NoSQL生态系统

译：iammutex

13.1 NoSQL其名
- 13.1.1 SQL及其关联型结构
- 13.1.2 NoSQL的启示
- 13.1.3 特性概述

13.2 NoSQL数据模型及操作模型
- 13.2.1 基于key值存储的NoSQL数据模型
  - Key-Value 存储
  - Key - 结构化数据 存储
  - Key - 文档 存储
  - BigTable 的列簇式存储
- 13.2.2 图结构存储
- 13.2.3 复杂查询
- 13.2.4 事务机制
- 13.2.5 Schema-free的存储

13.3 数据可靠性
- 13.3.1 单机可靠性
  - 控制fsync的调用频率
  - 使用日志型的数据结构
  - 通过合并写操作提高吞吐性能
- 13.3.2 多机可靠性

13.4 横向扩展带来性能提升
- 13.4.1 如非必要，请勿分片
  - 读写分离
  - 使用缓存
- 13.4.2 通过协调器进行数据分片
- 13.4.3 一致性hash环算法
  - Hash环图*
  - 备份数据
  - 优化的数据分配策略
- 13.4.4 连续范围分区
  - BigTable的处理方式

故障处理
基于范围分区的NoSQL项目
13.4.5 选择哪种分区策略
13.5 一致性
13.5.1 关于CAP理论
13.5.2 强一致性
13.5.3 最终一致性
版本控制与冲突
冲突解决
读时修复
Hinted Handoff
Anti-Entropy
Gossip
13.6 写在最后的话
13.7 致谢
相关信息

与本书中提到的其它主题不同，NoSQL不是一个工具，而是由一些具有互补性和竞争性的工具组成的一个概念，是一个生态圈。这些被称作NoSQL的工具，在存储数据的方式上，提供了一种与基于SQL语言的关系型数据截然不同的思路。要想了解NoSQL，我们必须先了解现有的这些工具，去理解那些让他们开拓出新的存储领域的设计思路。

如果你正在考虑使用NoSQL，你应该会马上发现你有很多种选择。NoSQL系统舍弃了许了传统关系型数据库的方便之处，而把一些通常由关系型数据库本身来完成的任务交给了应用层来完成。这需要开发人员更深入的去了解存储系统的架构和具体实现。

13.1 NoSQL其名

在给NoSQL下定义之前，我们先来试着从它的名字上做一下解读，顾名思义，NoSQL系统的数据操作接口应该是非SQL类型的。但在NoSQL社区，NoSQL被赋予了更具有包容性的含义，其意为Not Only SQL，即NoSQL提供了一种与传统关系型数据库不太一样的存储模式，这为开发者提供了在关系型数据库之外的另一种选择。有时候你可能会完全用NoSQL数据库代替关系型数据

加，但你也可以同时使用关系型和非关系型存储来解决具体的问题。

在进入NoSQL的大门之前，我们先来看看哪些场景下使用关系型数据库更合适，哪些使用NoSQL更合适。

13.1.1 SQL及其关联型结构

SQL是一种任务描述性的查询语言，所谓任务描述性的查询语言，就是说它只描述他需要系统做什么，而不告诉系统如何去做。例如：查出39号员工的信息，查出员工的名字和电话，只查找做会计工作的员工信息，计算出每个部门的员工总数，或者是对员工表和经理表做一个联合查询。

简单的说，SQL让我们可以直接向数据库提出上述问题而不必考虑数据是如何在磁盘上存储的，使用哪些索引来查询数据，或者说用哪种算法来处理数据。在关系型数据库中有一个重要的组件，叫做查询优化器，正是它来推算用哪种操作方式能够更快的完成操作。查询优化器通常比一般的数据库用户更聪明，但是有时候由于没有充足的信息或者系统模型过于简单，也会导致查询优化器不能得出最有效的操作方式。

作为目前应用最广的数据库系统，关系型数据库系统以其关联型的数据模型而命名。在关联型的数据模型中，在现实世界中的不同类型的个体被存储在不同的表里。比如有一个专门存员工的员工表，有一个专门存部门的部门表。每一行数据又包含多个列，比如员工表里可能包含了员工号，员工工资，生日以及姓名等，这些信息项被存在员工表中的某一列中。

