链滴

存储（八）—— Partition

作者：xinzhongtianxia
原文链接：https://ld246.com/article/1573205890650
来源网站：链滴

如果partition这个词听起来不是那么亲切，说起其他名字，想必你会更熟悉一些，比如，ES里面的shad，HBase里面的region，mysql里面的分库分表，kafka里面的。。额。。kafka里面就叫做partitio。partition（数据分片）是一种更加广泛的称呼。

partition和replication

上一节讲到的replication解决的是数据备份、高可用、更靠近用户的问题，有了replication之后，数更靠近用户了，性能上的提升，主要体现在网络的传输长度上——同样的数据从北京传到上海肯定比北京传到广州快。

那么，把数据放到网络上之前呢？先得把它（要传输的数据）从一大堆数据（数据库）中找出来吧。么，同样的存储方式，从1亿条数据中查找快还是从1万条数据中查找快呢？

所以，你需要把数据规模尽可能的控制在一定范围内。分而治之的思想，人类从古到今一直都在使用：

国家划分省，省划分市，市划分县，县划分乡镇，乡镇划分村庄；

大批量数据处理总是要先Map然后再Reduce；

还有似乎已经不是那么火的微服务以及越来越火的k8s。

总的来说，partition的两个最根本的好处是：

1. 方便治理——皇帝总不会直接治理平民的，没那么大精力。

2. 控制数据规模带来的性能提升——从1亿条数据中查找快还是从1万条数据中查找快呢？想想索引原理。

3. 还有一个个人认为最吸引人的好处，放在最后讲。

除此之外，某些情况下，数据不得不做partition，Redis为啥要搞集群？因为一台机器内存有限，装下所有数据，所以得把数据划分成一块一块的装进集群的一个个节点里面，而节点小了，做起replicaton来也更加方便和灵活了。

所以，partition和replication并不冲突，你可以认为它们是数据世界的两个维度——每一个partitio都可以做replication，同样的，每一个replication也都可以做partition。

数据的分片方式

给你一大堆数据，让你分成一小块一小块的，你咋分？

一般常见的分片方式有两种：按区间分片和按hash分片。

按区间分片

这种分片方式一般出现在时间和空间相关的数据存储中。

比如天气数据，或者监控系统的监控数控，一般都是按照时间段进行划分的。这种数据一般都产生的较“均匀”，比如监控系统每秒钟采集一次系统的CPU使用率，天气系统没小时采集一次温度数据。

