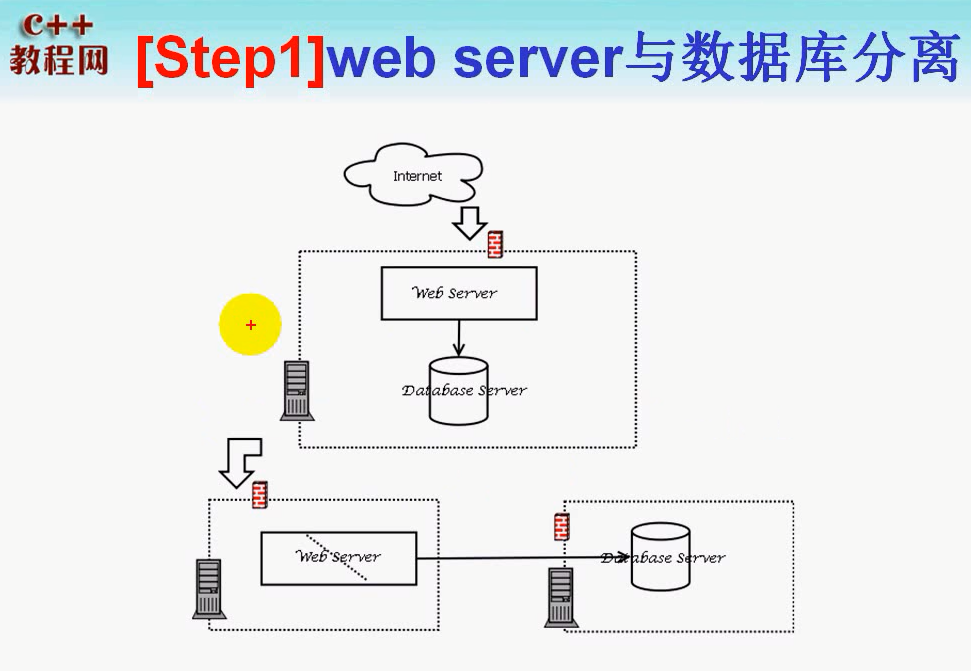


大型网站只是大并发服务器的一个典型事例

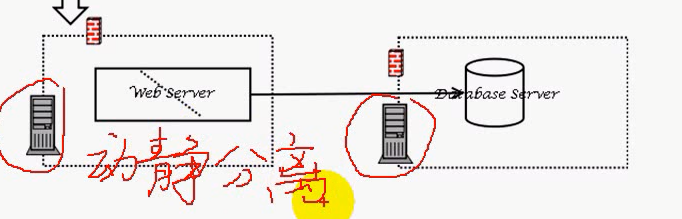


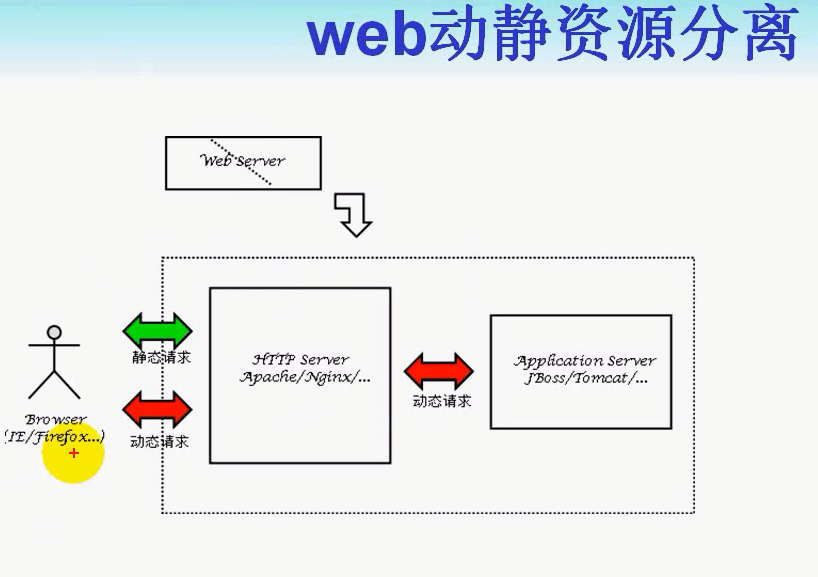
Web服务器包含http服务器与应用服务器

开始 访问量小 web server和db server部署在一起 ---- 但是 两者互相影响 一旦有一个有问题 就会影响另一个

所以 web服务器和数据库服务器分开部署

Web服务器还可以做一个动静资源分离

