分布式系统中的分区

什么是分区？

对于非常大的数据集，或非常高的吞吐量，仅仅进行复制是不够的：我们需要将数据进行**分区**，也称为**分片**。

简单而言，**将一个大的数据集分为多个小的数据集，将这些小的数据集散布在更多的节点上，每一个小的数据集都作为一个独立的数据库进行处理，使得系统压力均匀的散布在多个节点上。**

例如将数据库中的一张表按照主键范围或其他方式分割成多张表，这就是分区。

分区的问题？

1. 如何将请求导向正确的分区？
2. 如何避免偏斜，即避免将大量数据集中导向同一个分区。
3. 当节点增加或减少时，如何优雅的调整分区在节点上的分布。

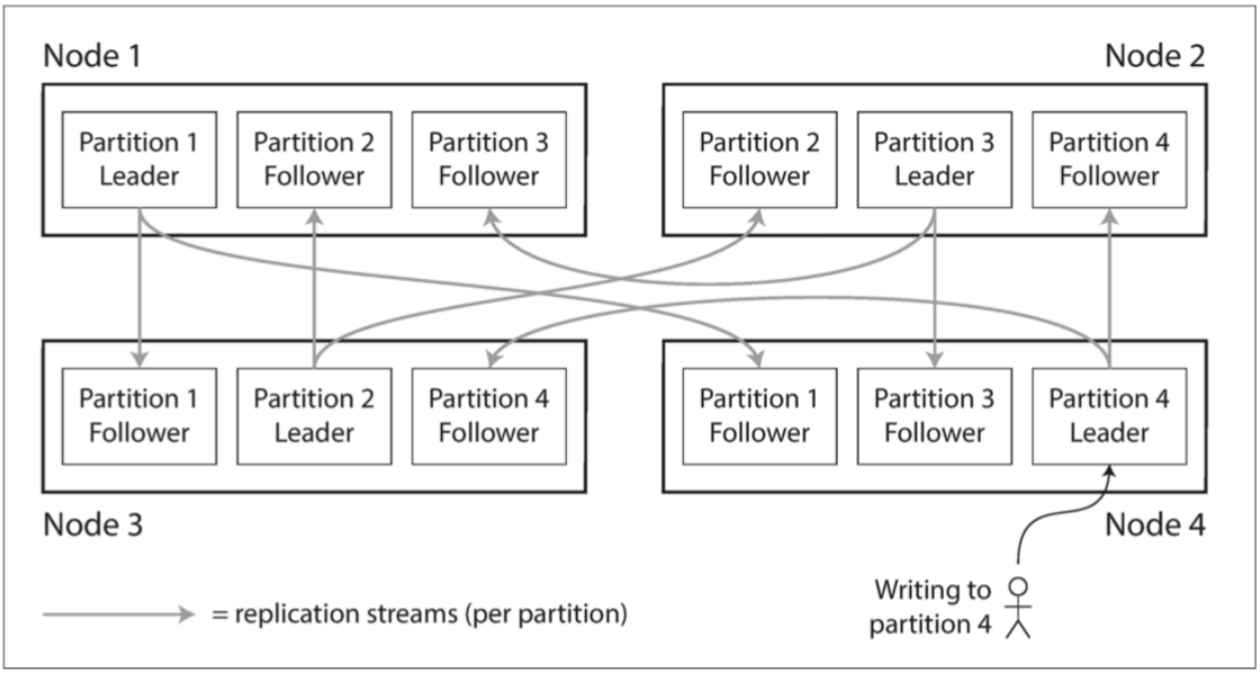
分区的优点？

1. 分区主要是为了**可伸缩性**，大数据集可以分布在多个磁盘上，并且查询负载可以分布在多个处理器上，因此减少了单个节点的压力，增加了整个系统的吞吐量。
2. 大型、复杂的查询可能会跨越多个节点并行处理，加快速度。

分区与复制

分区通常与复制结合使用，使得每个分区的副本存储在多个节点上，即使每条记录属于同一个分区，但是这个分区仍有可以分布在不同的节点上，提高容错。

节点通常被认为是一台主机，**一个节点可以有多个分区**，如果使用主从复制模型，每个分区领导者被分配给一个节点，追随者被分配个其他节点，每个节点可能是某些分区的领导者，同时是其他分区的追随者。



