Chunk 信息，Master 节点的回应也可能包含了紧跟着这些被请求的 Chunk 后面的 Chunk 的信息。在实际应用中，这些额外的信息在没有任何代价的情况下，避免了客户端和 Master 节点未来可能会发生的几次通讯。

系统交互:

在设计这个系统时，一个重要的原则是最小化所有操作和 Master 节点的交互。带着这样的设计理念，我们现在描述一下客户机、Master 服务器和 Chunk 服务器如何进行交互，下图描述的租约（lease）和变更顺序。



①客户机向 Master 节点询问哪一个 Chunk 服务器持有当前的租约，以及其它副本的位置。如果没有一个 Chunk 持有租约，Master 节点就选择其中一个副本建立一个租约（这个步骤在图上没有显示）。

②Master 节点将主 Chunk 的标识符以及其它副本（又称为 secondary 副本、二级副本）的位置返回给客户机。客户机缓存这些数据以便后续的操作。只有在主 Chunk 不可用，或者主 Chunk 回复信息表明它已不再持有租约的时候，客户机才需要重新跟 Master 节点联系。

③客户机把数据推送到所有的副本上。客户机可以以任意的顺序推送数据。Chunk 服务器接收到数据并保存在它的内部 LRU 缓存中，一直到数据被使用或者过期交换出去。由于数据流的网络传输负载非常高，通过分离数据流和控制流，我们可以基于网络拓扑情况对数据流进行规划，提高系统性能，而不用去理

会哪个 Chunk 服务器保存了主 Chunk。

④当所有的副本都确认接收到了数据，客户机发送写请求到主 Chunk 服务器。这个请求标识了早前推送到所有副本的数据。主 Chunk 为接收到的所有操作分配连续的序列号，这些操作可能来自不同的客户机，序列号保证了操作顺序执行。它以序列号的顺序把操作应用到它自己的本地状态中。

⑤主 Chunk 把写请求传递到所有的二级副本。每个二级副本依照主 Chunk 分配的序列号以相同的顺序执行这些操作。

⑥所有的二级副本回复主 Chunk，它们已经完成了操作。

⑦主 Chunk 服务器回复客户机。任何副本产生的任何错误都会返回给客户机。在出现错误的情况下，写入操作可能在主 Chunk 和一些二级副本执行成功。（如果操作在主 Chunk 上失败了，操作就不会被分配序列号，也不会被传递。）客户端的请求被确认为失败，被修改的 region 处于不一致的状态。我们的客户机代码通过重复执行失败的操作来处理这样的错误。在从头开始重复执行之前，客户机会先从步骤 3 到步骤 7 做几次尝试。

3.1.3 MapReduce

MapReduce 概述:

MapReduce 是一种计算模型，用以进行大数据量的计算。Hadoop 的 MapReduce 实现，和 Common、HDFS 一起，构成了 Hadoop 发展初期的三个组件。MapReduce 将应用划分为 Map 和 Reduce 两个步骤，其中 Map 对数据集上的独立元素进行指定的操作，生成键-值对形式中间结果。Reduce 则对中间结果中相同“键”的所有“值”进行规约，以得到最终结果。MapReduce 这样的功能划分，非常适合在大量计算机组成的分布式并行环境里进行数据处理。

