论文：NoSQL 在云计算中的应用

前言

随着云计算热潮的到来，越来越多的人认识到云计算是未来技术发展的一个主流方向，未来更多应用和技术会附着于云计算的概念。

什么是云计算

狭义云计算是指 IT 基础设施的交付和使用模式，指通过网络以按需、易扩展的方式获得所需的资源（硬件、平台、软件）。 提供资源的网络被称为"云"。"云"中的资源在使用者看来是可以无限扩展的，并且可以随时获取，按需使用，随时扩展，按使用付费。这种特性经常被称为像水 电一样使用 IT 基础设施。

广义云计算是指服务的交付和使用模式，指通过网络以按需、易扩展的方式获得所需的服务。这种服务可以是 IT 和软件、互联网相关的，也可以使任意其他的服务。

关系型数据库

数据库是一种用来保存数据的技术手段，经过了几代的发展，目前主流的数据库是关系型数据库。

关系型数据库是指采用了关系模型来组织数据的数据库。关系模型是在 1970 年由 IBM 的研究员 E.F.Codd 博士首先提出，在之后的几十年中，关系模型的概念 得到了充分的发展并逐渐成为数据库架构的主流模型。简单来说，关系模型指的就是二维表格模型，而一个关系型数据库就是由二维表及其之间的联系组成的一个数 据组织。下面列出了关系模型中的常用概念。

关系：可以理解为一张二维表，每个关系都具有一个关系名，就是通常说的表名。

元组：可以理解为二维表中的一行，在数据库中经常被称为记录。

属性：可以理解为二维表中的一列，在数据库中经常被称为字段。

域：属性的取值范围，也就是数据库中某一列的取值限制。

关键字：一组可以唯一标识元组的属性。数据库中常称为主键，由一个或多个列组成。

关系模式：指对关系的描述，其格式为：关系名（属性 1，属性 2，…，属性 N）。在数据库中通常称为表结构。

二维表结构是非常贴近逻辑世界的一个概念，关系模型相对网状、层次等其他模型来说更容易理解。

关系型数据库的优点是：通用的 SQL 语言使得操作关系型数据库非常方便，程序员甚至于数据管理员可以方便地在逻辑层面操作数据库，而完全不必理解其底层实现。

同时，关系型数据库易于维护：丰富的完整性（实体完整性、参照完整性和用户定义的完整性）大大降低了数据冗余和数据不一致的概率。

技术发展

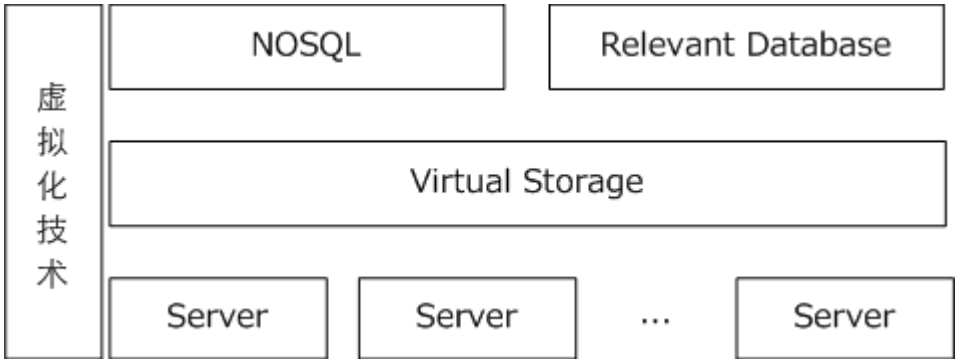
目前云计算已经得到充分的发展，各个厂商都看到云计算的商机，但是真正的发展还有一定局限性，很少有具体落地的技术。
在目前的互联网应用中，Facebook，Google 等主流网站都在承受着巨大的访问压力和数据流量。
同时，通过研究发现，这类应用涉及到的数据事物相当少，主要是简单非事物处理，这样可以抛弃臃肿的关系型数据库。
针对这类业务，Google 提出了基于 GFS 的 BigTable 技术，Facebook 的 Cassandra。

什么叫做 NOSQL

目前依据标准我们将 BigTable，Cassandra 归类为 NOSQL。
我们将目前流行的 Nosql 的接口模型做一个比较：

	Data Model	Query API
Cassandra	Columnfamily	Thrift
CouchDB	Document	map/reduce views
HBase	Columnfamily	Thrift, REST
MongoDB	Document	Cursor
Neo4J	Graph	Graph
Redis	Collection	Collection
Riak	Document	Nested hashes
Scalaris	Key/value	get/put
Tokyo Cabinet	Key/value	get/put
Voldemort	Key/value	get/put