键值数据的分区

分区目标是将数据和查询负载均匀分布在各个节点上。

如果分区不公平，例如大量数据集中导向同一个分区，这被称为**偏斜（skew）**，数据偏斜的存在使分区效率下降很多。

在极端的情况下，所有的负载可能压在一个分区上，其余所有节点都是空闲的，瓶颈落在这一个繁忙的节点上。

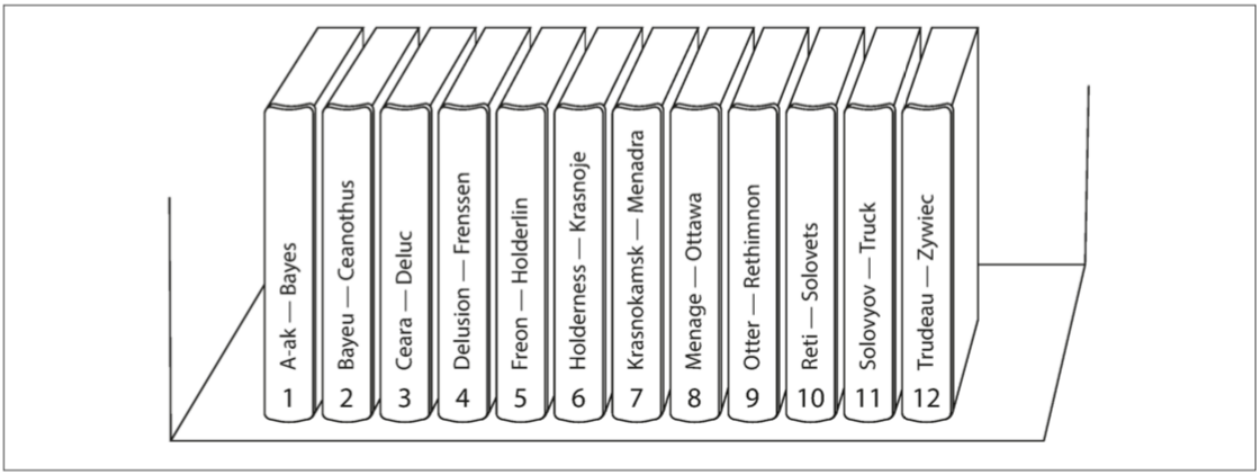
不均衡导致的高负载的分区被称为热点（hot spot）。

怎么避免热点？避免热点最简单的方法是将记录随机分配给节点，但这缺点是读特定的值时，不知道在哪个节点上，必须并行查所有的节点。

以下我们来探讨一下常见的分区方式。

根据键的范围分区

一种分区的方法是为每个分区指定一块连续的键范围（从最小值到最大值），类似纸质的百科全书、类似于数据库主键切分。



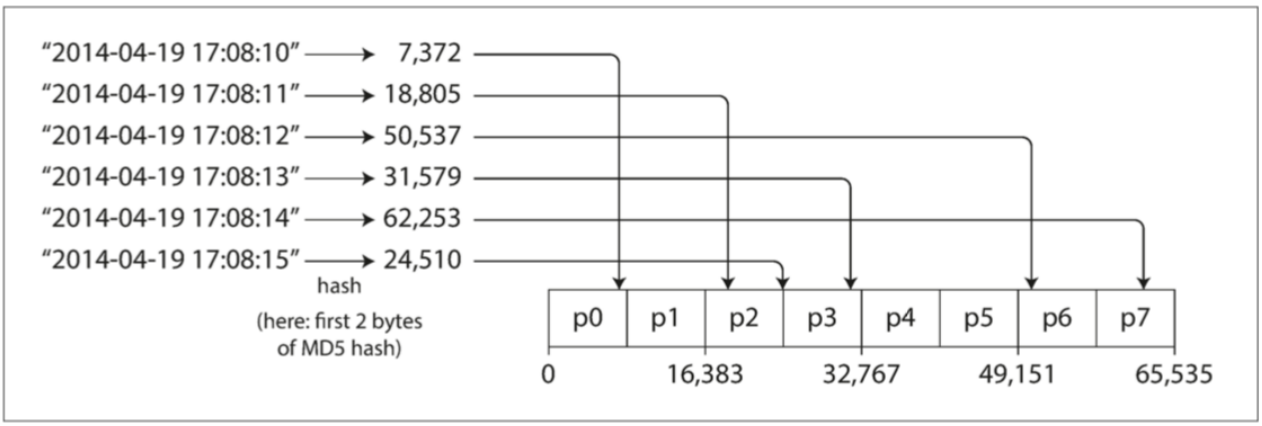
根据键的范围分区优点是查询稳定高效，由于所有分区按照键排序，无论是定值查询或者是范围查询，都可以在$log(N)$的时间内定位，无须遍历整个分区（前提是查询请求必须是主键）。

缺点也是很明显的，由于不是随机分布的，分区可能无法很好的均匀分布在各个节点中，这就**可能造成偏斜与热点问题**。例如某种数据库是按照时间为主键(以天为单位)分区的，那么当天产生的请求可能会全部路由到同一个分区中。

根据键的散列分区

另一种分区方式是根据键的散列分区，目前很多分布式数据存储使用散列函数来分区。

一个好的散列函数可以将偏斜的数据均匀分布。



根据键的散列分区优点是擅长在分区之间公平地分配键，并且定值查询非常高效。但是要注意，散列分区可以有效的消除数据偏斜与热点问题，但无法完全避免，相同的键仍然会产生相同的哈希，在极端情况下，所有的读写操作都是针对同一个键的，所有的请求都会被路由到同一个分区。