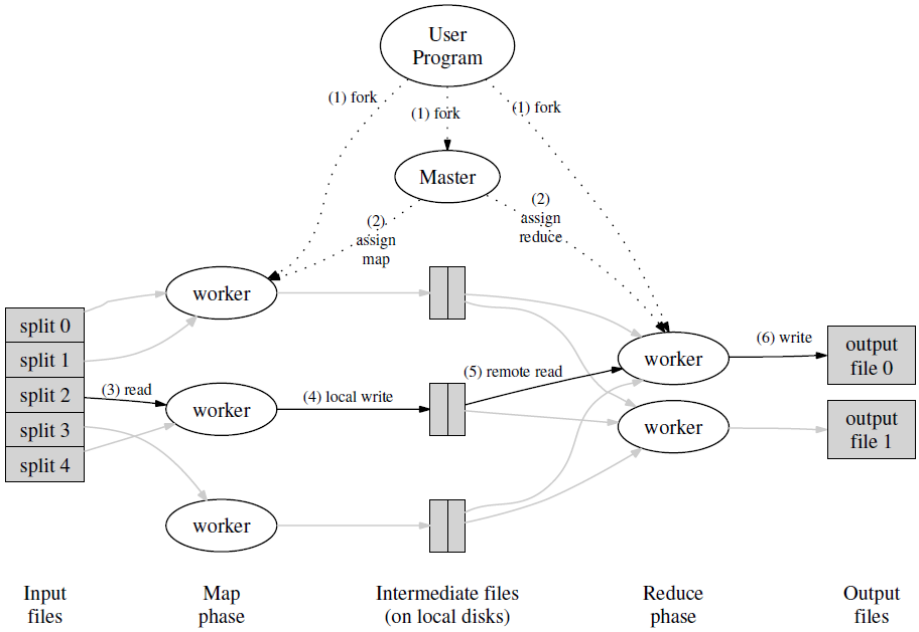
映射和化简:

简单说来，一个映射函数就是对一些独立元素组成的概念上的列表（例如，一个测试成绩的列表）的每一个元素进行指定的操作（比如前面的例子里，有人发现所有学生的成绩都被高估了一分，他可以定义一个“减一”的映射函数，用来修正这个错误）。事实上，每个元素都是被独立操作的，而原始列表没有被更改，因为这里创建了一个新的列表来保存新的答案。这就是说，Map 操作是可以高度并行的，这对高性能要求的应用以及并行计算领域的需求非常有用。

而化简操作指的是对一个列表的元素进行适当的合并（继续看前面的例子，如果有人想知道班级的平均分该怎么做？他可以定义一个化简函数，通过让列表中的元素跟自己的相邻的元素相加的方式把列表减半，如此递归运算直到列表只剩下一个元素，然后用这个元素除以人数，就得到了平均分）。虽然他不如映射函数那么并行，但是因为化简总是有一个简单的答案，大规模的运算相对独立，所以化简函数在高度并行环境下也很有用。

MapReduce 实现原理:

下图是一个统计词频的 MapReduce 程序框架示意。



从上图可以看出，master 相当于 Hadoop 中的 jobtracker，其负责调度和监控集群中的 worker(tasktracker)。输入分片后被分配到 Map 阶段的各个 worker(tasktracker)上进行具体的计算，然后进行统一的 shuffle，最后分配到 Reduce 阶段的各个 worker(tasktracker)上进行归并输出。用户程序需要分别实现 map 和 reduce 函数以及其输入输出类型即可。具体的实现过程有以下步骤：

①MapReduce 库先把 user program 的输入文件划分为 M 份（M 为用户定义），每一份通常有 16MB 到 64MB，如图左方所示分成了 split0~4；然后使用 fork 将用户进程拷贝到集群内其它机器上。

②user program 的副本中有一个称为 master，其余称为 worker，master 是负责调度的，为空闲 worker 分配作业（Map 作业或者 Reduce 作业），worker 的数量也是可以由用户指定的。

③被分配了 Map 作业的 worker，开始读取对应分片的输入数据，Map 作业数量是由 M 决定的，和 split 一一对应；Map 作业从输入数据中抽取出键值对，每一个键值对都作为参数传递给 map 函数，map 函数产生的中间键值对被缓存在内存中。

④缓存的中间键值对会被定期写入本地磁盘，而且被分为 R 个区，R 的大小是由用户定义的，将来每个区会对应一个 Reduce 作业；这些中间键值对的位置会被通报给 master，master 负责将信息转发给 Reduce worker。

⑤master 通知分配了 Reduce 作业的 worker 它负责的分区在什么位置（肯定不止一个地方，每个 Map 作业产生的中间键值对都可能映射到所有 R 个不同分区），当 Reduce worker 把所有它负责的中间键值对都读过来后，先对它们进行排序，使得相同键的键值对聚集在一起。因为不同的键可能会映射到同一个分区也就是同一个 Reduce 作业（谁让分区少呢），所以排序是必须的。