NoSQL 主要是针对频繁访问并数据结构简单的数据操作使用对象数据库来实现，即提高了访问速度，也可以灵活的进行扩展。
怎么将数据库技术同云计算技术结合?
先看一下下面的简单结构图：



在这个图中，基于虚拟化的技术来实现（当然，并非只有虚拟化的技术才能实现，虚拟化只是手段之一）。
对于关系型数据库，系统通过云存储的技术，虚拟一份磁盘空间，提供给关系型数据库使用， 但是这存在局限性：一个表空间或一个表，只能放在实际上是同一个的物理磁盘上，不能将他分配到不同的实际物理磁盘上，否则会严重影响查询比较的性能，所以只能是一个

理论设想。

但是对于 NOSQL，已经有了好多的技术来实现，比如 HBase，BigTable 等，不过这里需要说明的是，如果希望很好的发挥性能，对于云存储的技术要求还是很高，需要云存储提供类似 RAID 的技术，目前来说，有 GFS 和 Hadoop，来实现云存储中文件系统的高可用性。

Bigtable 的原理

Bigtable 是一个比较典型的 Nosql 数据库，文件系统的基础是 Google File System。

每个 Table 都是一个多维的稀疏图 sparse map。Table 由行和列组成，并且每个存储单元 cell 都有一个时间戳。在不同的时间对同一个存储单元 cell 有多份拷贝，这样就可以记录数据的变动情况。在他的例子中，行是 URLs ，列可以定义一个名字，比如：contents。Contents 字段就可以存储文件的数据。或者列名是：”language”，可以存储一个“EN”的语言代码字符串。

为了管理巨大的 Table，把 Table 根据行分割，这些分割后的数据统称为：Tablets。每个 Tablets 大概有 100-200 MB，每个机器存储 100 个左右的 Tablets。底层的架构是：GFS。由于 GFS 是一种分布式的文件系统，采用 Tablets 的机制后，可以获得很好的负载均衡。比如：可以把经常响应 的表移动到其他空闲机器上，然后快速重建。

Tablets 在系统中的存储方式是不可修改的 immutable 的 SSTables，一台机器一个日志文件。当系统的内存满后，系统会压缩一些 Tablets。由于 Jeff 在论述这点的时候说的很快，所以我没有时间把 听到的都记录下来，因此下面是一个大概的说明：

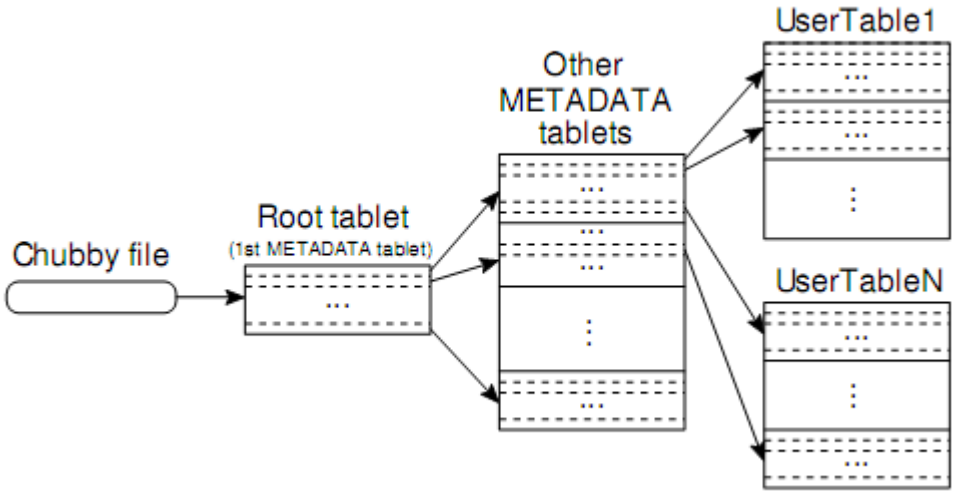
压缩分为：主要和次要的两部分。次要的压缩仅仅包括几个 Tablets，而主要的压缩时关于整个系统的压缩。主压缩有回收硬盘空间的功能。Tablets 的位置实际上是存储在几个特殊的 BigTable 的存储单元 cell 中。看起来这是一个三层的系统。

客户端有一个指向 METAO 的 Tablets 的指针。如果 METAO 的 Tablets 被频繁使用，那个这台机器就会放弃其他的 tablets 专门支持 METAO 这个 Tablets。METAO tablets 保持着所有的 META1 的 tablets 的记录。这些 tablets 中包含着查找 tablets 的实际位置。（老实说翻译到这里，我也不太明白。）在这个 系统中不存在大的瓶颈，因为被频繁调用的数据已经被提前获得并进行了缓存。