> 针对这种极端情况的解决办法是在主键后面添加一位随机数。

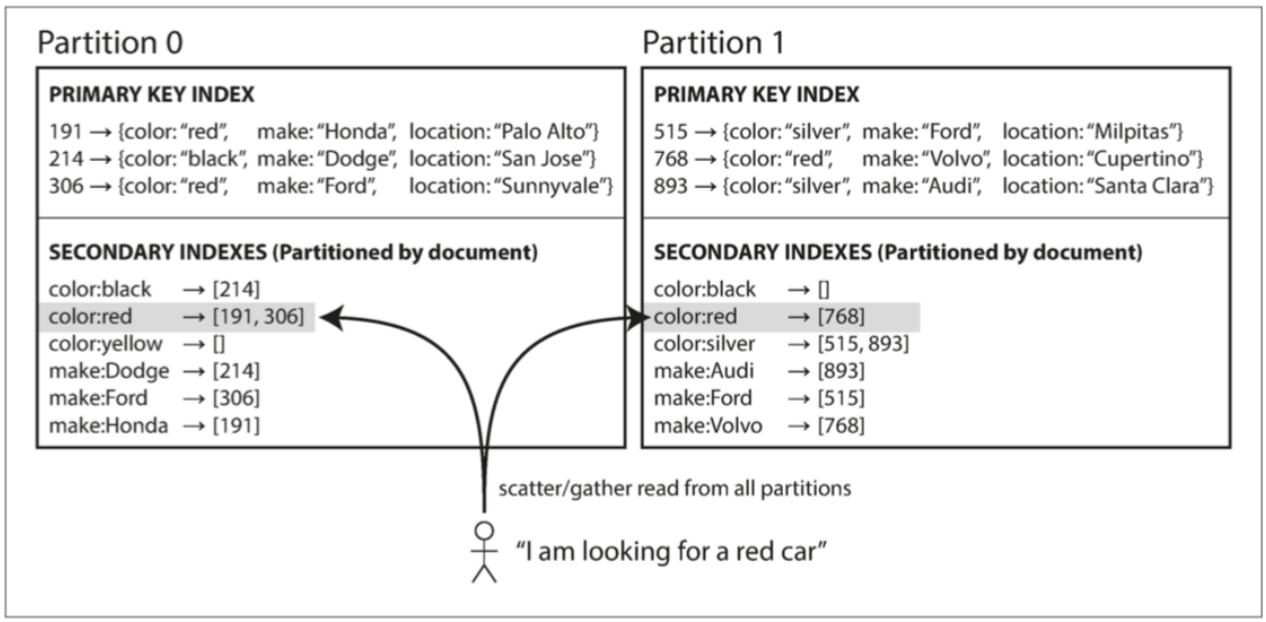
缺点是失去了高效执行范围查询的能力，范围查询要么不支持，要么需要查询所有分区。

分区与次级索引

上述分区都是主键的分区，如何在分区引入二级索引？

基于文档的二级索引进行分区

在这种方式下，每个分区自己内部维护自己的二级索引，例如下图中 color 为次级索引：



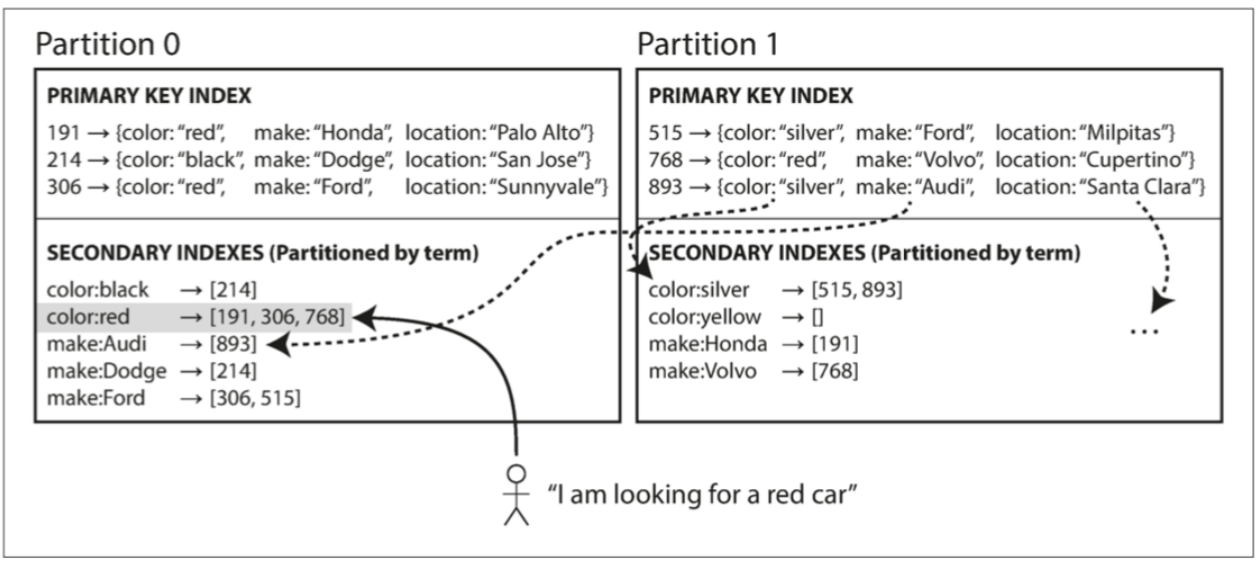
这样做的优点是应对次级索引变化简单，每个分区各自维护自己的二级索引，在本地维护速度会相当快。