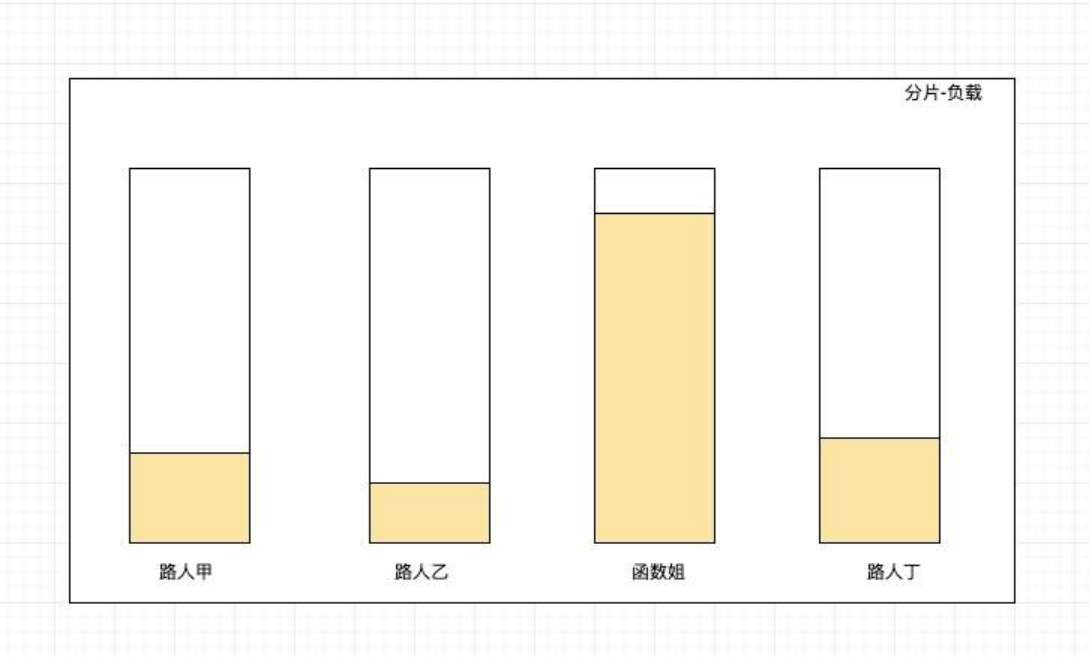
按hash分片。

这种数据多出现在关系型数据库中，比如电商系统、社交系统的用户信息，一般按用户id进行hash分。

热点key问题

无论以哪种方式进行分片，都应带尽可能避免热点问题。

热点问题是指绝大多数数据的读写集中在某一个或者某几个partition上，使得partition的性价比大打扣。微博大V就是数据热点问题一个典型的场景，如下图所示：



处理不好热点数据的问题，不但系统性能得不到提升，还会因为partition带来额外的资源浪费以及维成本。

热点key的问题，一直都是业界头疼的问题，至今似乎并没有一个完美的解决方案，微博的一次次崩真是难为某浪的研发同学了。

一个分片方式的好坏最直接的体现就是数据是否分的足够均匀，最理想的状态是每个分片的数据读写度都是差不多的，也即不存在热点数据。

二级索引

我们总是可以很方便的根据一级索引（主键索引）找到数据的分片所在，比如一个社交系统的用户信表，我们根据用户id（int类型）分了100张表，每个id % 100后就是该id所在的分表序号。

所以根据用户id来查找数据时，总是很迅速，很方便。可是，一旦按照用户id之外的维度来查找数据，问题就来了。比如，现在我想找出年龄最大的用户，尽管每张表上age字段都建立了索引，我们也得不遍历100张表，找出每个表年龄最大的用户，然后比较之后，得出我们想要的数据。

上面提到的情况是把二级索引和一级索引维护在同一张表中的情况。

仔细想想，我们还可以这样搞：把用户数据再按照年龄分100张表，1岁一张表（简单期间，不考虑现因素），我们就可以直接从序号100的表中找出年龄最大的用户数据来了。当然为了节省存储空间，们可以只存储年龄和用户id的映射，拿到用户id后，在根据id的分表规则找到用户信息。

你看，这不就和mysql的二级索引一个道理吗？所以，计算机世界的原理都是一样的。

业界还有这样一种常见的做法：

用hbase存储基本数据，用es存储其他字段的全文索引。查找时，先通过es查出数据的rowkey，然后rowkey去hbase查出具体数据。道理也是一样的。

Rebalance

维持各个分片数据量的均衡是一门艺术。

一个好的散列方式是维持数据均衡的前提。一般来说，数据的随机性越高，进入不同分片的概率就越于相同。

但是不幸的是，这个世界的本质是符合正态分布的，总是有少数的个体产生多数的数据，传说中的二原则。。。

比如，你很开心的把用户信息按用户id的最后一位数字均匀的散列到了10张表中，然后悲催的发现，几个用户的数据量特别大，更加悲催的是，这几个用户的id的最后一个数字都是7，然后悲哀的事情生了：序号是7的那张表的数据量疯狂增长，其他表的数据量则一直很缓和。

一段时间以后，你发现，序号是7的表数据量很快到达了千万级别，其他表也就几十万，这可如何是？？？？

预分区

在最开始的时候，预估数据的增长速度和能达到的规模，预先分表，这是目前mysql分库分表的常见法，实施起来比较简单，但是对数据预估的准确性和散列的把握要求很高，毕竟一开始分表的数目过，会浪费资源，查询起来也不方便（二级索引查询需要遍历所有分表）。但是一开始分表数目太小，来数据量增大后需要再次分表时，简直是噩梦。

但是，如果估计的比较准确，整个系统的性价比会大大得到提升，会把partition最大的好处——并行挥的淋漓尽致。

除了预分区，还有一种方式是动态分区，使每个分区维自动的持在固定的大小，常见于各种分布式NoQL存储系统中。

动态分区

像HBase，HDFS，Redis集群，ES等，不需要事先预留出足够的存储资源，而是在资源不够时，动态加就好了，也就是常说的扩容。

拿hbase举例，hbase会维持每一个partition的大小，当某个partition到达一定阈值时，会进行动分，一个partition分裂成两个，分裂的过程类似于B+数索引节点的分裂。这样我们就无需担心预估不确的问题了。

