# 数据库缓存层的优化

**什么是数据库缓存**

**为什么要使用缓存**

**使用MySQL查询缓存**

**使用Redis缓存**

**什么是数据库缓存**

MySQL等一下常见的关系型数据库的数据都存储在磁盘当中，在高并发场景下，业务应用对MySQL产生的增、删、改、查的操作造成巨大的I/O开销和查询压力、这无疑对数据库和服务器都是一种巨大的压力，为了解决此类问题，缓存数据的概念应运而生。

极大的解决数据库服务器的压力

提高应用数据的响应速度

常见的缓存形式：内存缓存、文件缓存

我们本节重点讨论内存缓存

缓存数据是为了让客户端很少甚至不访问数据库服务器进行数据的查询、高并发下、能最大程度的降低对数据库服务器的访问压力。

用户请求-->数据查询-->连接数据库服务器并查询数据-->将数据缓存起来(HTML、内存、JSON、序列化数据)-->显示给客户端

用户再次请求或者新用户访问-->数据查询-->直接从缓存中获取数据-->显示给客户端。

**缓存方式的选择**

**缓存常见的选择**

**缓存数据的实时性**

**缓存数据的隐定性**

启用MySQL查询缓存

极大地降低CPU使用率

Query\_cache\_type

查询缓存类型、有0、1、2三各取值。0则不使用查询缓存。1表示始终使用查询缓存。2表示按需使用查询缓存。

Query\_cache\_type为1时、亦可关闭查询缓存

SELECT SQL\_NO\_CACHE \* FROM my\_table WHERE condition;

Query\_cache\_size

默认情况下query\_cache\_size为0、表示为查询缓存预留的内存为0、则无法使用查询缓存

SET GLOBAL query\_cache\_size=134217728;

查询缓存可以看做是SQL文本和查询结果的映射

第二次查询的SQL和第一次查询的SQL完全相同、则会使用缓存SHOW STATUS LIKE 'Qcache\_hits'；查看命中次数表结构或数据发生改变时，查询缓存中的数据不再有效

**清理缓存**

FLUSH QUERY CACHE;//清理查询缓存内存碎片

RESET QUERY CACHE;//从查询缓存中移除出所有查询

FLUSH TABLES; //关闭所有打开的表，同时该操作将会清空查询缓存中的内容

**使用Memcache缓存查询数据**

对于大型站点，如果没有中间缓存层，当流量打入数据库层时，即便有之前的几层为我们挡住一部分流量，但是在大并发的情况下，还是会有大量请求涌入数据库层，这样对于数据库服务器的压力冲击很大，响应速度也会下降，因此添加中间缓存层很有必要。

memcache 是一套分布式的高速缓存系统，由LiveJournal的Brad Fitzpatrick开发，但目前被许多网站使用以提升网站的访问速度，尤其对于一些大型的、需要频繁访问数据库的网站访问速度提升效果十分显著。

**工作原理**

Memcache是一个高性能的分布式的内存对象缓存系统，通过在内存里维护一个统一的巨大的hash表，它能够用来存储各种格式的数据，包括图像、视频、文件以及数据库检索的结果等。简单的说就是将数据调用到内存，然后从内存中读取，从而大大提高读取速度。

**工作流程**

先检查客户端的请求数据是否在memcached中、如有、直接把请求数据返回、不再对数据库进行任何操作；如果请求的数据不在memcached中，就去查数据库、把从数据库中获取的数据返回给客户端、同时把数据缓存一份到memcached中。

**方法**

获取:get(key)

设置：set(key,val,expire)

删除：delete(key)

**通用缓存机制**

用查询的方法名+参数作为查询时的key value对中的key值使用Redis缓存查询数据

#### 使用Redis缓存查询数据

**与Memcache的区别**

性能相差不大，如果是持久化就使用redis如果使用，复杂的数据结构就使用memcache;

Redis在2.0版本后增加了自己的VM特性，突破物理内存的限制，Memcache可以修改最大可用内存，采用LRU算法

与Memcache的区别Redis，依赖客户端来实现分布式读写

Memcache本身没有数据冗余机制

Redis支持（快照、AOF）,依赖快照进行持久化，aof增强了可靠性的同时，对性能有所影响

Memcache不支持持久化，通常做缓存，提升性能；

Memcache在并发场景下，用cas保证一致性、redis事务支持比较弱，只能保证事务中的每个操作连续执行

Redis支持多种类型的数据类型

Redis用于数据量较小的高性能操作和运算上

Memcache用于在动态系统中减少数据负载，提升性能；适合做缓存，提高性能

**其他数据**

**Session**

**Session\_set\_save\_handler**

# MySQL数据库层的优化

**优化方向**

**优化方案**

**数据表数据类型优化**

**索引优化**

**SQL语句的优化**

**存储引擎的优化**

**数据表结构设计的优化**

**数据库服务器架构的优化**

**数据表数据类型优化**

字段使用什么样的数据类型更合适

字段使用什么样的数据类型性能更快

Tinyint、smallint、bigint

Tinyint:范围是0到255，适合如人的年龄之类的

Smallint：范围几百到几千，适合古建筑，或是古树的年龄

Biging:是上千万或是上亿

Char:适合手机号码，md5加密密码,等固定长度的数据

Varchar:存储字符串长度不固定的数据

Enum:特定，固定的分类可以使用enum存储，效率更快

IP地址的存储：使用整型存储IP地址

**建立合适的索引**

索引的创建原则

索引不是越多越好，在合适的字段上创建合适的索引。在where后面有列的字段上创建索引更合适

复合索引的前缀原则

Like查询%的问题

全表扫描优化

比如在一张表上面有100条记录，然后我查询的时候我要查询其中的98条，本身来说我们的mysql的优化器就会放弃索引的使用因为我们在进行显示查询每一条数据的时候如果你使用索引的化，就是先去看索引在去看数据先去看索引在去看数据89条的化就得去看98遍索引所以本身来说我们全表扫描会比看索引更快所以这个地方我们就可以直接去全表扫描。