缺点是查询时必须查询所有的分区，并且要进行去重合并，这会浪费大量的时间。

事实上由于二级索引很少用到，为维持高效性，目前大多数解决方案都是基于文档的方式。

基于关键词的二级索引进行分区

在这种情况下，我们**维护一个全局的二级索引表**，这张全局的表也应该采用分区的方式存储，这种分区叫做**关键词分区**，查询或修改二级索引都需要额外导向全局二级索引表，



这样做的优点是查询只需查一次全局的二级索引表，无须查询所有分区。

缺点是维护代价较高，每次维护都需要一次额外的对二级索引表的网络 IO，在网络拥挤时代价显得更为严重。

分区再平衡

随着时间的推移，数据库会有各种变化：

- 查询吞吐量增加，所以您想要添加更多的 CPU 来处理负载。
- 数据集大小增加，所以您想添加更多的磁盘和 RAM 来存储它。
- 机器出现故障，其他机器需要接管故障机器的责任。

所有这些更改增加或移除一个或多个节点，而**节点变化会导致分区拓扑发生变化**，数据和请求可能会从一个节点移动到另一个节点。

将分区（负载）从集群中的一个节点向另一个节点移动的过程称为 **再平衡（rebalancing）**。

同样的，将一个请求从一个节点导向另一个节点的过程也成为 **再平衡（rebalancing）**。

无论使用哪种分区方案，再平衡通常都要满足一些最低要求：

- 再平衡之后，负载（数据存储，读取和写入请求）应该在集群中的节点之间公平地共享。
- 再平衡发生时，数据库应该继续接受读取和写入。
- 节点之间只移动必须的数据，以便快速再平衡，并减少网络和磁盘I/O负载。

反面教材：hash mod N

如果使用基于散列的分区方法，并且分区与节点的映射关系为 $hash \ mod \ N, N=节点总数$，那么一旦 N 发生变化，绝大多数分区可能都会发生变化。

考虑以下例子：

3个机器节点，10个数据 的哈希值分别为1,2,3,4,...,10。使用的哈希函数为：$m=hash(o) \ mod \ 3$

```
机器0 上保存的数据有：3，6，9
机器1 上保存的数据有：1，4，7，10
机器2 上保存的数据有：2，5，8
```

当增加一台机器后，此时n = 4，使用的哈希函数为：$m=hash(o) \ mod \ 4$，各个机器上存储的数据分别为：

```
机器0 上保存的数据有：4，8
机器1 上保存的数据有：1，5，9
机器2 上保存的数据有：2，6，10
机器3 上保存的数据有：3，7
```

只有数据1和数据2没有移动，所以**当集群中数据量很大时，采用这种哈希函数，在节点数量动态变化的情况下会造成大量的数据迁移**，导致网络通信压力的剧增，严重情况，还可能导致数据库宕机。