⑥reduce worker 遍历排序后的中间键值对，对于每个唯一的键，都将键与关联的值传递给 reduce 函数，reduce 函数产生的输出会添加到这个分区的输出文件中。

⑦当所有的 Map 和 Reduce 作业都完成了，master 唤醒正版的 user program，MapReduce 函数调用返回 user program 的代码。

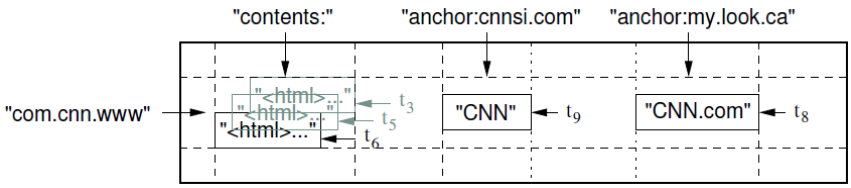
所有执行完毕后，MapReduce 输出放在了 R 个分区的输出文件中（分别对应一个 Reduce 作业）。用户通常并不需要合并这 R 个文件，而是将其作为输入交给另一个 MapReduce 程序处理。整个过程中，输入数据是来自底层分布式文件系统（GFS）的，中间数据是放在本地文件系统的，最终输出数据是写入底层分布式文件系统（GFS）的。而且我们要注意 Map/Reduce 作业和 map/reduce 函数的区别：Map 作业处理一个输入数

据的分片，可能需要调用多次 map 函数来处理每个输入键值对；Reduce 作业处理一个分区的中间键值对，期间要对每个不同的键调用一次 reduce 函数，Reduce 作业最终也对应一个输出文件。

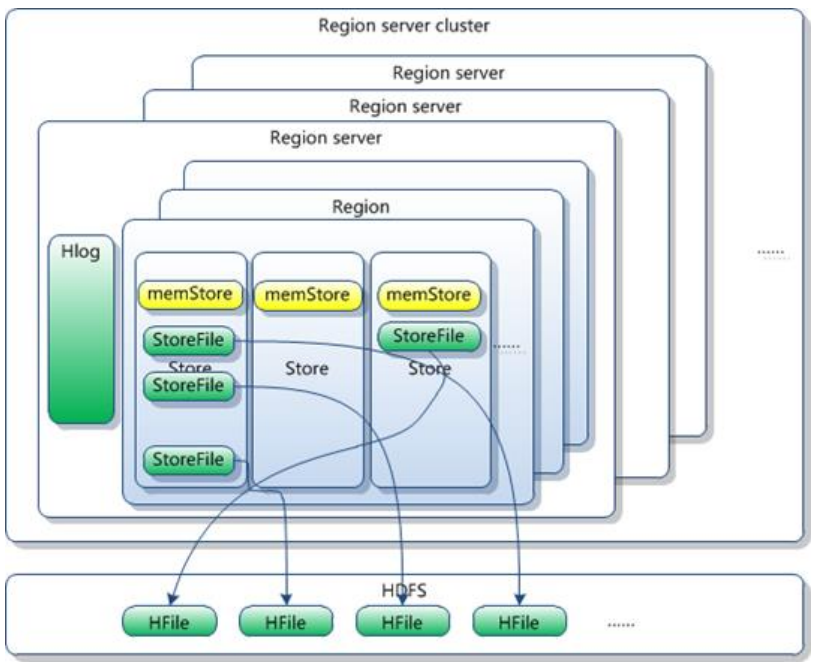
3.2 HBase

Google 发表了 Bigtable 系统论文后，开源社区就开始在 HDFS 上构建相应的实现 HBase。HBase 是一个针对结构化数据的可伸缩、高可靠、高性能、分布式和面向列的动态模式数据库。和传统关系数据库不同，HBase 采用了 Bigtable 的数据模型：增强的稀疏排序映射表(Key/Value)，其中，键由行关键字，列关键字和时间戳构成。HBase 提供了对大规模数据的随机、实时读写访问，同时，HBase 中保存的数据可以使用 MapReduce 来处理，它将数据存储和并行计算完美地结合在一起。

