为了满足Google迅速增长的数据处理需求，Google设计并实现了Google文件系统(GFS，Google File System)。GFS与过去的分布式文件系统拥有许多相同的目标，例如性能、可伸缩性、可靠性以及可用性。

GFS最开始的设计思路是：

1.组件失效被认为是常态事件，而不是意外事件：

由于参与运算与处理的节点数目非常庞大，通常会使用上千个节点进行共同计算，因此，每时每刻总会有节点处在失效状态。需要通过软件程序模块，监视系统的动态运行状况，侦测错误，并且将容错以及自动恢复系统集成在系统中。

2.以通常的标准衡量，我们的文件非常巨大：

Google系统中的文件大小与通常文件系统中的文件大小概念不一样，文件大小通常以G字节计。另外文件系统中的文件含义与通常文件不同，一个大文件可能包含大量数目的通常意义上的小文件。所以，设计预期和参数，例如I/O操作和块尺寸都要重新考虑。

3.绝大部分文件的修改是采用在文件尾部追加数据，而不是覆盖原有数据的方式：

在Google应用(如搜索)中对大部分文件的修改，不是覆盖原有数据，而是在文件尾追加新数据。对文件的随机写是几乎不存在的。对于这类巨大文件的访问模式，客户端对数据块缓存失去了意义，追加操作成为性能优化和原子性(把一个事务看做是一个程序。它要么被完整地执行，要么完全不执行)保证的焦点。

4.应用程序和文件系统 API 的协同设计提高了整个系统的灵活性：

文件系统的某些具体操作不再透明，而且需要应用程序的协助完成，应用程序和文件系统API的协同设计提高了整个系统的灵活性。例如，放松了对GFS一致性模型的要求，这样不用加重应用程序的负担，就大大简化了文件系统的设计。还引入了原子性的追加操作，这样多个客户端同时进行追加的时候，就不需要额外的同步操作了。

由此，Google在设计中做出了如下的假设：

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。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.负载通常包含两种读：大型的流式读（顺序读），和小型的随机读。前者通常一次读数百KB以上，后者通常在随机位置读几个KB。

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，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。

6.高性能的稳定网络带宽远比低延迟重要。

Google在2004年发表了一篇论文：MapReduce: Simplified Data Processing on Large Clusters，介绍了他们内部如何实现和使用MapReduce。

简单地说，MapReduce是一个受限的分布式并行编程模型，可用于处理和输出很大的数据集。而编写MapReduce任务的用户只需要实现两个函数：

Map函数：输入一个key/value数据，输出一个key/value形式的中间数据集。

Reduce函数：输入是一个中间数据的key和一个与这个key对应的value集合。它负责将这些值按照一定的规则进行“合并”。

MapReduce全程概览

MapReduce框架对输入的文件数据分成M片，每份数据的大小为16~64MB（可由用户配置）。

在多台机器上开始运行User Program：包括一个master、多个map worker和多个reduce worker。

master主要负责map worker和reduce worker的状态管理和任务分发。

map worker从GFS读取分配到的文件数据，并进行相应的处理。MapReduce框架的调度会尽量使map worker运行的机器与数据靠近，以提高数据传输的效率。所以，数据传输可以是本地，也可能是网络。

map worker的输出缓存在内存中，并定期刷到本地磁盘上。这些中间数据的位置信息会通过心跳信息告诉master，master记下这些信息后，通知reduce worker。数据存储在本地。

reduce worker通过RPC从map worker读取需要的中间数据。数据通过网络传输。

reduce worker对中间数据进行“合并”处理后，输出结果。

容错

map worker

map任务执行完成后宕机：因为中间数据存储在本地磁盘，需要重新执行。

map任务执行完成前宕机：需要重新执行。

reduce worker

reduce任务执行完成后宕机：因为数据存储在GFS，不需要重新执行。

reduce任务执行完成前宕机：需要重新执行，输出文件可以覆盖原来的（文件名一样）。

master宕机，任务失败。（master是个单点）

优化

局部性：MapReduce用于[大数据](https://cloud.tencent.com/solution/bigdata?from=10680" \t "_blank)集的处理，其主要瓶颈是网络带宽。通过优化调度，可以让执行MapReduce任务的机器尽可能靠近机器。（同一机器==>同一机架==>同一机房...）

任务粒度：执行MapReduce任务的过程其实就是M个Map任务+R个Reduce任务。M和R必须比机器数大很多才会有利于负载均衡。

备份任务：当MapReduce任务即将执行完成时，MapReduce框架会针对那些还在执行的任务，启动一个对应的备份任务。之后，只要主任务或备份任务执行完成，MapReduce任务就完成了。这样可以有效避免整个MapReduce任务被少部分比较慢的机器拖死。