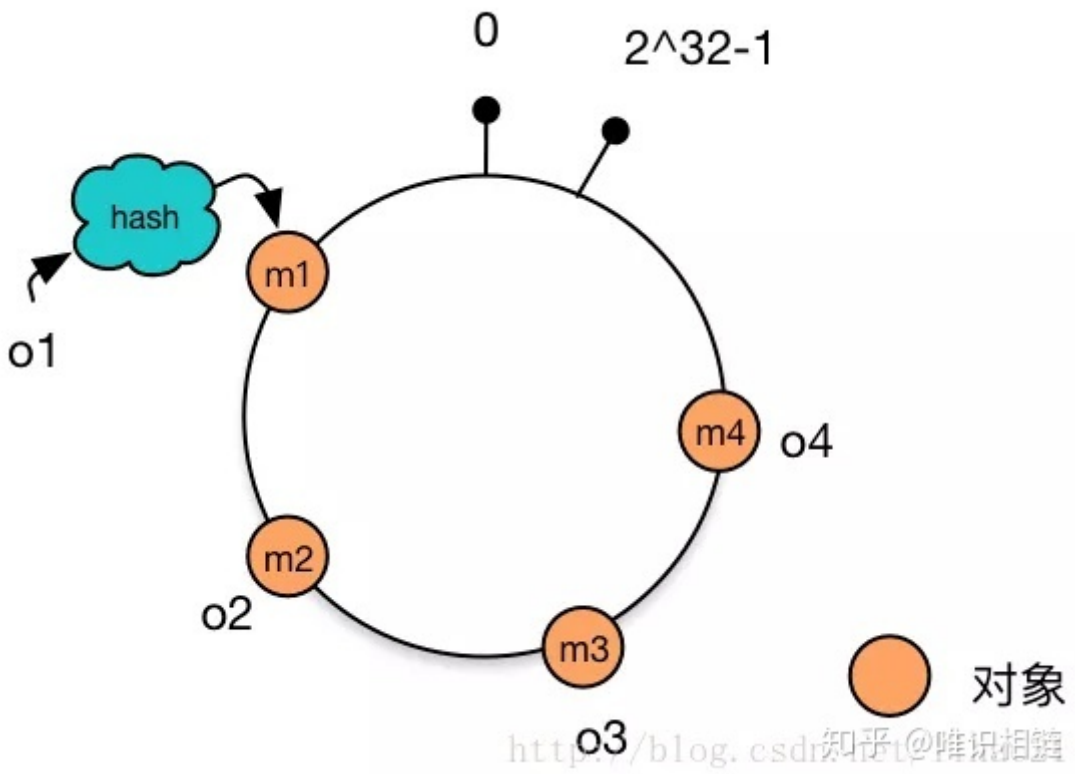
一致性哈希算法

所以我们需要一种哈希算法使得再平衡发生时，不会产生大量的数据迁移，一致性哈希算法是一种解决方案，一致性哈希算法并不是一个具体的哈希算法，而是一种抽象的描述。

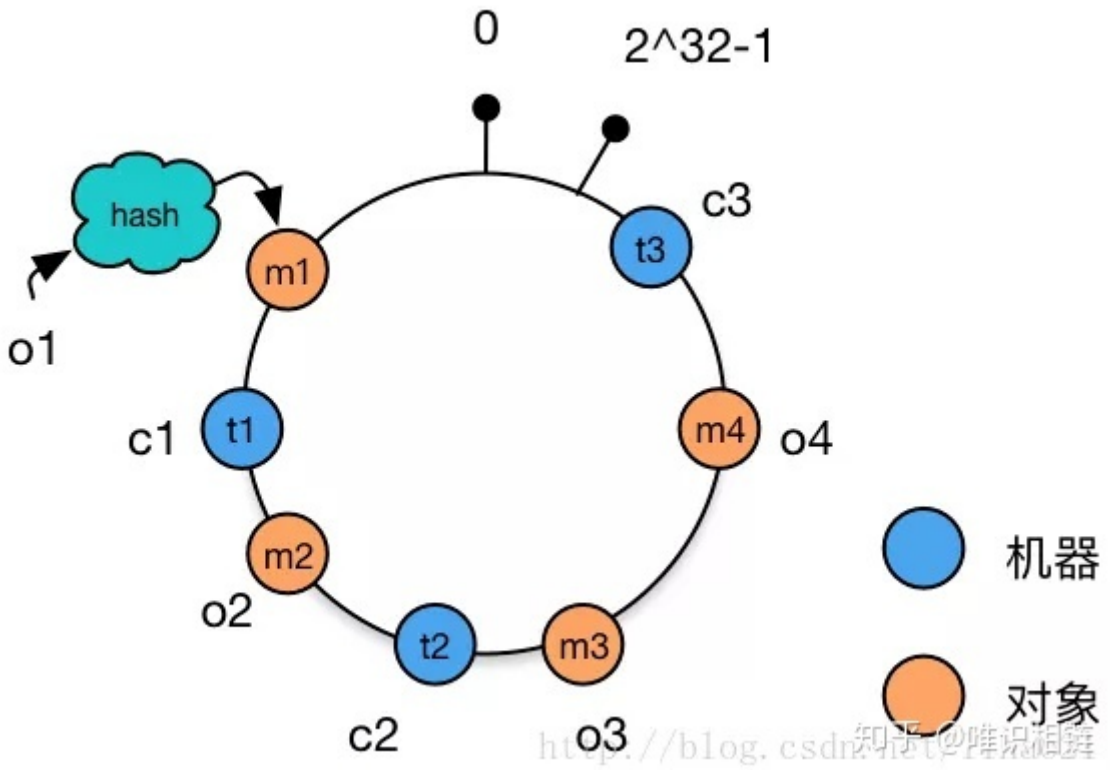
在上述简单哈希中，我们仅仅只对数据进行哈希，然后模 N 得到节点编号。在一致性哈希算法中，我们不仅对数据进行哈希，还会对节点进行哈希，并且会将哈希值抽象看作在一个环上。