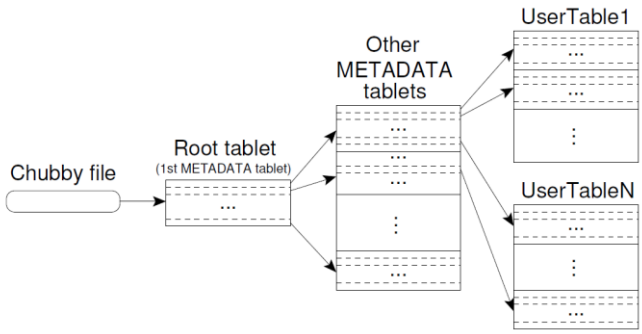
HBase 的数据模型实质上就是一个巨大的 map，key 是一个三元组，包括行关键字，列关键字和时间戳，value 是一个字符串。对于行而言，对每一行的操作是原子的，数据按行排序，一定范围的行形成一个 region(tablet)。对于列而言，以列族：标识符的形式存在，列族的存在主要是出于权限控制，以及数据获取局部性上的考虑。对于时间戳而言，以倒序的形式进行排列，方便获取最新数据以及旧版本数据的垃圾回收。下图是一个简单的例子，作为行关键字的 url 经倒序处理，使得同一主机的 url 相邻存放。



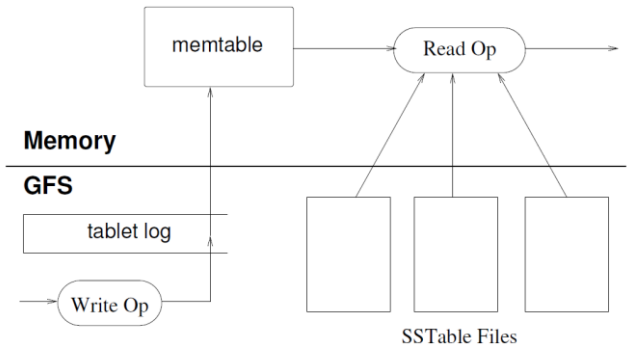
HBase 的控制主要由 master 和 zookeeper(chubby)完成，存储主要在 regionserver 集群上面。每一个节点对应一个 regionserver。一个 regionserver 对应多个 region(tablet)和一个 log，log 用于宕机恢复。一个 region(tablet)对应多个 store，每个列族是存储在一个 store 上的。一个 store 对应一个 memStore(memtable)和多个 storeFile(SSTable)，memStore(memtable)为内存中的数据，storeFile(SSTable)为存储在 HDFS 上的数据，两者会根据运行需求进行相应的调度。具体如下图所示。



HBase 与 Bigtale 相同，使用三层类似 B+树的结构来保存 region(tablet)位置。第一层是保存 zookeeper(chubby)里面的文件，它持有 root region(tablet)的位置。第二层 root region(tablet)是.META.表的第一个 region(tablet)其中保存了.META.z 表其它 region(tablet)的位置。通过 root region(tablet)，我们就可以访问.META.表的数据。.META.是第三层，它是一个特殊的表，保存了 HBase 中所有数据表的 region(tablet)位置信息。下图是 Bigtale 中 tablet 的索引示例。



至于 HBase 的读写过程可以用下图进行解释。当执行写操作的时候，先写日志，然后再写内存表，这样可以提高容错性，便于灾难恢复。当执行读操作的时候，首先查找内存表，如果无法找到的话，再进行合理的调度，更新内存表，然后进行访问。



通过 Bigtable 的 benchmark 实验我们可以发现，Bigtable 对于各种需求不同的应用都有良好的支持，这些需求主要指数据量的大小以及延迟的快慢。

3.3 Gora

Gora 是一个开源的 ORM 框架，主要为大数据提供内存数据模型与数据的持久化。Gora 原生支持使用 Hadoop 的 MapReduce 框架进行计算，目前已经整合到 Nutch 2.0 中了。另一方面，通过 gora-hbase 模块，Gora 可以将 HBase 作为自己的后端进行数据存储。通过对 Gora 的使用，我们可以将 Nutch 中爬取，解析和索引建立等工作完美地进行分布式地实现，这样使得整个搜索引擎获得了可扩展的性能。