 这两种架构都很简单 但是都不满足高可用目标 ---- 全是单点



浏览器的请求 动态资源请求 和 静态资源请求

静态资源请求 html js css img

动态资源请求 jsp php

所以 web server 分为http服务器（处理静态请求）和应用服务器（处理动态请求）

主流的http服务器：Apache nginx ---- 也可以处理php动态请求 但是 不能处理jsp动态请求

应用服务器 jboss tomcat

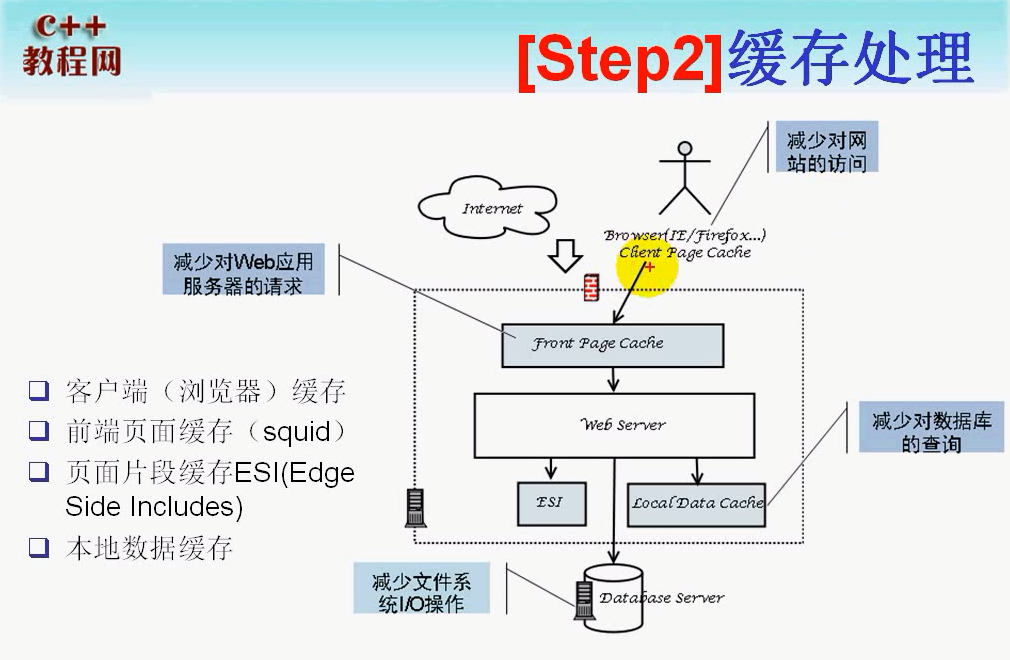
【图中标示的分别是 Http Server和Application Server】