关联型的模型与SQL是紧密想连的。简单的查询操作，比如查询符合某个条件的所有行（例：employeeid = 3, 或者 salary > $20000）。更复杂一些的任务会让数据库做一些额外的工作，比如跨表的联合查询（例：查出3号员的部门名称是什么）。一些复杂的查询，比如统计操作（例：算出所有员工的平均工资），甚至可能会导致全表扫描。

关联型的数据模型定义了高度结构化的数据结构，以及对这些结构之间关系的严格定义。在这样的数据模型上执行的查询操作会比较局限，而且可能会导致复杂的数据遍历操作。数据结构的复杂性及查询的复杂性，会导致系统产生如下的一些限制：

- 复杂导致不确定性。使用SQL的一个问题就是计算某个查询的代价或者产生的负载几乎是不可能的。使用简单的查询语言可能会导致应用层的逻辑更复杂，但是这样可以将存储系统的工作简单化，让它只需要响应一些简单的请求。
- 对一个问题建模有很多种方式。其中关联型的数据模型是非常严格的一种：表结构的定义规定了表中每一行数据的存储内容。如果你的数据结构化并没有那么强，或者对每一行数据的要求比较灵活，那可能关联型的数据模型就太过严格了。类似的，应用层的开发人员可能对关联型的数据结构并不满意。比如很多应用程序是用面向对象的语言写的，数据在

这些语言中通常是以列表、队列或集合的形式组织的，程序员们当然希望他们的数据存储层也能和应用层的数据模型一致。

- 当数据量增长到一台机器已经不能容纳，我们需要将不同的数据表分布到不同的机器。而为了避免在不同机器上的数据表在进行联合查询时需要跨网络进行。我们必须进行反范式的数据库设计，这种设计方式要求我们把需要一次性查询到的数据存储在一起。这样做使得我们的系统变得就像一个主键查询系统一样，于是我们开始思考，是否有其它更适合我们数据的数据模型。

通常来说，舍弃多年以来的设计思路是不明智的。当你要把数据存到数据库，当考虑到SQL与关联型的数据模型，这些都是数十年的研究的开发成果，提供丰富的数据模型，提供复杂操作的保证。而当你的问题涉及到大数据量，高负载或者说你的数据结构在SQL与关联型数据模型下很难得到优化，NoSQL可能是更好的选择。

13.1.2 NoSQL的启示

NoSQL运动受到了很多相关研究论文的启示，这所有论文中，最核心的有两个。

Google的BigTable[CDG+06]提出了一种很有趣的数据模型，它将各列数据进行排序存储。数据值按范围分布在多台机器，数据更新操作有严格的一致性保证。

Amazon的Dynamo[DHJ+07]使用的是另外一种分布式模型。Dynamo的模型更简单，它将数据按key进行hash存储。其数据分片模型有比较强的容灾性，因此它实现的是相对松散的弱一致性：最终一致性。

接下来我们会深入介绍这些设计思想，而实际上在现实中这些思想经常是混搭使用的。比如像HBase及其它一些NoSQL系统他们在设计上更接受BigTable的模型，而像Voldemort 系统它就和Dynamo更像。同时还有像Cassandra这种两种特性都具备的实现（它的数据模型和BigTable类似，分片策略和一致性机制和Dynamo类似）。

13.1.3 特性概述

NoSQL系统舍弃了一些SQL标准中的功能，取而代之的是提供了一些简单灵活的功能。NoSQL的构建思想就是尽量简化数据操作，尽量让执行操作的效率可预知。在很多NoSQL系统里，复杂的操作都是留给应用层来做的，这样的结果就是我们对数据层进行的操作得到简化，让操作效率

可预知。

NoSQL系统不仅舍弃了很多关系数据库中的操作。它还可能不具备关系数据库以下的一些特性：比如通常银行系统中要求的事务保证，一致性保证以及数据可靠性的保证等。事务机制提供了在执行多个命令时的all-or-nothing保证。一致性保证了如果一个数据更新后，那么在其之后的操作中都能看到这个更新。可靠性保证如果一个数据被更新，它就会被写到持久化的存储设备上（比如说磁盘），并且保证在数据库崩溃后数据可恢复。