3.4 Nutch

3.4.1 Inject

关键思想: 所要爬取的 url 数量庞大，使用单台机器爬取无法满足需求。我们把 url 分割后分配到不同的节点上去，然后将他们插入到 HBase 的 webpage 表中。

Map: (offset, line, reversedUrl, page)。对存放 url 的文件中的每一行，根据正则匹配规范化 url，根据正则匹配、前缀、后缀、域名过滤 url，翻转 url 按域名排序，最后保存下获取时间，获取间隔，元数据，分数，记号等，插入到 HBase 的 webpage 表中的每一行。

Reduce: 默认 Reducer。

3.4.2 Generate

关键思想: 存放在 HBase 的 webpage 表中的 url 数量庞大，我们要按需求选择出一部分的 url，把它们 map 到不同的节点上去，做一些标记，准备下一阶段的获取。

Map: (reversedUrl, page, (url, scorce), page)。依次检查 HBase 的 webpage 表中的每一行的标记，距离，获取调度，评分等，判断 url 是否需要进行下一阶段的获取。

Reduce: ((url, scorce), page, reversedUrl, page)。依次记录同一主机名和域名下 url 的数量，并设置标记等，以便下一阶段的获取。

3.4.3 Fetch

关键思想: 对于需要获取的 url，我们将它们 map 到不同的节点上去，并在每个节点内使用多线程的形式进行并行获取。

Map: (reversedUrl, page, random_id, (conf, reversedUrl, page))。依次检查 HBase 的 webpage 表中的每一行的序号，标记，可恢复项等，判断 url 是否需要获取。

Reduce: (random_id, (conf, reversedUrl, page), reversedUrl, page)。这里采用了生产者消费者模型。一个生产者不停地向队列中填充所要获取的 url，多个消费者不停地从队列中获取 url 并执行获取。获取的过程依次为，检查机器人协议，检查爬取延迟调度，获取页面内容，判断返回状态等。

3.4.4 Parse

关键思想: 上一阶段已经将 url 对应的网页下载到 HBase 的 webpage 表里，我们现在需要将 url 分配到不同的节点上，然后将网页中各个域的值抽取出来，最后存放在 HBase 中的 webpage 表里。

Map: (reversedUrl, page, reversedUrl, page)。处理过程就是将网页中各个域抽取出来，例如 text，title，signature 和 outlinks 等，然后将相应的值存放在 HBase 的 webpage 表里。

Reduce: 默认 Reducer。

3.4.5 Updatedb

关键思想: 上一阶段提取出了每个网页中的 outlinks，本阶段就是将所有的 url 分配到不同的节点上去，并准备对所有的 outlinks 进行获取。

Map: (reversedUrl, page, (reversedOut, score), pageOut)。该过程主要检测 outlink 的距离是否在可接受范围内。

Reduce: ((reversedOut, score), pageOut, reversedOut, pageOut)。该过程主要更新新生成的 outlinks 中对应的 inlinks 域，并准备好下一轮爬取。

3.4.6 Solrindex

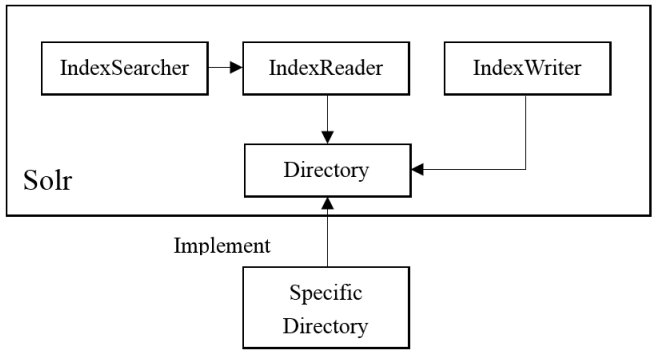
关键思想: 我们把 url 分配到不同的节点上，生成抽象的 document，然后向 solr 服务器建立索引，以便之后的查询。