Http服务器又称为前端服务器

应用服务器又称为后端服务器 主要处理实际系统的应用逻辑和商业逻辑

随着网站的访问量提高 网站变慢了 --- 这时候可以采用缓存处理

\*\*浏览器这边做一个缓存 减少对网站的访问



在前端服务器这边 做一些静态页面缓存 ---- Front Page Cache

---- 减少对应用服务器的请求

动态页面相对静态的部分 使用ESI做页面片段缓存

数据库的数据做本地数据缓存

----- 如果使用文件系统 那么就会频繁对文件系统IO操作 这样 用数据库减少IO

\*\*网站访问量继续增加 所以 提高速度

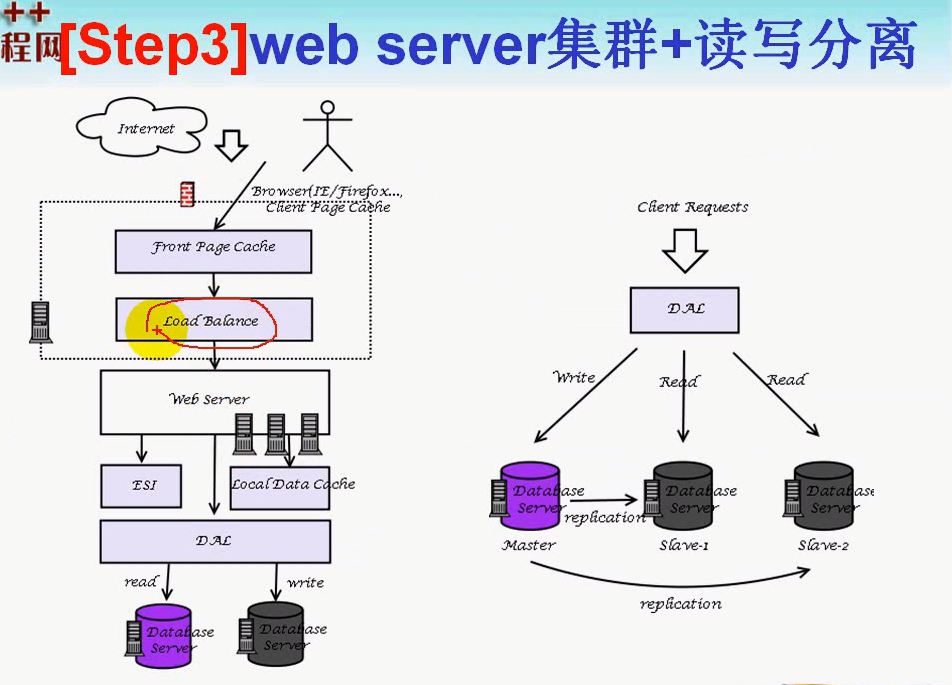
web应用服务器增加了 ---- 这时候要做负载均衡

**前端服务器的负载均衡**

**后端的应用服务器也需要做负载均衡**

**这样 数据库的瓶颈就会出现 ---- DB做读写分离**

**一般读操作比写操作频繁 ---- 读数据库比写的数据库多分配几个 --- 使用replication机制 对数据库的数据进行同步 因为读写分离了**

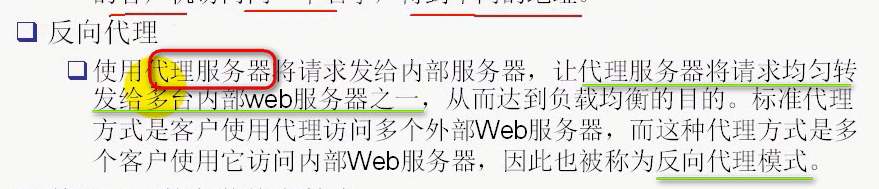


主流的数据库都有类似的机制 主从服务器的结构 ---- 这样又达到数据库服务器的负载均衡 --- 写的均衡到写库 读的均衡到读库

负载均衡







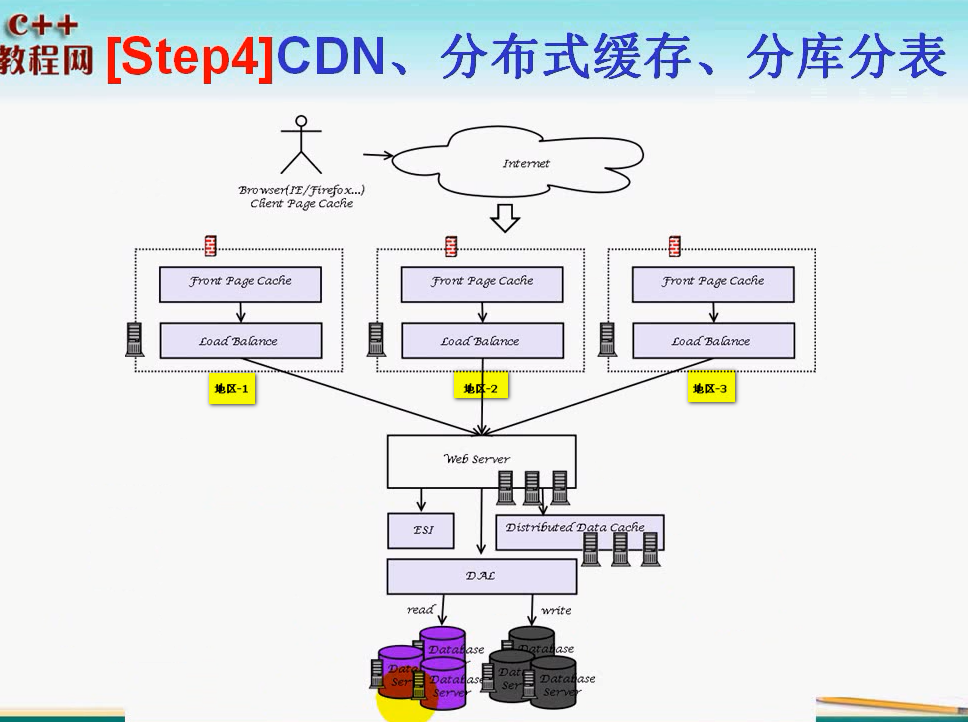
 外部访问 进入到NAT服务器 NAT的地址映射 映射到内部web服务器 和反向代理服务器类似



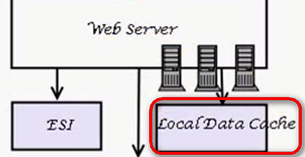
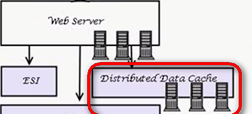
 ---- 使用任务服务器 把任务发给负载小的服务器

CDN ---- 内容分发网络 --- 是在各个运营商 各个地区增加内容分发网络 CDN的汉语意思 ---- 不同的地区访问网站 速度都得到提升 不同的运营商接入都得到了提升