or条件索引使用情况

or前的这个列如果有索引后面这个列如果没有索引它就用不到索引

字符串类型索引失效的问题

对于字符串类型的话我们需要注意当我们进行查询的时候如果字符串里面存的是一个整形然后我进行查询的时候不加引号直接用整型来查这时索引会失效。如果我们要查的话尽量用咱们的引号。

**SQL语句的优化**

优化查询过程中的数据访问

使用limit

返回列不用\*

优化长难句的查询语句

变复杂为简单

切分查询

比如现在要一次性删除一千万的数据，那如果这样删做的话对mysql的压力很大，那怎么办，那我们就每次都删一百万。然后去做一个停顿做一个休眠，暂停多长时间以后在去做一个删除。

分解关联查询

将关联查询分解开，分成多个查询，这样有利于我们去做数据层面的缓存。

**优化特定类型的查询语句**

**优化count()**

,没有where条件的count(\*)是最快的，如果有where条件可以使用近似值

**优化关联查询**

可以针对关联查询语句的一些left join 、right join针对它的一些字段创建索引，针对性的使用特定的关联查询方式来进行操作，然后就是优化子查询，但是不建议使用子查询我们可以使用关联查询类替代之查询。

优化grouP by和distinct

优化limit和union

**存储引擎优化**

尽量使用InnoDB存储引擎

**数据表结构设计的优化**

分区优化

通过特定的策略对数据表进行物理拆分，比如一张表它会分出来

多个文件当然对于这多个文件来说我们是根据不同的策略比如说你根据不同的地域。比如北京一个区，上海一个区广州一个区，也可以是热点数据一个区不常用数据一个区

一张表的分区可以使用partition by来进行操作。但是对于我们的分区来说，我们如果说对现有表要做分区的话需要区修改表结构

分库分表

水平拆分

所谓水平拆分就是把我们的一些记录按照行级来进行拆分，比如说我们现在有一亿条记录。这一亿条数据里面我们其中有一百条是活跃数据其他的都是不活跃的，那这样的话我就可以把这一百条数据给它拆出来，拆到另外一张表里面去。然后当我去请求我们的这个活跃数据的查询的时候我只要去请求我们这个活跃的这张表就可以了，它只有一百条数据查询起来会非常的快。而如果是一亿条的话本身对我们的mysql来说就是一个非常大的难度。

垂直拆分

而对于垂直拆分来说我们可以把一些常用的列拆分出来，不常用的列拆分到另外的表里面去，当然对于垂直拆分来说我们会有一定的问题，什么样的问题啊！如果说我们要查询常用的列和不常用的列我都要显示怎么办？这时我们是不是通过join来进行连接对不对，而水平拆分我们要想把这个活跃数据和不活跃数据要显示我们是不是要通过unin来进行连接所以在这的话我们要对我们的分库分表来说呢

**数据库架构的优化**

**主从复制**

**读写分离**

**双主热备**

**负载均衡**

通过LVS的三种基本模式失效负载均衡

MyCat数据库中间件实现负载均衡

# Web服务器的负载均衡、请求分发

七层负载均衡的实现

基于URL等应用层信息的负载均衡

Nginx的proxy是它一个很强大的功能，实现了7层负载均衡

**七层负载均衡**

功能强大、性能卓越、运行稳定

配置简单灵活

能够自动剔除工作部正常的后端服务器

上传文件使用异步模式

支持多种分配策略，可以分配权重，分配方式灵活

**Nginx负载均衡**

内置策略、扩展策略

内置策略：IP Hash、加权轮询

扩展策略：fair策略、通用hash、一致性hash

**加权轮询策略**

首先将请求都分给高权重的机器、直到该机器的权值降到了比其他机器低、才开始将请求分给下一个高权重的机器。

当然有后端机器都down掉时，Nginx会立即将所有机器的标志位清城初始状态，以避免造成所有的机器都处在timeout的状态。

**IP Hash策略**

Nginx内置的另一个负载均衡策略，流程和轮询很类似，只是其中的算法和具体的策略有些变化

IP Hash算法是一种变相的轮询算法

**fair策略**

根据后端服务器的响应时间判断负载情况，从中选出负载最轻的机器行分流。

**通用Hash、一致性Hash策略**

通用hash比较简单，可以以Nginx内置的变量为key进行hash、一致性hash采用Nginx内置的一致性hash环，支持memcache

**Nginx配置**

四层负载均衡的实现

通过报文中的目标地址和端口，再加上负载均衡设备设置的服务器选择方式，决定最终选择的内部服务器。

LVS实现服务器集群负载均衡有三种方式、NAT、DR和TUN