BigTable是谷歌在2006年的一篇论文中提出来的，为后来hbase的出现提供了理论基础。

Bigtable是一个稀疏的、分布式的、持久化存储的多维排序Map。Map的索引是行关键字、列关键字以及时间戳；Map中的每个vaulue都是一个未经解析的byte数组。

Bigtable中的行关键字可以是任意的字符串，并且每行的读写操作都是原子的；Bigtable中的行关键字是按照字典顺序排序存储的，表中的行都可以进行动态分区，每个分区叫tablet，tablet是数据分布和负载均衡的最小单位。由于行键是按照字典序存储的，所以查询时以行关键字作为条件查询速度毫秒级。

列关键字组成的集合叫做“列族”，列族时访问控制的基本单位。列关键字的命名语法如下：列族：限定词。

每行数据都会有用来当作版本号的时间戳，可以系统自动赋值，也可以用户自己指定。最新的数据行排在最前面。还可以利用时间戳来进行垃圾收集。

Bigtable使用Google的分布式文件系统GFS存储日志文件和数据文件。BigTable内部存储数据的问文件时Google SSTable格式的。SSTable是一个持久化的、排序的、不可更改的Map<key,value>结构，其值都是任意的byte串，因此使用key查询速度很快。Big Table还依赖一个高可用的、序列化的分布式锁服务组件——Chubby。BigTable使用Chubby完成以下几个任务：

确保在任何时间内最多只有一个活动的Master副本；

存储BigTable数据的自引导指令的位置

查找Tablet服务器，以及在Tablet服务器失效时进行善后；

存储BigTable模式信息

存储访问控制列表。

BigTable包括了三个主要的组件：链接到客户程序的库、一个Mater服务器和多个Tablet。针对系统工作负载的变化情况，BigTable可以动态的向集群添加或者删除Tablet服务器。

Master服务器主要为Tablet服务器分配Tablets、检测新加入的或者过期失效的Tablet服务器、对Tablet服务器进行负载均衡、以及对保存在GFS上的文件进行垃圾收集。除此之外，还处理模式的相关修改操作，例如建立表和列族。

每个Tablet服务器都管理一个Tablet的集合，每个Tablet的服务器负责处理它所加载的Tablet的读写操作，以及在Tablets过大时，对其进行分割。

客户端读取的数据都不经过Master服务器；客户程序直接和Tablet服务器通信进行读写操作。

在任何一个时刻，一个Tablet只能分配给一个Tablet服务器。Master服务器记录了当前有那些活跃的Tablet服务器、那些Tablet分配给了那些Tablet服务器、那些Tablet还没有被分配。

BigTable使用Chubby跟踪记录Tablet服务器的状态。当一个Tablet服务器启动时，它在Chubby的一个指定目录下建立一个有唯一性名字的文件，并且获取该文件的独占锁。Master服务器实时监控着这目录，因此Master服务能够知道有新的Tablet服务器加入了。只要文件存在Tablet服务器就会试图重新获得对该文件的独占锁，如果文件不存在了，那么

Tablet服务器就不能在提供服务了。

Master 服务器从Chubby 获取一个唯一的Master 锁，用来阻止创建其它的Master 服务器实例；

Master 服务器扫描Chubby 的服务器文件锁存储目录，获取当前正在运行的服务器列表；

Master 服务器和所有的正在运行的Tablet 表服务器通信，获取每个Tablet 服务器上Tablet 的分配信息；

Master 服务器扫描METADATA 表获取所有的Tablet 的集合。

在扫描的过程中，当Master 服务器发现了一个还没有分配的Tablet，Master 服务器就将这个Tablet 加入

未分配的Tablet 集合等待合适的时机分配。

Tablet的持久化状态信息保存在GFS上。更新操作提交到REDO日志中。这些更新操作中，最近提交的那些放在一个排序的缓存中，我们称这个缓存为memtable；较早更新存放在一系列的SSTable中。

随着写操作的执行，memtable的大小不断增加。当memtable的尺寸到达一个门限值的时候，这个memtable就会被冻结，然后创建一个新的memtable；被冻结住的memtable会被转换成SSTable，然后写入GFS。

客户程序可以将多个列族组合成一个局部性群族。对Tablet中的每个局部性群族都生成一个单独的SSTable。将同城不会一起访问的列族分割成不同的局部性群族可以提高读取操作的效率

客户程序可以控制一个局部性群族的SSTable是否需要压缩，一般使用两遍的、可定制的压缩

为了提高读操作的性能，Tablet服务器使用二级缓存的策略，一级用来缓存Tablet服务器通过SSTable接口的Key-Value对；Block是二级缓存，用来缓存从GFS读取的SSTable的Block。

整个BigTable设计符合大部分大数据程序的需求，打破了关系型数据库的结构化存储，能够部署在成千上万台服务器上，可以存储PB级数据，对整个互联网行业的快速发展提供了坚实的理论基础与成功案例。