【CDN就是把请求 从当前一个区域 拓展到全国各地 全世界各地】

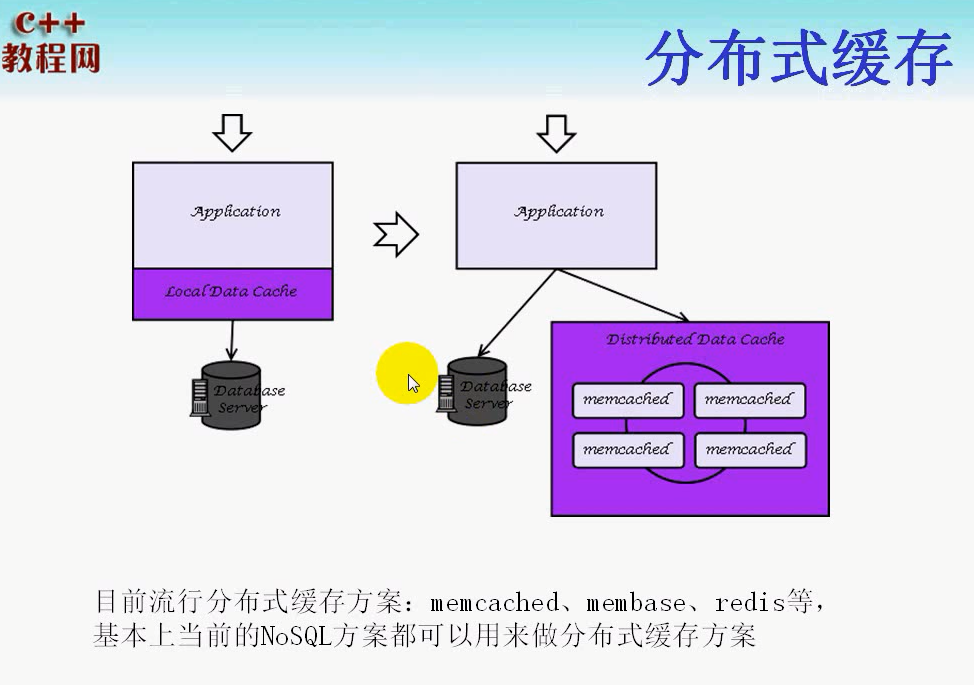


刚才的 Local Data Cache 本地数据缓存变成了 分布式数据缓存 Distributed Data Cache

 -🡪 

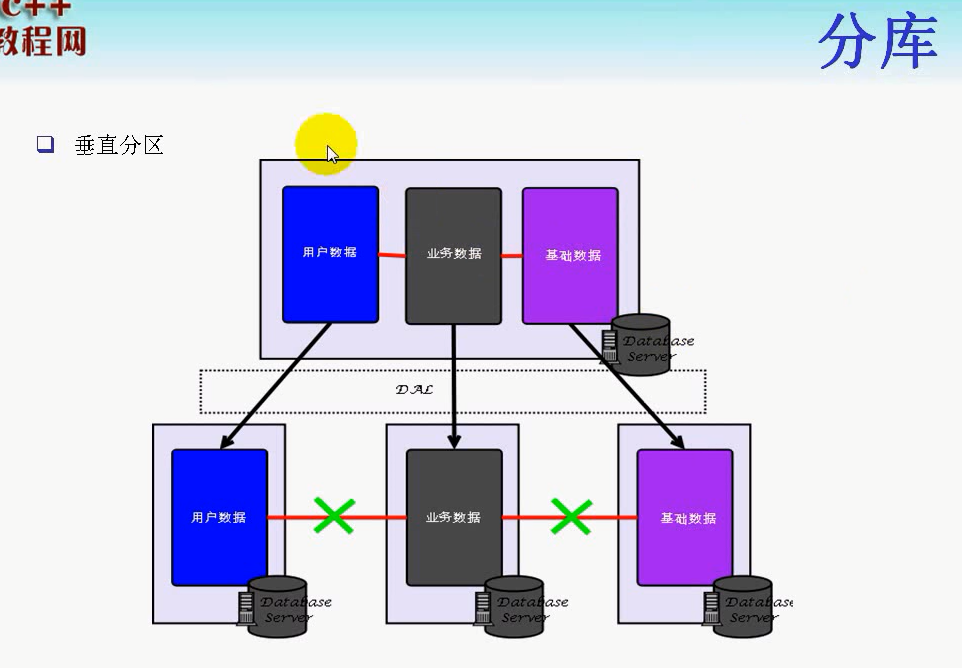
分布式缓存对所有的主机都是共享的

然后对数据库进行分库分表 --- 因为数据库的数据量比较大 在并发量大的时候 对数据库进行读写操作就会出现**锁竞争** ---- 这样数据库进行分库 分到更多的主机之上 避免所竞争  分库之后 数据库中的表还会有大表 --- 可以大表分成小表 更大的程度避免锁竞争