现在系统返回到对 列的说明：列是类似下面的形式：family:optional_qualifier。在他的例子中，行：www.search-analysis.com 也许有列：”contents:其中包含 html 页面的代码。“ anchor:cnn.com/news” 中包含着 相对应的 url，”anchor:www.search-analysis.com/” 包含着链接的文字部分。列中包含着类型信息。

注意这里说的是列信息，而不是列类型。列的信息是如下信息，一般是：属性/规则。 比如：保存 n 份数据的拷贝 或者 保存数据 n 天长等等。当 tablets 重新建立的时候，就运用上面的规则，剔出不符合条件的记录。由于设计上的原因，列本身的创建是很容易的，但是跟列相关的功能确实非常复杂的，比如上文提到 的 类型和规则信息等。为了优化读取速度，列的功能被分割然后以组的方式存储在所建索引的机器上。这些被分割后的组作用于 列 ,然后被分割成不同的 SSTables。这种方式可以提高系统的性能，因为小的，

频繁读取的列可以被单独存储，和那些大的不经常访问的列隔离开来。

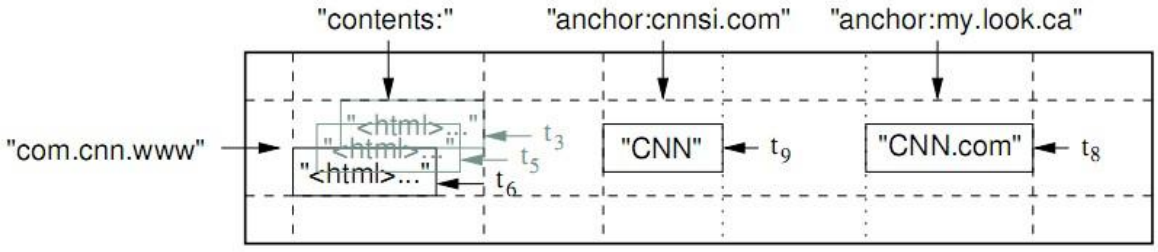


在一台机器上的 所有的 tablets 共享一个 log，在一个包含 1 亿的 tablets 的集群中，这将会导致非常多的文件被打开和写操作。新的 log 块经常被创建，一般是 64M 大小，这个 GFS 的块大小相等。当一个机器 down 掉后，控制机器就会重新发布他的 log 块到其他机器上继续进行处理。这台机器重建 tablets 然后询问控制机器 处理结构的存储位置，然后直接对重建后的数据进行处理。

这个系统中有很多冗余数据，因此在系统中大量使用了压缩技术，压缩前先寻找相似的行，列，和时间 数据。

BMDiff 提供了非常快的写速度： 100MB/s – 1000MB/s 。Zippy 是和 LZW 类似的。Zippy 并不像 LZW 或者 gzip 那样压缩比高，但是他处理速度非常快。

系统举例介绍如下：



一个存储 Web 网页的例子的表的片断。行名是一个反向 URL。contents 列族存放的是网页的内容，anchor 列族存放引用该网页的锚链接文 本（alex 注：如果不知道 HTML 的 Anchor，请 Google 一把）。CNN 的主页被 Sports Illustrater 和 MY-look 的主页引用，因此该行包含了名为"anchor:cnnsi.com"和 "anchhor:my.look.ca"的列。每个锚链接只有一个版本（alex 注：注意时间戳标识了列的版本，t9 和 t8 分别标识了两个锚链接的版本）；而 contents 列则有三个版本，分别由时间戳 t3，t5，和 t6 标识。

行

表中的行关键字可以是任意的字符串（目前支持最大 64KB 的字符串，但是对大多数用户，10-100 个字节就足够了）。对同一个行关键字的读或者写 操作都是原子的（不管读或者写这一行里多少个不同列），这个设计决策能够使用户很容易的理解程序在对同一个行进行并发更新操作时的行为。

Bigtable 通过行关键字的字典顺序来组织数据。表中的每个行都可以动态分区。每个分区叫做一个”Tablet”，Tablet 是数据分布和负 载均衡调整的最小单位。这样做的结果是，当操作只读取行中很少几列的数据时效率很高，通常只需要很少几次机器间的通信即可完成。用户可以通过选择合适的行 关键字，在数据访问时有效利用数据的位置相关性，从而更好的利用这个特性。举例来说，在 Webtable 里，通过反转 URL 中主机名的方式，可以把同一个 域名下的网页聚集起来组织成连续的行。具体来说，系统可以把 maps.google.com/index.html 的数据存放在关键字 com.google.maps/index.html 下。把相同的域中的网页存储在连续的区域可以让基于主机和域名的分析更加有效。