通过放宽对上述几点特性的要求，NoSQL系统可以为一些非银行类的业务提供以性能换稳定的策略。而同时，对这几点要求的放宽，又使得NoSQL系统能够轻松的实现分片策略，将远远超出单机容量的大量数据分布在多台机器上的。

由于NoSQL系统还处在萌芽阶段，本章中提到的很多NoSQL架构都是用于满足各种不同用户的需求的。对这些架构进行总结不太可能，因为它们总在变化。所以希望你能记住的一点是，不同的NoSQL系统的特点是不同的，通过本章的内容，希望你能该根据自己的业务情况来选择合适的NoSQL系统，而本章内容不可能给你直接的答案。

当你去考查一个NoSQL系统的时候，下面的的几点是值得注意的：

- 数据模型及操作模型：你的应用层数据模型是行、对象还是文档型的呢？这个系统是否能支持你进行一些统计工作呢？
- 可靠性：当你更新数据时，新的数据是否立刻写到持久化存储中去了？新的数据是否同步到多台机器上了？
- 扩展性：你的数据量有多大，单机是否能容下？你的读写量求单机是否能支持？
- 分区策略：考虑到你对扩展性，可用性或者持久性的要求，你是否需要一份数据被存在多台机器上？你是否需要知道数据在哪台机器上，以及你能否知道。
- 一致性：你的数据是否被复制到了多台机器上，这些分布在不同点的数据如何保证一致性？
- 事务机制：你的业务是否需要ACID的事务机制？
- 单机性能：如果你打算持久化的将数据存在磁盘上，哪种数据结构能满足你的需求（你的需求是读多还是写多）？写操作是否会成为磁盘瓶颈？
- 负载可评估：对于一个读多写少的应用，诸如响应用户请求的web应用，我们总会花很多精力来关注负载情况。你可能需要进行数据规模的监控，对多个用户的数据进行汇总统计。你的应用场景是否需要这样的功能呢？

尽管后三点在本章没有独立的小节进行描述，但它们和其它几点一样重要，下面就让我们对以上的几点进行阐述。

13.2 NoSQL数据模型及操作模型

数据库的数据模型指的是数据在数据库中的组织方式，数据库的操作模型指的是存取这些数据的方式。通常数据模型包括关系模型、键值模型以及各种图结构模型。的操作语言可能包括SQL、键值查询及MapReduce等。NoSQL通常结合了多种数据模型和操作模型，提供了不一样的架构方式。

13.2.1 基于key值存储的NoSQL数据模型

在键值型系统中，复杂的联合操作以及满足多个条件的取数据操作就不那么容易了，需要我们创造性的建立和使用键名。如果一个程序员既想通过员工号查到员工信息，又想通过部门号查到员工信息，那么他必须建立两种类型的键值对。例如 emplyee:30 这个键用于指向员工号为30的员工信息。employee_departments:20 可能用到指向一个包含在20号部门工作的所有员工工号的列表。这样数据库中的联合操作就被转换成业务层的逻辑了：要获取部门号为20的所有员工的信息，应用层可以先获取employee_departments:20 这个列表，然后再循环地拿这个列表中的ID通过获取employee:ID得到所有员工的信息。

键值查找的一个好处是，对数据库的操作模式是固定的，这些操作所产生的负载也是相对固定且可预知的。这样像分析整个应用中的性能瓶颈这种事，就变得简单多了。因为复杂的逻辑操作并不是放在数据库里面黑箱操作了。不过这样做之后，业务逻辑和数据逻辑可能就没那么容易分清了。

下面我们快速地把各种键值模型的数据结构简单描述一下。看看不同的NoSQL系统的在这方面的不同实现方式。

Key-Value 存储

Key-Value存储可以说是最简单的NoSQL存储。每个key值对应一个任意的数据值。对NoSQL 系统来说，这个任意的数据值是什么，它并不关心。比如在员工信念数据库里，exployee:30 这个key对应的可能就是一段包含员工所有信息的二进制数据。这个二进制的格式可能是Protocol Buffer、Thrift或者Avro都无所谓。