左边是本地数据缓存 右边是分布式数据缓存

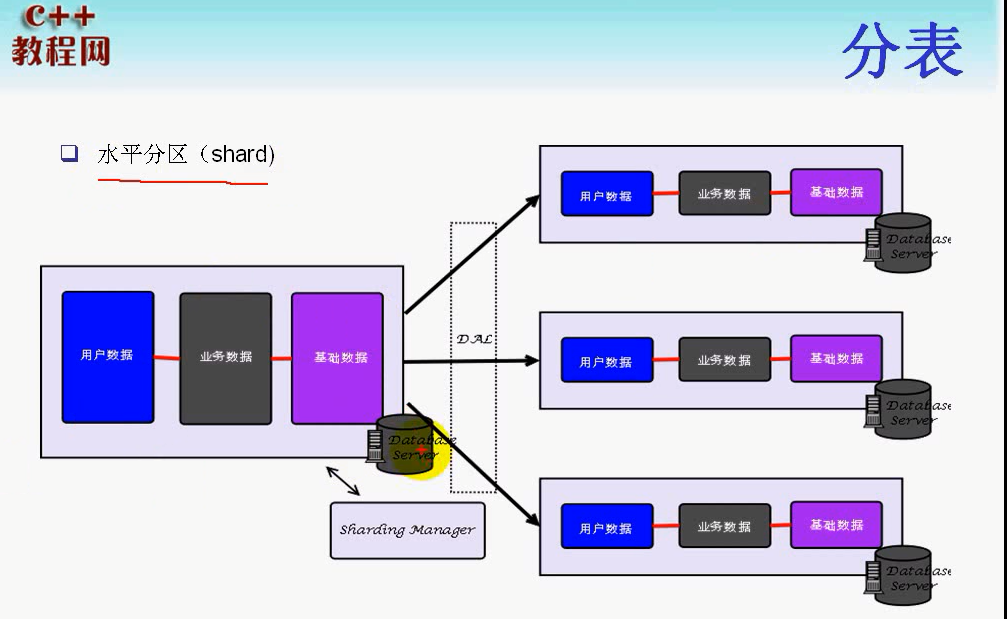
分库

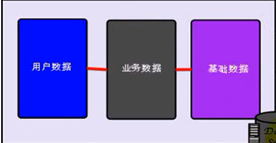
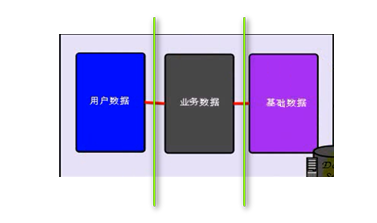


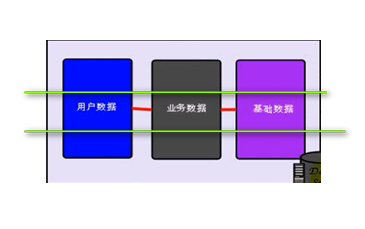
把一个数据库服务器 分成三个数据库服务器 减少数据库在访问数据时候 造成的锁竞争

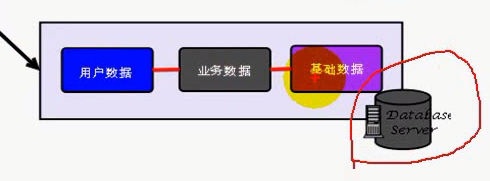
垂直分区之后 每一个数据库中可能有大表

然后水平分区

 【分片shard 也叫水平分区】

【 垂直切分 这个切的方向就是垂直的 所以叫垂直切分

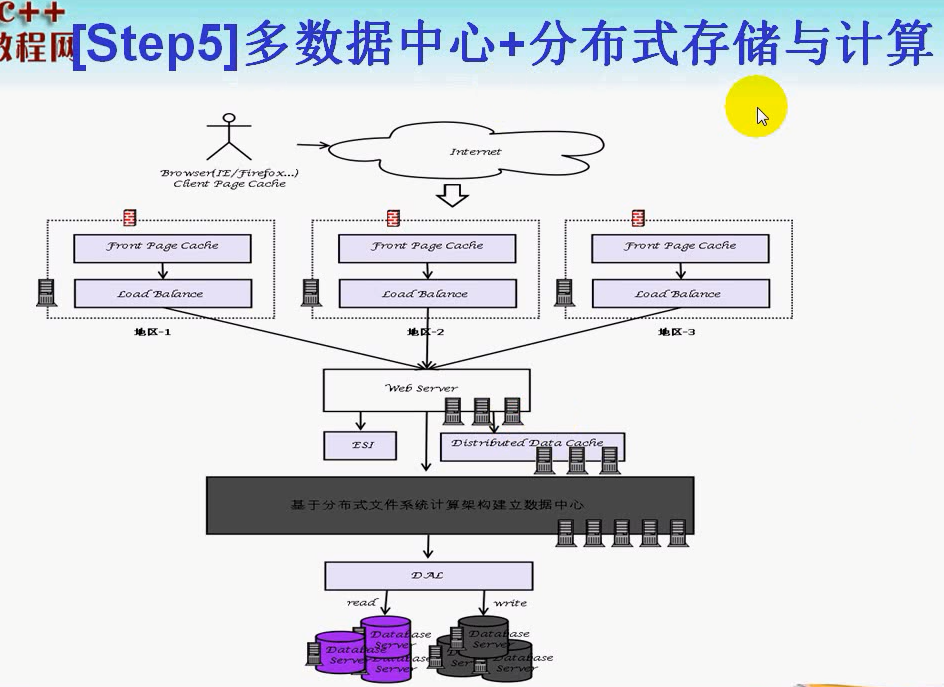
横着切 这个就是水平切分】

水平切分 每个数据库的总和就是总的数据库的数据

User表1w条记录 分摊到3个数据库

--- Hash算法 由于数据分摊到多个数据库 访问数据库的时候 要增加一个DAL数据访问层 通过DAL到指定的数据库中获取 DAL编写的难度增加了 ------