列族

列关键字组成的集合叫做“列族”，列族是访问控制的基本单位。存放在同一列族下的所有数据通常都属于同一个类型（系统可以把同一个列族下的数据压缩 在一起）。列族在使用之前必须先创建，然后才能在列族中任何的列关键字下存放数据；列族创建后，其中的任何一个列关键字下都可以存放数据。根据系统的设计 意图，一张表中的列族不能太多（最多几百个），并且列族在运行期间很少改变。与之相对应的，一张表可以有无限多个列。

列关键字的命名语法如下：列族：限定词。 列族的名字必须是可打印的字符串，而限定词的名字可以是任意的字符串。比如，Webtable 有个列族 language，language 列族用来存放撰 写网页的语言。系统在 language 列族中只使用一个列关键字，用来存放每个网页的语言标识 ID。Webtable 中另一个有用的列族是 anchor； 这个列族的每一个列关键字代表一个锚链接，如图一所示。Anchor 列族的限定词是引用该网页的站点名；Anchor 列族每列的数据项存放的是链接文本。

访问控制、磁盘和内存的使用统计都是在列族层面进行的。在系统的 Webtable 的例子中，上述的控制权限能帮助系统管理不同类型的应用：系统允许 一些应用可以添加新的基本数据、一些应用可以读取基本数据并创建继承的列族、一些应用则只允许浏览数据（甚至可能因为隐私的原因不能浏览所有数据）。

时间戳

在 Bigtable 中，表的每一个数据项都可以包含同一份数据的不同版本；不同版本的数据通过时间戳来索引。Bigtable 时间戳的类型是 64 位 整型。Bigtable 可以给时间戳赋值，用来表示精确到毫秒的“实时”时间；用户程序也可以给时间戳赋值。如果应用程序需要避免数据版本冲突，那么它必 须自己生成具有唯一性的时间戳。数据项中，不同版本的数据按照时间戳倒序排序，即最新的数据排在最前面。

为了减轻多个版本数据的管理负担，系统对每一个列族配有两个设置参数，Bigtable 通过这两个参数可以对废弃版本的数据自动进行垃圾收集。用户可以指定只保存最后 n 个版本的数据，或者只保存“足够新”的版本的数据（比如，只保存最近 7 天的内容写入的数据）。

在 webtable 的举例里，contents:列存储的时间戳信息是网络爬虫抓取一个页面的时间。上面提及的垃圾收集机制可以让系统只保留最近三个版本的网页数据。

对于性能来说

Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups	% in memory	Latency-sensitive?
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Google Analytics	20	29%	10	1	1	0%	Yes
Google Analytics	200	14%	80	1	1	0%	Yes
Google Base	2	31%	10	29	3	15%	Yes
Google Earth	0.5	64%	8	7	2	33%	Yes
Google Earth	70	–	9	8	3	0%	No
Orkut	9	–	0.9	8	5	1%	Yes
Personalized Search	4	47%	6	93	11	5%	Yes

Table 2: Characteristics of a few tables in production use. *Table size* (measured before compression) and *# Cells* indicate approximate sizes. *Compression ratio* is not given for tables that have compression disabled.

一些表存储的是用户相关的数据，另外一些存储的则是用于批处理的数据；这些表在总的大小、 每个数据项的平均大小、从内存中读取的数据的比例、表的 Schema 的复杂程度上都有很大的差别。
经过分析发现：
这类 NOSQL 不需要象关系型数据库一样预先设计 schema，增加或者删除字段非常方便（on the fly）。
2.支持 range 查询：可以对 Key 进行范围查询。
3.高可用，可扩展：单点故障不影响集群服务，可线性扩展。
可以将 NOSQL 的数据模型想象成一个四维或者五维的 Hash。

关系型数据库的文件存储

传统的关系型数据库文件的存储方式为：磁带机/磁盘阵列/Raid 方式。
其中：
磁带机主要是用于做数据文件备份使用，因为速度慢，不能作为实际数据库运行的文件系统，不过具有价格低廉的优点；
磁盘存储阵列：这里说的阵列主要指单独的存储阵列服务器，这类服务器可以接驳多块磁盘，能够满足高速的存储要求，但是成本过高，服务单一；
Raid 的存储方式是指一台服务器具有 2 块或多块磁盘，通过相关程序可以配置为 Raid0...5 等多中存储形式，但是同样存在成本过高，并有单点故障的问题；
云计算服务中存储的方式和技术