这样也有一个坏处，就是刚开始时，分区比较少，整个系统的并行性并不是那么好，这个时候，有一

做法，就是预分区，起始阶段，预先创造几个空的partition，等数据量上来了，再让他们自动分裂。

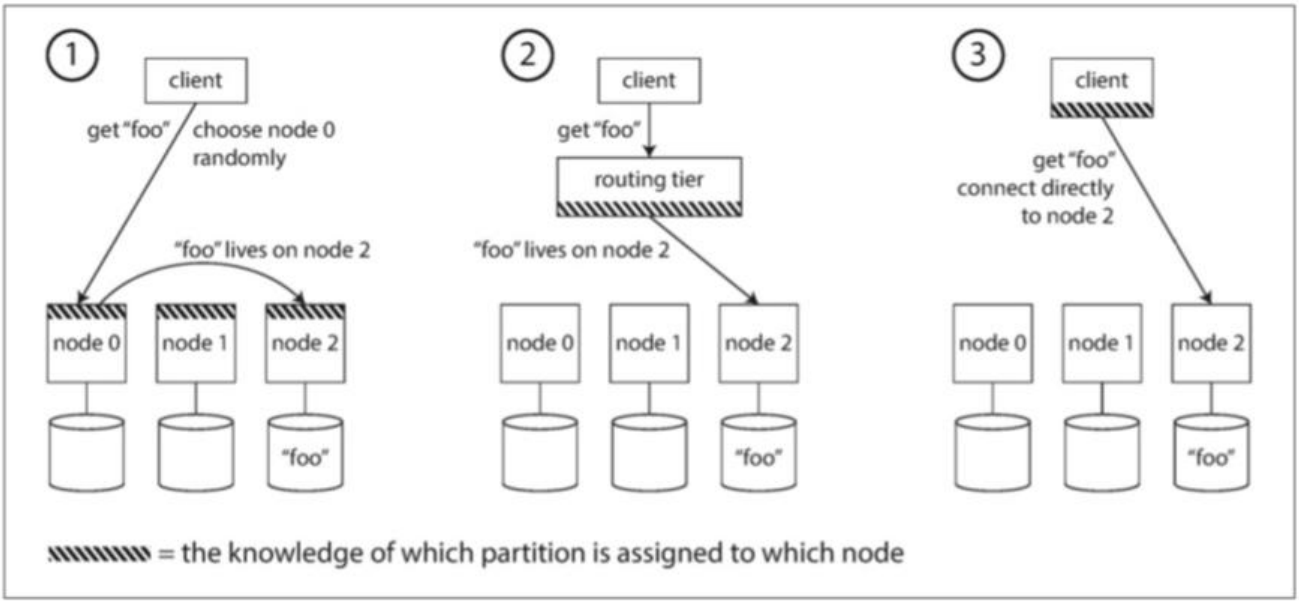
这种做法很好的结合了预估和动态分区的好处。

路由

数据被分隔成一片一片的，那么客户端进行读写时，怎么知道该去哪个分片上操作呢？这就是数据分带来的路由问题了。

数据路由问题的根本是找一个地方记录数据和分片的对应关系。

一般有三种方式，可以让客户端找到数据所在的分片，三种方式如下图（可耻的盗大师的图了）所示：



这三种方式，在行业中都有应用，下面分别说明。

数据节点维护对应关系

这种路由方式的请求流程如下：

1. 客户端随机找一个节点发起数据读写请求。
2. 收到请求的节点查找数据所在的正确节点。
3. 把请求转发到正确的节点上。

平时由于数据的增删和扩容缩容引起数据和节点的对应关系发生变化时，这种变化会在节点之间进行播，保证每个节点都知晓这种变化，这中去中心化数据一致性和时效性的实现还是比较困难的。