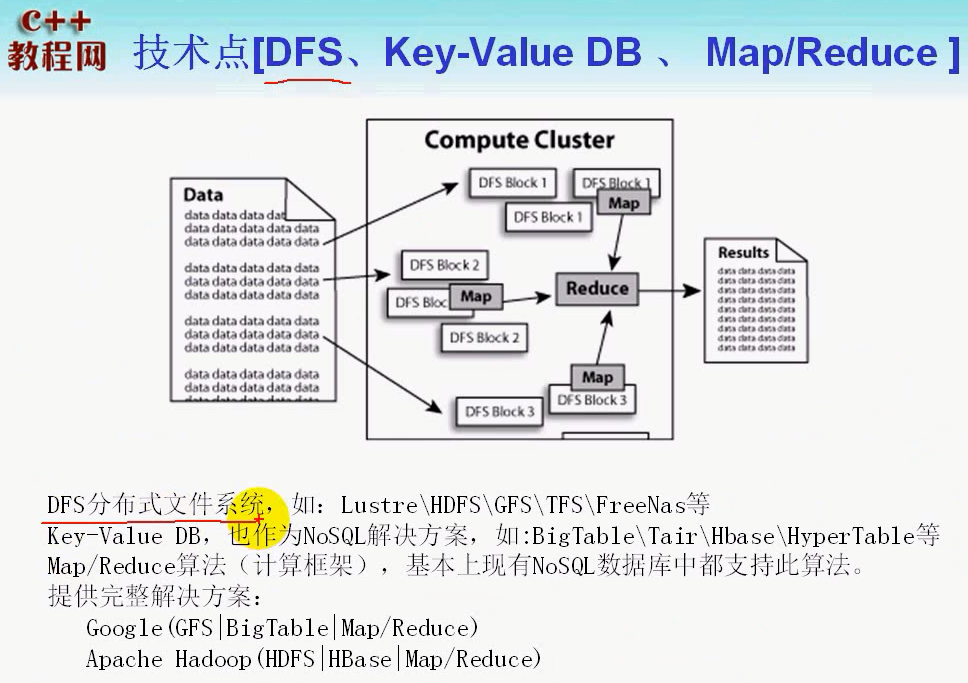
还可以既分库又分表





多了一个这个层面 ---- 主要原因在于 目前的大型网站 有些数据对一致性的要求不高 ---这样的数据 没必要不用保存在关系型数据库中 --- 关系型数据库的事务处理 和大表的链接 join 都是服务器的性能杀手 目前大型的网站 数据库服务器中的数据量是PB >TB 可以把关系数据库中 对于一致性要求低的数据 分离开来 ---这样就为DB减肥 放到数据中心 --- 数据中心使用NoSQL进行存储 --- 基于KV存储 一致性要求低 但是并发性高 访问NoSQL 能够达到更大的并发 但是 访问关系型数据库的数据 一定不如NoSQL的并发性来得高

——NoSQL可以用来处理一致性不是很高的 并发性高的场合 关系型数据库对事物要求高 锁的操作 遇到锁 就是服务器的性能杀手 对数据库的并发性能有较大的影响



NoSQL中的数据 可能存在DFS中 ---- 为什么要有自己的文件系统呢？

比如淘宝网站 --- 淘宝的商品有很多的图片 **这些图片可能比较小** 图片存放在OS的文件磁盘上 用OS的文件系统管理 操作系统的**磁盘的块**的大小都比较小 如果查找磁盘上的图片 就会在磁头来回进行转动 ---- 效率比较低