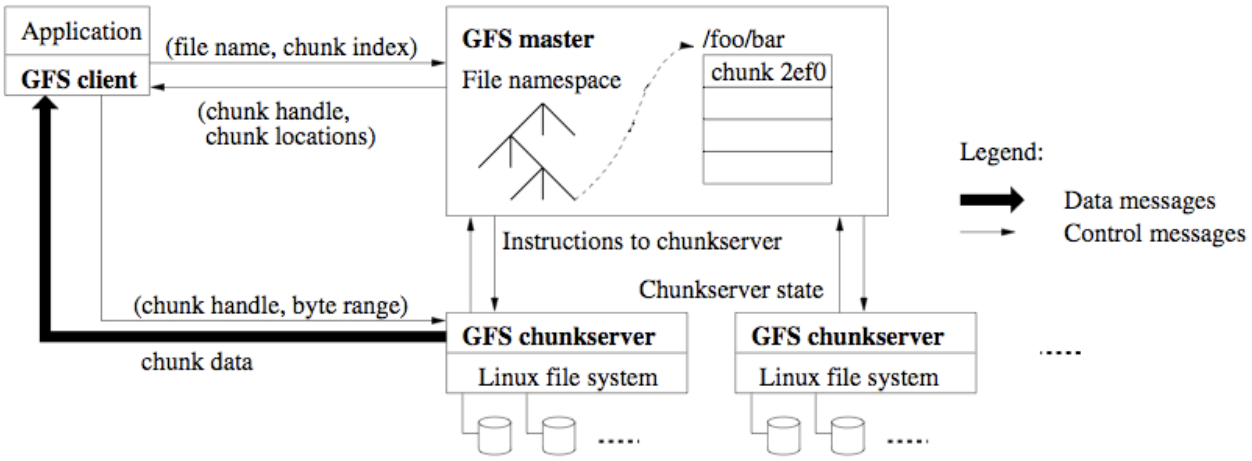
ZFS

依托于云计算的浪潮，云计算不仅包括计算能力云，同样也包含存储能力云。
在早期，还没有云计算概念的时候，SUN 公司就提出了 ZFS 的网络存储文件格式，他同时也是第一个 128 位的文件系统。
ZFS 使用存储池的概念来管理物理存储。以前，文件系统是在单个物理设备的基础上构造的。为了利用多个设备和提供数据冗余性，引入了卷管理器的概念来提供单个设备的映像，以便无需修改文件系统即可利用多个设备。此设计增加了更多复杂性，并最终阻碍了特定文件系统的继续发展，因为这类文件系统无法控制数据在虚拟卷上的物理放置。
ZFS 可完全避免使用卷管理。ZFS 将设备聚集到存储池中，而不是强制要求创建虚拟卷。存储池说明了存储的物理特征（设备布局、数据冗余等），并充当可以从其创建文件系统的

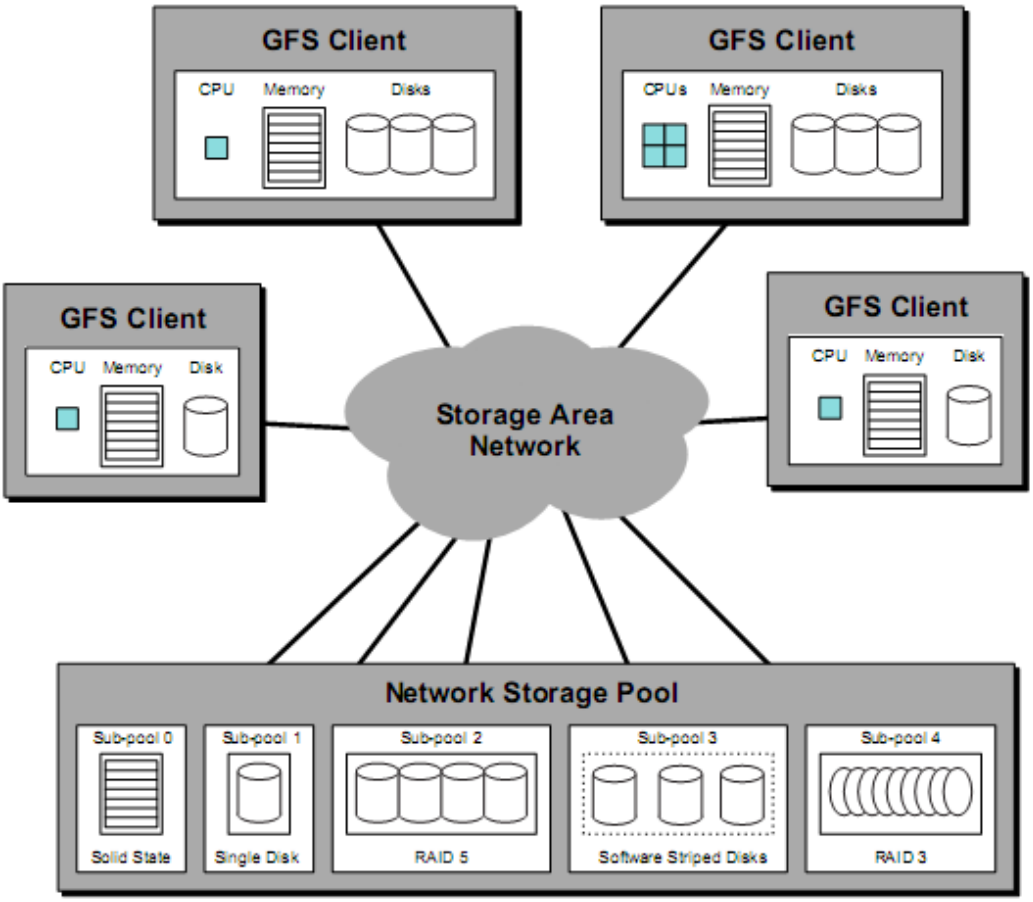
任意数据存储库。 文件系统不再仅限于单个设备，从而可与池中的所有文件系统共享空间。您不再需要预先确定文件系统的大小，因为文件系统会在分配给存储池的空间内自动增长。 添加新存储器后，无需执行其他操作，池中的所有文件系统即可立即使用所增加的空间。在许多方面，存储池都类似于虚拟内存系统。内存 DIMM 添加到系统后，操作系统并不强制您调用某些命令来配置该内存并将其指定给单个进程。系统中的所有进程都会自动使用所增加的内存。
现在，经过技术人员的努力，已经可以成功在 ZFS 上面运行 MYSQL 等数据库。