Map: (reversedUrl, page, reversedUrl, doc)。该过程主要生成序号，指纹，批序号，域权重和内容等项。

Reduce: 默认 Reducer。

3.5 Solr

Solr 是一个开源的搜索服务器，使用 Java 语言开发，主要基于 HTTP 和 Lucene 实现。Solr 中存储的资源是以 Document 为对象进行存储的。每个文档由一系列的 Field 构成，每个 Field 表示资源的一个属性。



上图解释了 Solr 的工作过程，核心数据结构是抽象类 Directory，这是索引读写的单位，并可简单地扩展到其他特定的实现。在索引构建阶段，使用 IndexWriter 分析文档，建立起倒排索引。在检索阶段，IndexSearcher 使用 IndexReader 读取索引，并根据查询进行相应的返回。

4 展示

4.1 BBS搜索

我们完成的复旦大学日月光华 BBS 搜索引擎，可以在复旦大学内网条件下以 10.171.5.222:3000 地址进行访问。整个项目的源代码和配置文件使用 Git 进行版本控制，以私有资产的形式托管在 BitBucket 上。

查询语法方面，目前使用了最为简单的规则：以空格代替逻辑与的形式。返回结果方面，目前强制返回前 100 条结果。可以发现，目前的功能较为简陋，但是从投入使用的情况来看，已经能在一定程度上进行有效的检索。更加复杂和全面的工作我们将在后续的时间里进行完善。

上图是复旦大学日月光华 BBS 搜索引擎的内网访问地址和用户界面。

4.2 BBS搜索结果

上图是输入查询”姜育刚”之后返回的部分内容。

上图是输入查询”姜育刚 课题组”之后返回的内容。

5 讨论

5.1 结果

首先，我们完成了复旦大学日月光华 BBS 搜索引擎，虽然目前功能还很简陋，但是已经可以投入使用。其次，我们完成了可扩展的搜索引擎解决方案，他人可以重用我们的项目成果快速搭建自己的搜索引擎。

5.2 未来工作

虽然从整体上看来，工作已经完成，但是还有很多细节需要进一步的工作。首先，我们需要设计出增量的 url 爬取器，这样定时进行爬取，才能使得搜索引擎检索的实时性增强。其次，我们需要对内容进行更加细粒度的提取，进一步开发出域检索，这样将极大地提升可用性。最后，我们还打算引入注册机制，从用户的浏览历史中建立模型，推荐感兴趣的网页。

References:

[1] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung: The Google file system. SOSP 2003:29-43.

[2] Jeffrey Dean, Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004:137-150.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, Robert Gruber: Bigtable: A Distributed Storage System for Structured Data. OSDI 2006:205-218.

[4] Tom White: Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale. O'Reilly 2012.

[5] Jimmy Lin, Chris Dyer: Data-Intensive Text Processing with MapReduce Morgan & Claypool Publishers 2010.

[6] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler: The Hadoop Distributed File System. MSST 2010:1-10.

[7] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, , Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O’Malley, Sanjay Radia, Benjamin Reed, and Eric

Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In Proc. of the 4th ACM Symposium on Cloud Computing, SOCC '13, 2013.

[8] Lars George: HBase - The Definitive Guide: Random Access to Your Planet-Size Data. O'Reilly 2011.

[9] Luiz André Barroso, Jeffrey Dean, Urs Hölzle: Web Search for a Planet: The Google Cluster Architecture. IEEE Micro (MICRO) 23(2):22-28 (2003).

[10] Jeffrey Dean, Monika Rauch Henzinger: Finding Related Pages in the World Wide Web. Computer Networks (CN) 31(11-16):1467-1479 (1999).

[11] Michael Burrows: The Chubby Lock Service for Loosely-Coupled Distributed Systems. OSDI 2006:335-350.

[12] Sergey Brin, Lawrence Page: The Anatomy of a Large-Scale Hypertextual Web Search Engine. Computer Networks (CN) 30(1-7):107-117 (1998).