redis集群就是用的这种方法对请求进行路由的。

客户端维护对应关系

这种路由的请求凡是很简单：客户端直接找到正确的节点发送请求。

但是这种路由方式对客户端的要求比较高，客户端在初次连接时，需要向各个节点学习其上的对应关，并实时维护对应关系的变更。

redis集群也是支持这种路由方式的。

其实从某种意义上说，mysql的分库分表也属于这种类型，只不过客户端提前知道了数据和节点的对

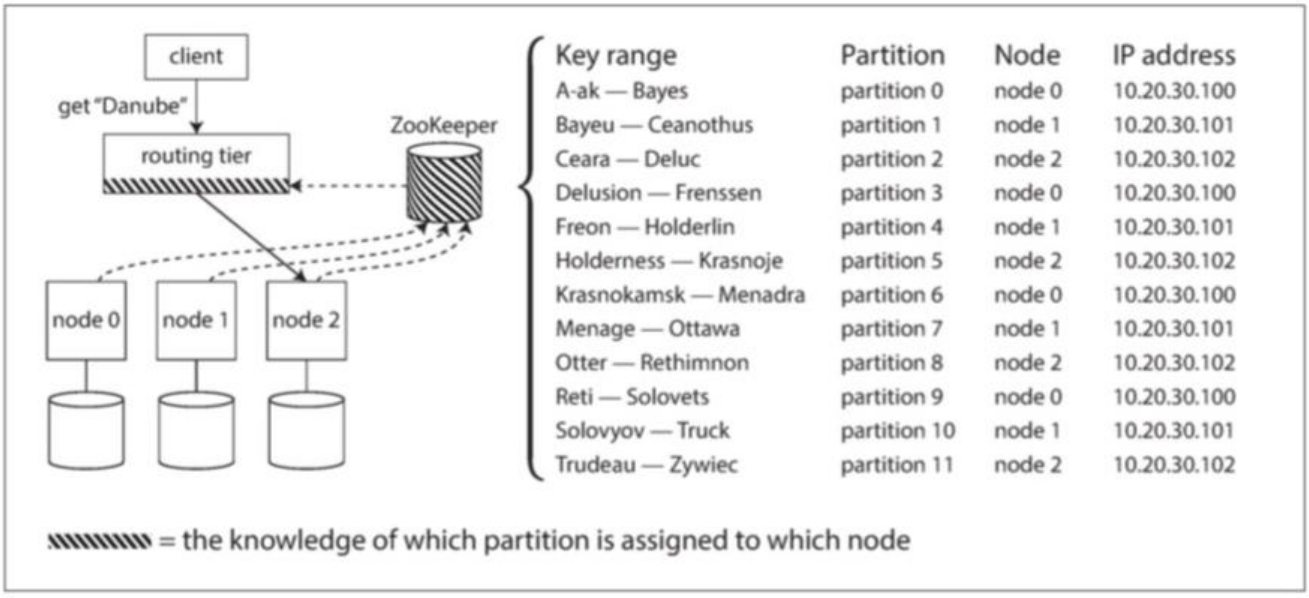
规则而已。

专门的节点维护对应关系

这种路由方式的请求流程如下所示：

1. 客户端询问这个专门的节点，要操作的数据在哪里。
2. 这个专门的节点告诉客户端一个ip。
3. 客户端向拿到的ip发起请求。

平时数据和节点的对应关系发生变化时，都会通知到这个专门的节点，通知多用ZooKeeper实现：



Key range	Partition	Node	IP address
A-ak — Bayes	partition 0	node 0	10.20.30.100
Bayeu — Ceanothus	partition 1	node 1	10.20.30.101
Ceara — Deluc	partition 2	node 2	10.20.30.102
Delusion — Frenssen	partition 3	node 0	10.20.30.100
Freon — Holderlin	partition 4	node 1	10.20.30.101
Holderness — Krasnoje	partition 5	node 2	10.20.30.102
Krasnokamsk — Menadra	partition 6	node 0	10.20.30.100
Menage — Ottawa	partition 7	node 1	10.20.30.101
Otter — Rethimnon	partition 8	node 2	10.20.30.102
Reti — Solovets	partition 9	node 0	10.20.30.100
Solovyov — Truck	partition 10	node 1	10.20.30.101
Trudeau — Zywiec	partition 11	node 2	10.20.30.102

这种方法应用比较广泛，像hbase啊，kafka啊都使用这种方式进行路由。

并行

开头提到的个人最喜欢的partition的好处就是数据分片带来的并行性。

我们都知道IO的资源是有限的，如果数据不分片，拿机械磁盘来说，不管你几个CPU，多少个线程，磁盘这里，都得串行的等磁头一次次转到数据所在的位置。而如果数据分散在多个磁盘上呢？比如来一个请求，要读取两条数据，一条在磁盘1上，一条在磁盘2上，那系统就可以同时将请求转发到磁盘和B上，同时读取，整个响应时间有了很大提升。

这种做法，其实并不只是体现在数据库上，想想，多个磁盘做RAID，还有对象存储把一个对象分成份存到不同的分片上，都让并行性有了很大提升。

结语

计算机的思想都来自于现实生活，很朴素但是也很有用，本系列到这里就接近尾声了，下一篇是最后篇了，最后一篇会尝试对整个系列进行一番总结，详见：存储（九）—— 道法自然