GFS

GFS 是 Google 公司开发的网络文件系统，主要用来满足 Google 的应用中出现的数据大量增加的情况。
目前 GFS 不是开源的，但是已经有人基于该类设计思想开发相关产品。
GFS 原理如下：



GFS 服务包括 GFS 主服务器和 GFS 块服务器，都运行于 Linux 系统上，并同时运行一个服务程序提供访问。
当文件被存储时，会被分为固定大小的块，并由一个 64 位的标识来作为这个块的句柄，这样，当需要读取的时候，主服务器会根据块的句柄和读取范围，确定块文件的位置；在保存的时候，为了提高系统的高可用性，同一个块会被复制到多个块服务器上。
对于 GFS 我们可以通过一个简单的结构图进行说明

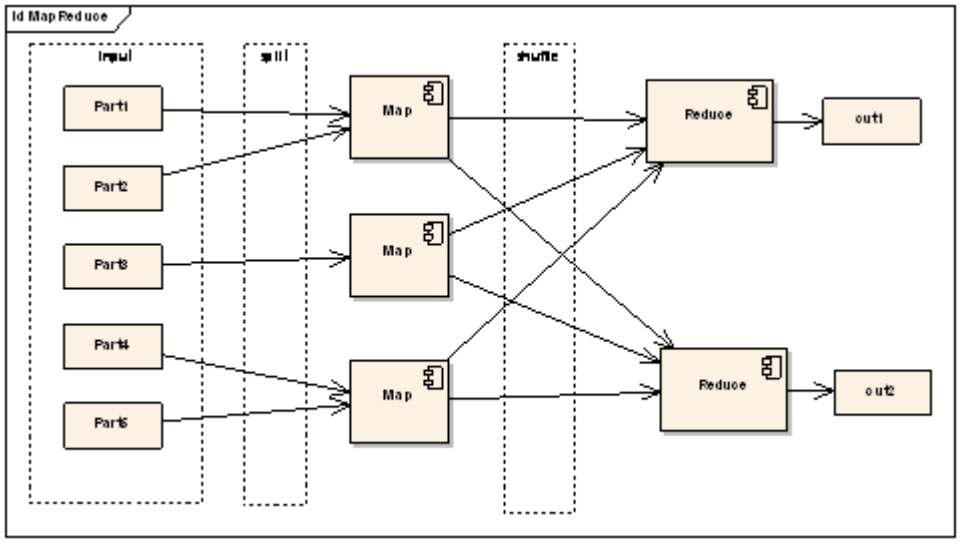


可以看到，对于这类分布式的存储，GFS 是通过网络存储池的方式来进行获得，而且对于存储的设备并无很大的限制。

Hadoop

Hadoop 是一个开源的类似 GFS 的实现，也是目前主流的分布式文件系统，Facebook，Amazon，Yahoo 即采用这个文件系统对客户提供服务。

Hadoop 框架中最核心的设计就是：MapReduce 和 HDFS。MapReduce 的思想是由 Google 的一篇论文所提及而被广为流传的，简单的 一句话解释 MapReduce 就是"任务的分解与结果的汇总"。HDFS 是 Hadoop 分布式文件系统（Hadoop Distributed File System）的缩写，为分布式计算存储提供了底层支持。