如果你使用上面说的Key-Value存储来保存你的结构化数据，那么你就得在应用层来处理具体的

数据结构：单纯的Key-Value存储是不提供针对数据中特定的某个属性值来进行操作。通常它只提供像set、get和delete这样的操作。以Dynamo为原型的Voldemort数据库，就只提供了分布式的Key-Value存储功能。BDB 是一个提供Key-Value操作的持久化数据存储引擎。

Key - 结构化数据 存储

Key对应结构化数据存储，其典型代表是Redis，Redis将Key-Value存储的Value变成了结构化的数据类型。Value的类型包括数字、字符串、列表、集合以及有序集合。除了set/get/delete 操作以为，Redis还提供了很多针对以上数据类型的特殊操作，比如针对数字可以执行增、减操作，对list可以执行 push/pop 操作，而这些对特定数据类型的特定操作并没有对性能造成多大的影响。通过提供这种针对单个Value进行的特定类型的操作，Redis可以说实现了功能与性能的平衡。

Key - 文档 存储

Key - 文档存储的代表有CouchDB、MongoDB和Riak。这种存储方式下Key-Value的Value是结构化的文档，通常这些文档是被转换成JSON或者类似于JSON的结构进行存储。文档可以存储列表，键值对以及层次结构复杂的文档。

MongoDB 将Key按业务分到各个collection里，这样以collection作为命名空间，员工信息和部门信息的Key就被隔开了。CouchDB和Riak把类型跟踪这种事留给了开发者去完成。文档型存储的灵活性和复杂性是一把双刃剑：一方面，开发者可以任意组织文档的结构，另一方面，应用层的查询需求会变得比较复杂。

BigTable 的列簇式存储

HBase和Cassandra的数据模型都借鉴自Google 的BigTable。这种数据模型的特点是列式存储，每一行数据的各项被存储在不同的列中（这些列的集合称作列簇）。而每一列中每一个数据都包含一个时间戳属性，这样列中的同一个数据项的多个版本都能保存下来。

列式存储可以理解成这样，将行ID、列簇号，列号以及时间戳一起，组成一个Key，然后将Value按Key的顺序进行存储。Key值的结构化使这种数据结构能够实现一些特别的功能。最常用的就是将一个数据的多个版本存成时间戳不同的几个值，这样就能很方便的保存历史数据。这种结构也能天然地进行高效的松散列数据（在很多行中并没有某列的数据）存储。当然，另一方面，对于那些很少有某一行有NULL值的列，由于每一个数据必须包含列标识，这又会造成空间的浪费。

这些NoSQL系统对BigTable数据模型的实现多少有些差别，这其中以Cassandra进行的变更最为显著。Cassandra引入了超级列（supercolumn）的概念，通过将列组织到相应的超级列中，可以在更高层级上进行数据的组织，索引等。这一做法也取代了locality groups的概念（这一概念的实现可以让相关的几个行的数据存储在一起，以提高存取性能）。

13.2.2 图结构存储

图结构存储是NoSQL的另一种存储实现。图结构存储的一个指导思想是：数据并非对等的，关系型的存储或者键值对的存储，可能都不是最好的存储方式。图结构是计算机科学的基础结构之一，Neo4j和HyperGraphDB是当前最流行的图结构数据库。图结构的存储与我们之前讨论过的几种存储方式很不同，这种不同几乎体现在每一个方面，包括：数据模型、数据查询方式、数据在磁盘上的组织方式、在多个结点上的分布方式，甚至包括对事务机制的实现等等。由于篇幅的限制，对这些方面我们暂时不做深入的讨论，这里需要你了解的是，某些数据结构使用图结构的数据库进行存储可能会更好。

13.2.3 复杂查询

在NoSQL存储系统中，有很多比键值查找更复杂的操作。比如MongoDB可以在任意数据行上建立索引，可以使用Javascript语法设定复杂的查询条件。BigTable型的系统通常支持对单独某一行的数据进行遍历，允许对单列的数据进行按特定条件地筛选。CouchDB允许你创建同一份数据的多个视图，通过运行MapReduce任务来实现一些更为复杂的查询或者更新操作。很多NoSQL系统都支持与Hadoop或者其它一些MapReduce框架结合来进行一些大规模数据分析工作。