首先对数据对象进行哈希，假设哈希的结果 $0<=hash<=Integer.MAX_VALUE$，将结果抽象的看成一个环。

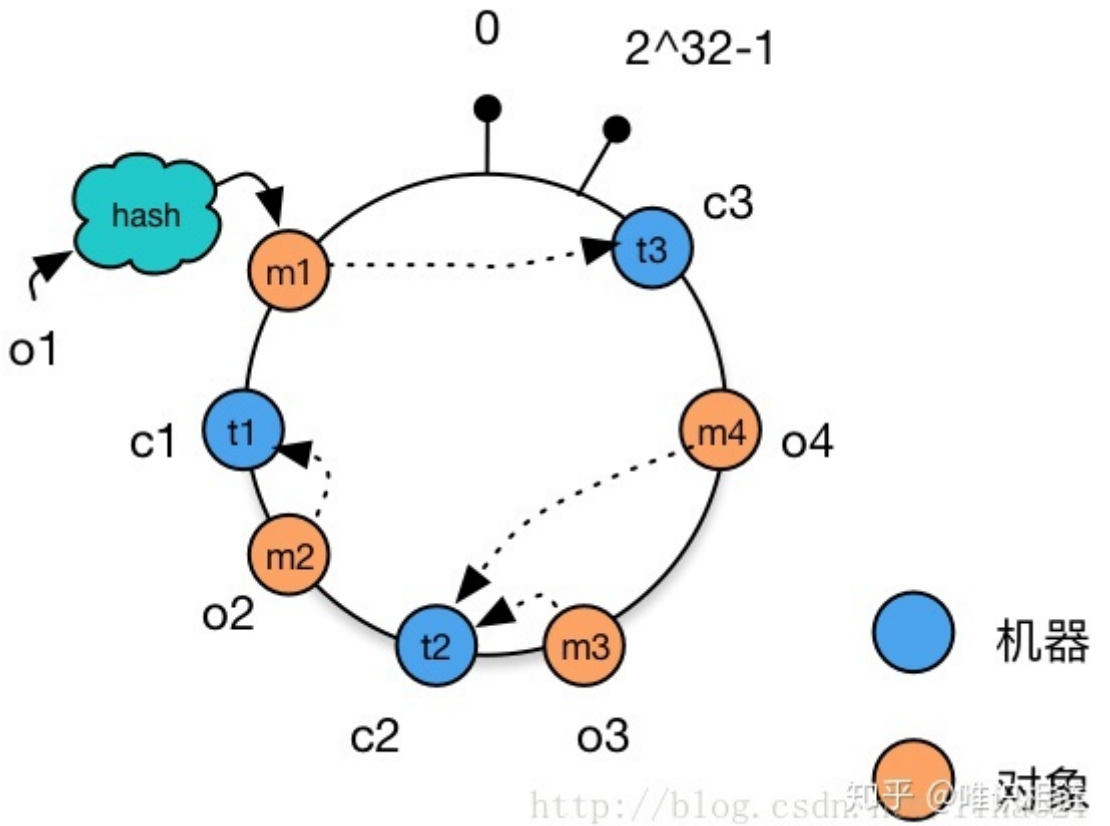
这里采用哈希算法，o1 映射到 m1，o2 映射到 m2，以此类推。



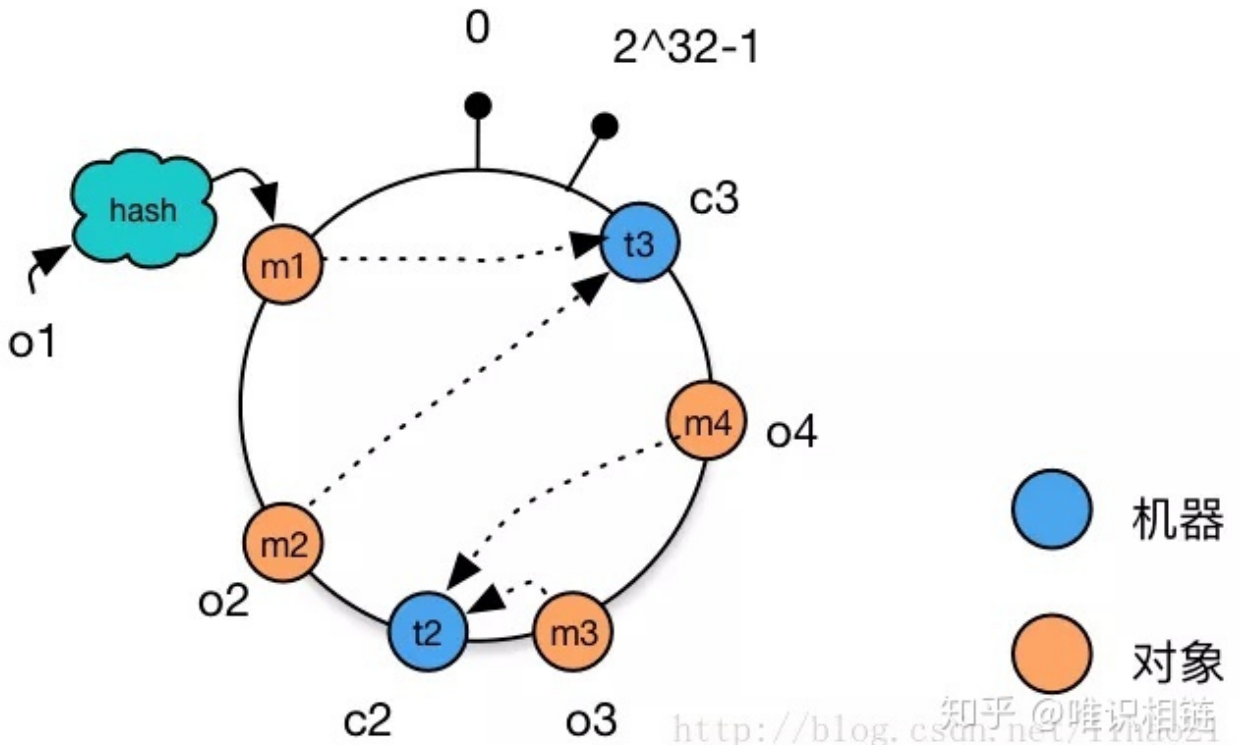
现在，我们可以为节点也分配一个唯一的主键，通过该键同样进行哈希映射，假设现在节点 c1 映射到 t1 上，节点 c2 映射到 t2 上，以此类推。