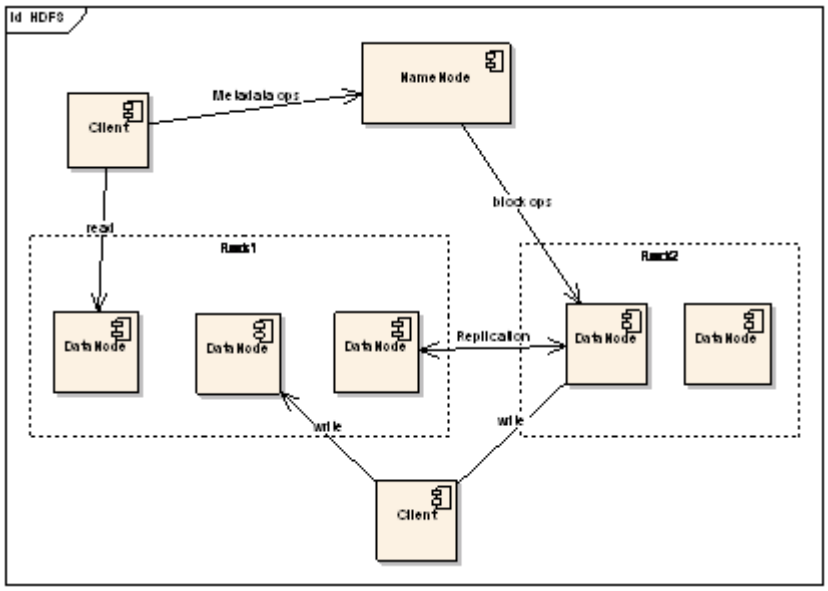


MapReduce 结构示意图

在 Map 前还可能会对输入的数据有 Split（分割）的过程，保证任务并行效率，在 Map 之后还会有 Shuffle（混合）的过程，对于提高 Reduce 的效率以及减小数据传输的压力有很大的帮助。后面会具体提及这些部分的细节。

HDFS 是分布式计算的存储基石，Hadoop 分布式文件系统基本的几个特点：

1. 对于整个集群有单一的命名空间。
2. 数据一致性。适合一次写入多次读取的模型，客户端在文件没有被成功创建之前无法看到文件存在。
3. 文件会被分割成多个文件块，每个文件块被分配存储到数据节点上，而且根据配置会由复制文件块来保证数据的安全性。



HDFS 结构示意图

这里对 HDFS 三个重要角色进行介绍：NameNode、DataNode 和 Client。NameNode 可以看作是分布式文件系统中的管理者，主要负责管理文 件系统的命名空间、集群配置信息和存储块的复制等。NameNode 会将文件系统的 Meta-data 存储在内存中，这些信息主要包括了文件信息、每一个 文件对应的文件块的信息和每一个文件块在 DataNode 的信息等。DataNode 是文件存储的基本单元，它将 Block 存储在本地文件系统中，保存了 Block 的 Meta-data，同时周期性地将所有存在的 Block 信息发送给 NameNode。Client 就是需要获取分布式文件系统文件的应用程 序。这里通过三个操作来说明他们之间的交互关系。

文件写入：

1. Client 向 NameNode 发起文件写入的请求。
2. NameNode 根据文件大小和文件块配置情况，返回给 Client 它所管理部分 DataNode 的信息。
3. Client 将文件划分为多个 Block，根据 DataNode 的地址信息，按顺序写入到每一个 DataNode 块中。

文件读取：

1. Client 向 NameNode 发起文件读取的请求。
2. NameNode 返回文件存储的 DataNode 的信息。
3. Client 读取文件信息。

文件 Block 复制：

1. NameNode 发现部分文件的 Block 不符合最小复制数或者部分 DataNode 失效。
2. 通知 DataNode 相互复制 Block。
3. DataNode 开始直接相互复制。

如何将云存储同数据库进行结合

1．关系型数据库和云存储的结合

这里我们根据 Mysql 数据库和 ZFS 系统的结合列举一下一些简单的关键步骤：

- 使用 Solaris 或 OpenSolaris，并搭建 ZFS 系统；
- 根据 MySQL 存处引擎数据快的大小，调整 ZFS 的数据块的大小：# zfs set recordsize=16K matrix/mysqldata
- 给日志分一个单独的存储池
- 关掉 Innodb 的双重写入功能
- 限制内存的使用：set zfs:zfs_arc_max = 0x200000000
- 对于写比较多的 MyISAM，可以采用分开的 ZIL（ZFS Intent Log）