13.2.4 事务机制

传统的关系型数据库在功能支持上通常很宽泛，从简单的键值查询，到复杂的多表联合查询再到事务机制的支持。而与之不同的是，NoSQL系统通常注重性能和扩展性，而非事务机制。

传统的SQL数据库的事务通常都是支持ACID的强事务机制。A代表原子性，即在事务中执行多个操作是原子性的，要么事务中的操作全部执行，要么一个都不执行;C代表一致性，即保证进行事务的过程中整个数据加的状态是一致的，不会出现数据花掉的情况;I代表隔离性，即两个事务不会相互影响，覆盖彼此数据等;D表示持久化，即事务一量完成，那么数据应该是被写到安全的，持久化存储的设备上（比如磁盘）。

ACID的支持使得应用者能够很清楚他们当前的数据状态。即使如对于多个事务同时执行的情况下也能够保证数据状态的正常性。但是如果要保证数据的一致性，通常多个事务是不可能交叉执行的，这样就导致了可能一个很简单的操作需要等等一个复杂操作完成才能进行的情况。

对很多NoSQL系统来说，对性能的考虑远在ACID的保证之上。通常NoSQL系统仅提供对行级别的原子性保证，也就是说同时对同一个Key下的数据进行的两个操作，在实际执行的时候是会串行的执行，保证了每一个Key-Value对不会被破坏。对绝大多数应用场景来说，这样的保证并不会引起多大的问题，但其换来的执行效率却是非常可观的。当然，使用这样的系统可能需要我们在应用层的设计上多做容错性和修正机制的考虑。

在NoSQL中，Redis在事务支持这方面上不太一样，它提供了一个MULTI命令用来将多个命令进行组合式的操作，通过一个WATCH命令提供操作隔离性。这和其它一些提供test-and-set这样的低层级的隔离机制类似。

13.2.5 Schema-free的存储

还有一个很多NoSQL都有的共同点，就是它通常并没有强制的数据结构约束。即使是在文档型存储或者列式存储上，也不会要求某一个数据列在每一行数据上都必须存在。这在非结构化数据的存储上更方便，同时也省去了修改表结构的代价。而这一机制对应用层的容错性要求可能会更高。比如程序可能得确定如果某一个员工的信息里缺少lastname这一项，是否算是错误。或者说某个表结构的变更是否对所有数据行起了作用。还有一个问题，就是数据库的表结构可能会因为项目的多次迭代而变得混乱不堪。

13.3 数据可靠性

最理想状态是，数据库会把所有写操作立刻写到持久化存储的设备，同时复制多个副本到不同地理位置的不同节点上以防止数据丢失。但是这种对数据安全性的要求对性能是有影响的，所以不同的NoSQL系统在自身性能的考虑下，在数据安全上采取了不太一样的策略。

一种典型的出错情况是重启机器或者机器断电了，这时候需要让数据从内存转存到磁盘才能保证数据的安全性，因为磁盘数据不会因断电而丢失。而要避免磁盘损坏这种故障，就需要将数据冗余的存在其它磁盘上，比如使用RAID来做镜像，或者将数据同步到不同机器。但是有些时候针对同一个数据中心来说，是不可能做到完全的数据安全的，比如一旦发生飓风地震这种天灾，整个

机房机器可能都会损坏，所以，在这种情况下要保证数据的安全性，就必须将数据备份存储到其它地理位置上比较远的数据中心。所以，具体各个NoSQL系统在数据安全性和性能上的权衡策略也不太一样。

13.3.1 单机可靠性

单机可靠性理解起来非常简单，它的定义是写操作不会由于机器重启或者断电而丢失。通常单机可靠性的保证是通过把数据写到磁盘来完成的，而这通常会造成磁盘IO成为整个系统的瓶颈。而事实上，即使你的程序每次都把数据写到了磁盘，实际上由于操作系统buffer层的存在，数据还是不会立刻被写到物理磁盘上，只有当你调用fsync这个系统调用的时候，操作系统才会尽可能的把数据写到磁盘。