现在我们如何判定数据对象属于哪一个节点呢？在一致性哈希算法中，**数据对象属于该对象在环中所遇到的下一个节点（通常为顺时针）**，例如上述 o1 属于 c3，o4 与 o3 属于 c2，o2 属于 m2。



现在如果新增或删除一个节点，只会影响最多环上两个节点之间的数据，例如删除节点 c1 时，只会影响 c1 和 c2 之间的节点，现在 o2 被重定位至 c3 中。



但即使是采用一致性哈希算法，也无法避免数据偏斜问题，极端情况下，所有数据对象都被哈希聚集到环上的一端，所有的对象都被导向同一个分区。

简单的解决方案是采用更好的哈希算法，让数据均匀分布。

但这可能无法满足，另一种解决的方案是**加入更多的节点，让节点在环上均匀分布**，如果没有足够多的节点，我们可以让一个节点在环上存在多个哈希值，这种方案被称为**虚拟节点**。

但是要注意，虚拟节点并不是越多越好的，当虚拟节点较多时，新增一个物理节点相当于新增了多个节点，可能仍然会导致大量数据进行迁移，得不偿失。

某些时候，人为规定节点对应哈希，使节点在环上均匀分布可能是更好的选择。

对于请求而言，简单的方法是在环上进行二分查找，找到第一个 hash 比当前数据 hash 大的节点，然后导向它，这非常高效。

另一种办法是利用路由表或是服务注册功能，以 O1 的时间进行查找。

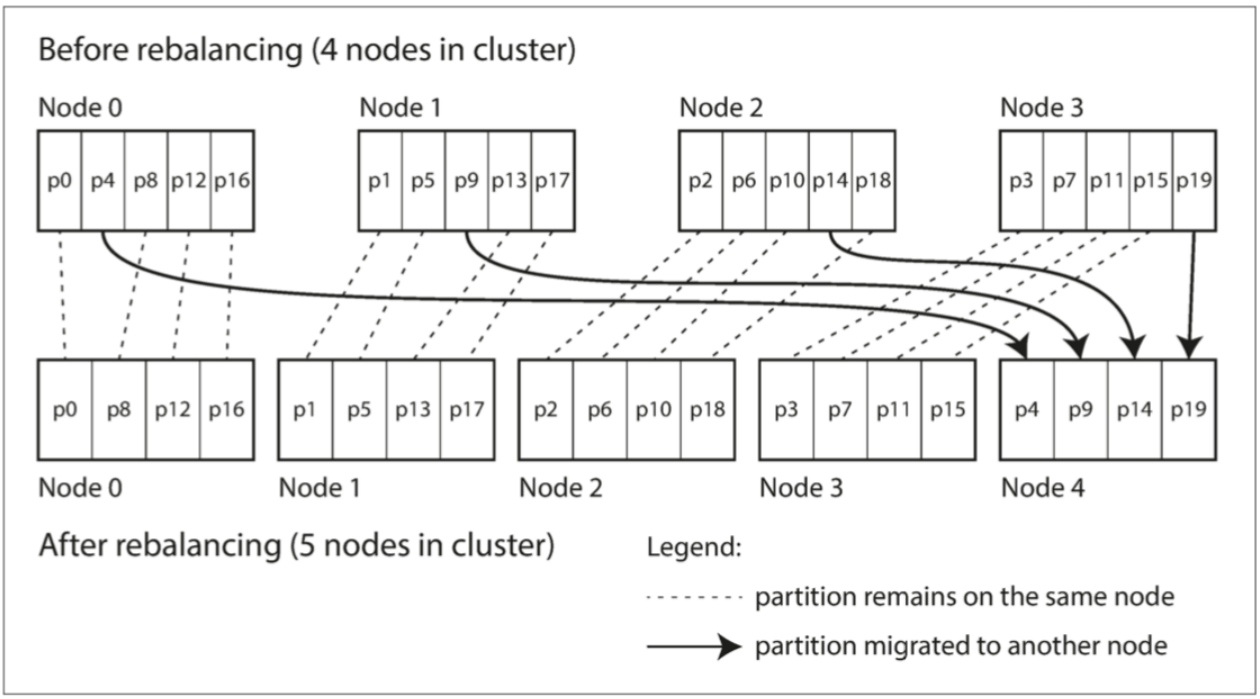
多一层抽象：分区 - 节点 多对一

一致性哈希算法是一种很优秀的解决方案，但实现较为复杂，这里还有一种简单高效的解决方案：**创建比节点更多的分区，并为每个节点分配多个分区，即额外加一层抽象。**

> 注意：在之前的分析中，为了简便，我们认为分区 - 节点 是 1 对 1 的，所以我们之前淡化了分区与节点的关系（但绝不是必须的，分区 - 节点 仍然可以多对一，不过