这个时候基于自己的文件系统进行存储 原理很简单 每个块可以增大 64M为一个块 将文件存储到块中 相关性大的存储到一个块中 另外的文件存储到另外的块 ---- 这样OS看到的文件比较大 ---- 存储的时候 先找到大块 – 这样不是小文件 散落在磁盘上面 --- 每个块不仅存放了实际的图片 还有图片存放在哪的信息 ------ **通过一定的hash算法** 就可以快速找到文件 不至于在磁盘上 来回转动磁头来找到文件【比如 NameNode DataNode就是这样】 ------ 这些NoSQL基于KeyValue存储 为了快速找到这些数据 这些NoSQL建立在分布式的文件系统之上 ------ 减少OS的IO

基于分布式的文件系统 NoSQL就是基于KV的数据库

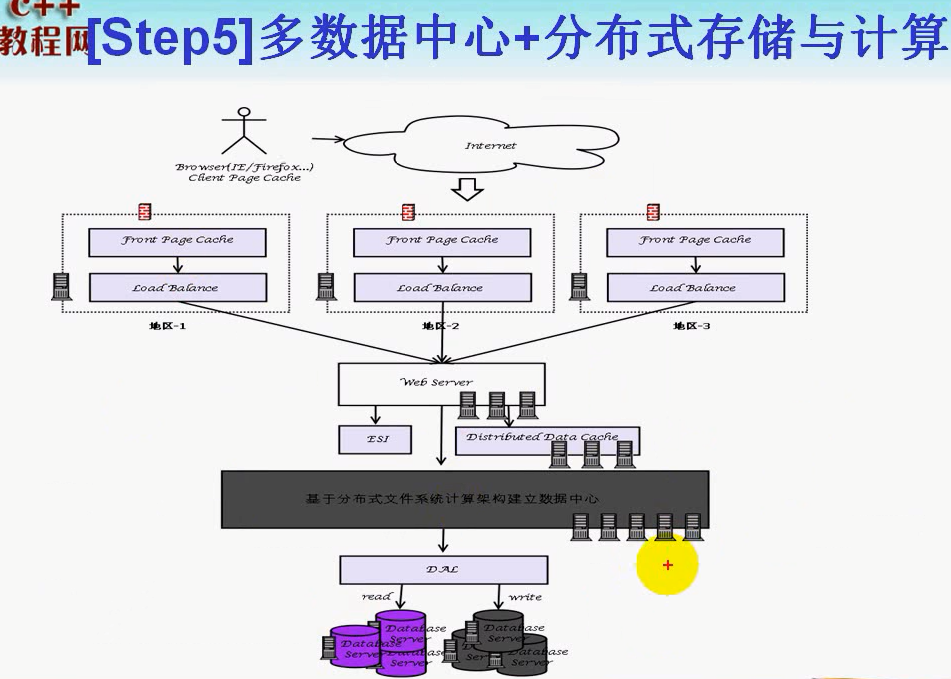
还有一个是MR ---分布式的计算框架

 这三个是Google的核心技术 成为GFS



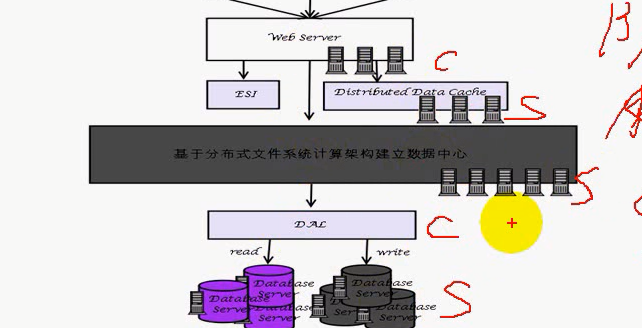
这就是架构演变过程

最终的大型网站架构的架构图



虽然是BS模式的架构 本质实际上是CS架构 这里面的服务器 web服务器访问缓存 ---- web服务器看成了C 缓存看成S

本质都是CS

 这些CS组件的编程 都可以看做服务器编程 Socket编程 --- 实现的时候 可以用java 也可以用c++

应用服务器也可以Java 也可以C++