一般的磁盘大概能进行每秒100-200次的随机访问，每秒30-100MB的顺序写入速度。而内存在这两方面的性能都有数量级上的提升。通过尽量减少随机写，取而代之的对每个磁盘设备进行顺序写，这样能够减少单机可靠性保证的代价。也就是说我们需要减少在两次fsync调用之间的写操作次数，而增加顺序写操作的数量。下面我们说一下一些在单机可靠性的保证下提高性能的方法。

控制fsync的调用频率

Memcached是一个纯内存的存储，由于其不进行磁盘上的持久化存储，从而换来了很高的性能。当然，在我们进行服务器重启或者服务器意外断电后，这些数据就全丢了。因此Memcached 是一个非常不错的缓存，但是做不了持久化存储。

Redis则提供了几种对fsync调用频率的控制方法。应用开发者可以配置Redis在每次更新操作后都执行一次fsync，这样会比较安全，当然也就比较慢。Redis也可以设置成N秒种调用一次fsync，这样性能会更好一点。但这样的后果就是一旦出现故障，最多可能导致N秒内的数据丢失。而对一些可靠性要求不太高的场合（比如仅仅把Redis当Cache用的时候），应用开发者甚至可以直接关掉fsync的调用：让操作系统来决定什么时候需要把数据flush到磁盘。
（译者：这只是Redis append only file的机制，Redis是可以关闭aof日志的，另外请注意：Redis本身支持将内存中数据dump成rdb文件的机制，和上面说的不是一回事。）

使用日志型的数据结构

像B+ 树这样的一些数据结构，使得NoSQL系统能够快速的定位到磁盘上的数据，但是通常这些数据结构的更新操作是随机写操作，如果你在每次操作后再调用一次fsync，那就会造成频繁的磁

盘随机访问了。为了避免产生这样的问题，像Cassandra、HBase、Redis和Riak都会把写操作顺序的写入到一个日志文件中。相对于存储系统中的其它数据结构，上面说到的日志文件可以频繁的进行fsync操作。这个日志文件记录了操作行为，可以用于在出现故障后恢复丢失那段时间的数据，这样就把随机写变成顺序写了。

虽然有的NoSQL系统，比如MongoDB，是直接在原数据上进行更新操作的，但也有许多NoSQL系统是采用了上面说到的日志策略。Cassandra和HBase借鉴了BigTable的做法，在数据结构上实现了一个日志型的查找树。Riak也使用了类似的方法实现了一个日志型的hash表（译者：也就是Riak的BitCask模型）。CouchDB对传统的B+树结构进行了修改，使得对树的更新可以使用顺序的追加写操作来实现（译者：这种B+树被称作append-only B-Tree）。这些方法的使用，使得存储系统的写操作承载力更高，但同时为了防止数据异构后膨胀得过大，需要定时进行一些合并操作。

通过合并写操作提高吞吐性能

Cassandra有一个机制，它会把一小段时间内的几个并发的写操作放在一起进行一次fsync调用。这种做法叫group commit，它导致的一个结果就是更新操作的返回时间可能会变长，因为一个更新操作需要等就近的几个更新操作一起进行提交。这样做的好处是能够提高写操作承载力。作为HBase底层数据支持的Hadoop 分布式文件系统HDFS，它最近的一些补丁也在实现一些顺序写和group commit的机制。

13.3.2 多机可靠性

由于硬件层面有时候会造成无法恢复的损坏，单机可靠性的保证在这方面就鞭长莫及了。对于一些重要数据，跨机器做备份保存是必备的安全措施。一些NoSQL系统提供了多机可靠性的支持。

Redis采用了传统的主从数据同步的方式。所有在master上执行的操作，都会通过类似于操作日志的结构顺序地传递给slave上再执行一遍。如果master发生宕机等事故，slave可以继续执行完master传来的操作日志并且成为新的master。可能这中间会导致一些数据丢失，因为master同步操作到slave是非阻塞的，master并不知道操作是否已经同步线slave了。CouchDB 实现了一个类似的指向性的同步功能，它使得一个写操作可以同步到其它节点上。