> 是多加层抽象罢了）。
>
> 在这里，分区 - 节点是多对一的，因此这里的哈希是指 分区 - 节点 之间的哈希，而不是 数据 - 分区 之间的哈希，请务必不要混淆。

哈希算法仍然是取模，例如假设本来具有 4 个节点，$m = hash \ mod \ 4$，现在新增一个节点 Node4，哈希算法变为 $m = hash \ mod \ 5$，那么只需要将 $hash \ mod \ 5 = 4$ 的分区分给 Node4 即可。这样只有 4 个数据分区需要移动。



这种方法的优点是简单高效，只有部分分区在节点中移动，键所在的分区也不会改变，唯一改变的是分区所在的节点。

Riak ，Elasticsearch ，Couchbase 和 Voldemort 中使用了这种再平衡的方法。

缺点是，**必须要遍历所有的分区**，计算散列，并将其分配给新节点。

请求路由

服务发现(service discovery)

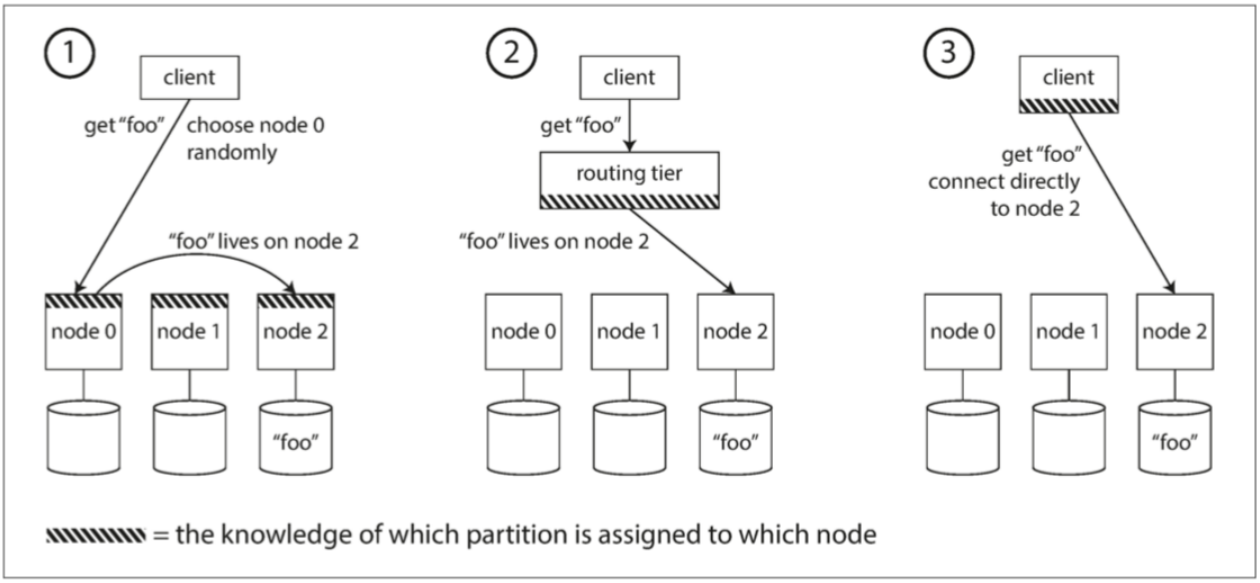
- 确定客户发出请求时，知道要连接哪个节点进行读取。

概括来说，这个问题有几种不同的方案（如图6-7所示）：

1. 允许客户联系任何节点（例如，通过**循环策略的负载均衡（Round-Robin Load Balancer）**）。如果该节点恰巧拥有请求的分区，则它可以直接处理该请求；否则，它将请求转发到适当的节点，接收回复并传递给客户端。

2. 首先将所有来自客户端的请求发送到路由层，它决定了应该处理请求的节点，并相应地转发。此路由层本身不处理任何请求；它仅负责分区的负载均衡。

3. 要求客户端知道分区和节点的分配。在这种情况下，客户端可以直接连接到适当的节点，而不需要任何中介。



关键问题：

- 如何了解分区-节点之间的分配关系变化？
- 解决方法：所有参与者达成共识。
- 分布式系统中有达成共识的协议，但很难被正确实现。

常见实现 ZooKeeper：

- 依赖于一个独立的协调服务，比如ZooKeeper来跟踪集群元数据
- 每个节点在ZooKeeper中注册自己，ZooKeeper维护分区到节点的可靠映射。
- 其他参与者（如路由层或分区感知客户端）可以在ZooKeeper中订阅此信息。
- 只要分区分配发生了改变，或者集群中添加或删除了一个节点，ZooKeeper就会通知路由层使路由信息保持最新状态。