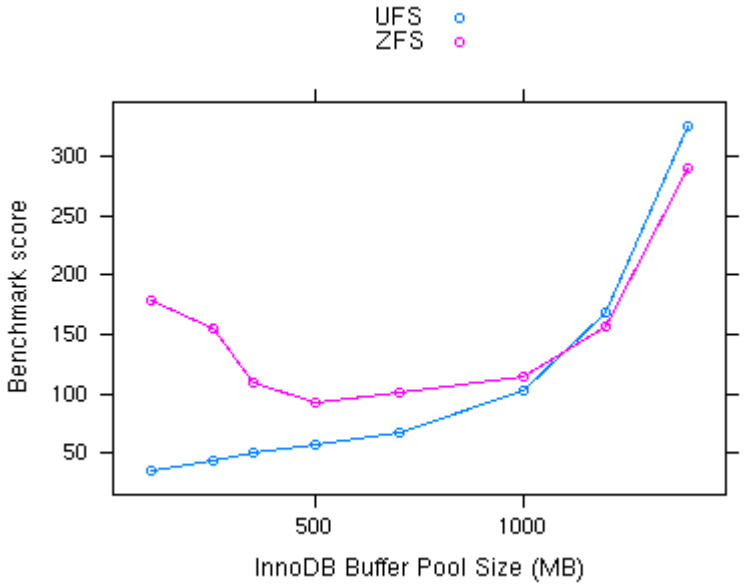
这样就将 MySql 和 ZFS 进行简单的结合。

同时对相关性能进行测试后，对比 UFS（Unix 的一种文件系统）如下：

InnoDB Buffer Pool Size	UFS Direct I/O	ZFS
100 MB	36.84	96.21
250 MB	45.70	80.69
350 MB	50.68	102.75
500 MB	57.32	94.99
700 MB	61.92	85.95
1000 MB	82.41	98.35
1200 MB	95.01	88.98
1400 MB	101.32	117.83

ZFS vs. UFS after 10 min. warm-up.

InnoDB Buffer Pool Size	UFS Direct I/O	ZFS
100 MB	35.71	179.18
250 MB	44.25	154.26
350 MB	50.60	110.14
500 MB	57.10	93.29
700 MB	66.57	101.35
1000 MB	103.47	114.18
1200 MB	168.66	156.04
1400 MB	325.05	290.14



可以看到，通过使用分布式文件系统，数据库的性能得到了很大程度的提高。

2. NoSQL 和云存储的结合

典型的就是 GFS 和 BigTable，Hadoop 和 HBase：

我们这里进行一个简单的整合测试

1:单线程 hbase 的文件存入

```
        String parentPath = "F:/pic/2003-zhujiajian";
        File[] files = getAllFilePath(parentPath);

        HBaseConfiguration config = new HBaseConfiguration();
        HTable table = new HTable(config, new Text("offer"));
        long start = System.currentTimeMillis();
        for (File file :files) {
            if(file.isFile()) {
                byte[] data = getData(file);
                createRecore(table,file.getName(),"image_big",data);
            }
        }
        long end = System.currentTimeMillis();
        System.out.println("time cost=" + (end-start));
```

输出：

108037206 bytes, 303 个 files write from local windows to remote hbase,cost 23328 or 21001 milliseconds

2:单线程 hadoop 的文件存入

```
Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(conf);
        Path src = new Path("F:/pic/2003-zhujiajian");
        Path dst = new Path("/user/zxf/image");
        long start = System.currentTimeMillis();
```

```
        fs.copyFromLocalFile(src, dst);
        long end = System.currentTimeMillis();
        System.out.println("time cost=" + (end-start));
```

输出:

108037206 bytes, 303 files write from local windows to remote hdfs,cost 26531 or 32407 milliseconds

3:单线程 hbase 的文件读取

花费的时间慢的难以置信

108037206 bytes, 303 files read from hdfs to local cost 479350 milliseconds

4:单线程 hadoop 的文件读取

108037206 bytes, 303 files read from hdfs to local cost 14188 milliseconds

5:深入测试

取几个文件对比

```
fileSize(byte)  hdfs time(ms) hbase time(ms)
 12341140        1313          14688
 708474          63            4359
 82535           15            3907
 55296           16            125
```

结论

我们可以看到，将数据库同云存储服务结合后，数据库提供的性能大幅提高，并提高了数据库的可用性和数据的安全性。
目前，大家对于云计算同数据库在技术上的结合投入约来越多，技术也不断得到积累，我们可以采用开源的 Hadoop+HBase 或 Cassandra 来搭建具有云数据库功能的架构。
同时，我们也可以将其他的比如 Memcache 内存数据库通过扩展变为简单的云数据库。
这类技术的采用会高效的利用服务器的硬件资源，减少污染，能够更快的相应访问信息。

http://docs.sun.com/app/docs/doc/819-7065/zfsover-2?l=zh&a=view
http://www.codechina.org/doc/google/gfs-paper/storage.html
http://www.google.com
http://dev.mysql.com/tech-resources/articles/mysql-zfs.html
http://hadoop.apache.org
http://www.meichua.com/archives/62.html