MongoDB提供了一个叫Replica Sets的架构方制，这个架构策略使得每一个文档都会保存在组成Replica Sets的所有机器上。MongoDB提供了一些选项，让开发者可以确定一个写操作是否已经同步到了所有节点上，也可以在节点数据并不是最新的情况下执行一些操作。很多其它的分布式NoSQL存储都提供了类似的多机可靠性支持。由于HBase的底层存储是HDFS，它也就自然的获

得了HDFS提供的多机可靠性保证。HDFS的多机可靠性保证是通过把每个写操作都同步到两个以上的节点来实现的。

Riak、Cassandra和Voldemort提供了一些更灵活的可配置策略。这三个系统提供一个可配置的参数N，代表每一个数据会被备份的份数，然后还可以配置一个参数W，代表每个写操作需要同步到多少能机器上才返回成功。当然W是小于N的。

为了应对整个数据中心出现故障的情况，需要实现跨数据中心的多机备份功能。Cassandra、HBase和Voldemort都实现了一个机架位置可知的配置，这种配置方式使得整个分布式系统可以了解各个节点的地理位置分布情况。如果一个操作需要等待另外的数据中心的同步操作成功才返回给用户，那时间就太长了，所以通常跨数据中心的同步备份操作都是异步进行的。用户并不需要等待另一个数据中心同步的同步操作执行成功。

13.4 横向扩展带来性能提升

上面我们讨论了对出错情况的处理，下面我们讨论另一种情况：成功！如果数据存储操作成功执行了，那么你的系统就要负责对这些数据进行处理。就要承担数据带来的负载。一个比较粗暴的解决方法是通过升级你的机器来提升单机性能：通过加大内存和添加硬盘来应对越来越大的负载。但是随着数据量的增大，投入在升级机器上的钱将将不会产生原来那么大的效果。这时候你就必须考虑把数据同步到不同的机器上，利用横向扩展来分担访问压力。

横向扩展的目标是达到线性的效果，即如果你增加一倍的机器，那么负载能力应该也能相应的增加一倍。其主要需要解决的问题是如何让数据在多台机器间分布。数据分片技术实际上就是将对数据和读写请求在多个机器节点上进行分配的技术，分片技术在很多NoSQL系统中都有实现，比如Cassandra、HBase、Voldemort和Riak等等，最近MongoDB和Redis也在做相应的实现。而有的项目并不提供内置的分片支持，比如CouchDB更加注重单机性能的提升。对于这些项目，通常我们可以借助一些其它的技术在上层来进行负载分配。

下面让我们来整理一些通用的概念。对于数据分片和分区，我们可以做同样的理解。对机器，服务器或者节点，我们可以统一的理解成物理上的存储数据的机器。最后，集群或者机器环都可以理解成为是组成你存储系统的集合。

分片的意思是，没有任何一台机器可以处理所有写请求，也没有任何一台机器可以处理对所有数据的读请求。很多NoSQL系统都是基于键值模型的，因此其查询条件也基本上是基于键值的查询，基本不会有对整个数据进行查询的时候。由于基本上所有的查询操作都是基本键值形式的，因此分片通常也基于数据的键来做：键的一些属性会决定这个键值对存储在哪台机器上。下面我

们将会对hash分片和范围分片两种分片方式进行描述。

13.4.1 如非必要，请勿分片

分片会导致系统复杂程序大增，所以，如果没有必要，请不要使用分片。下面我们先讲两种不用分片就能让系统具有扩展性的方法。

读写分离

大多数应用场景都是读多写少的场景。所以在这种情况下，可以用一个简单的方法来分担负载，就是把数据同步到多台机器上。这时候写请求还是由master机器处理，而读请求则可以分担给那些同步到数据的机器了。而同步数据的操作，通常是不会对master带来多大的压力的。

如果你已经使用了主从配置，将数据同步到多台机器以提供高可靠性了，那么你的slave机器应该能够为master分担不少压力了。对有些实时性要求不是非常高的查询请求，比如一些统计操作，你完全可以放到slave上来执行。通常来说，你的应用对实时性要求越低，你的slave机器就能承担越多的任务。

使用缓存

将一些经常访问的数据放到缓存层中，通常会带来很好的效果。Memcached 主要的作用就是将数据层的数据进行分布式的缓存。Memcached 通过客户端的算法（译者： 常见的一致性hash算法）来实现横向扩展，这样当你想增大你缓存池的大小时，只需要添加一台新的缓存机器即可。

由于Memcached仅仅是一个缓存存储，它并不具备一些持久存储的复杂特性。当你在考虑使用复杂的扩展方案时，希望你先考虑一下使用缓存来解决你的负载问题。注意，缓存并不是临时的处理方案：Facebook 就部署了总容量达到几十TB的Memcahced内存池。

通过读写分离和构建有效的缓存层，通常可以大大分担系统的读负载，但是当你的写请求越来越频繁的时候，你的master机器还是会承受越来越大的压力。对于这种情况，我们可能就要用到下面说到的数据分片技术了。

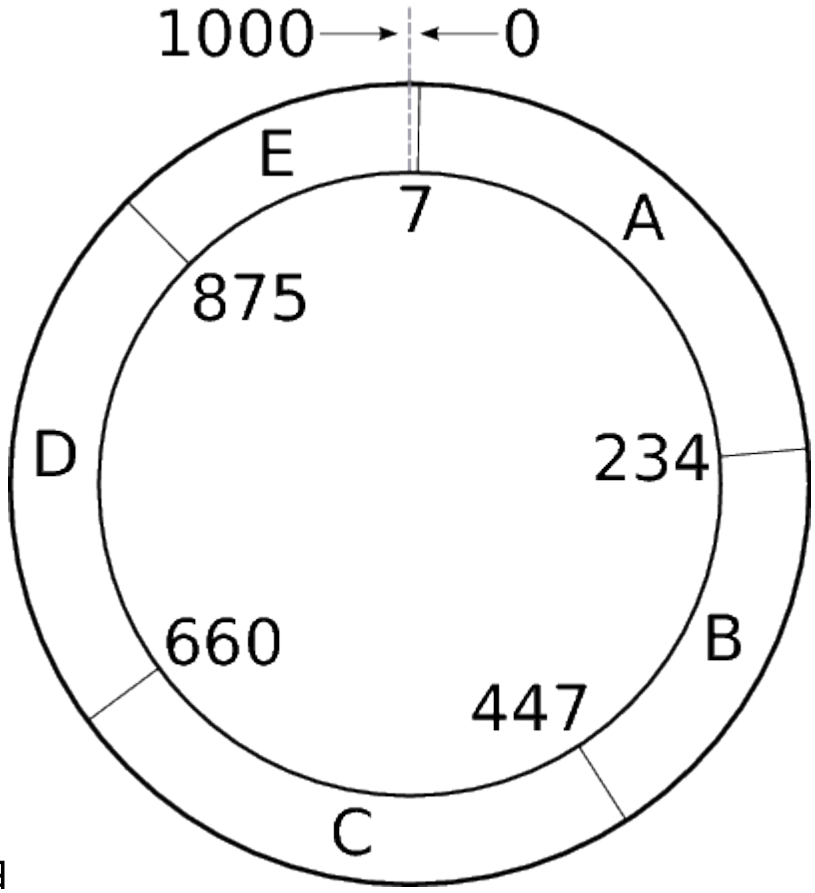
13.4.2 通过协调器进行数据分片

由于CouchDB专注于单机性能，没有提供类似的横向扩展方案，于是出现了两个项目：Lounge 和 BigCouch，他们通过提供一个proxy层来对CouchDB中的数据进行分片。在这种架构中，proxy作为CouchDB集群的前端机器，接受和分配请求到后端的多台CouchDB上。后端的 CouchDB 之间并没有交互。协调器会将按操作的key值将请求分配到下层的具体某台机器。

Twitter 自己实现了一个叫Gizzard的协调器，可以实现数据分片和备份功能。Gizzard不关心数据类型，它使用树结构来存储数据范围标识，你可以用它来对SQL或者NoSQL系统进行封装。通过对 Gizzard 进行配置，可以实现将特定范围内的数据进行冗余存储，以提高系统的容灾能力。

13.4.3 一致性hash环算法

好的hash算法可以使数据保持比较均匀的分布。这使得我们可以按这种分布将数据保存布多台机器上。一致性hash是一种被广泛应用的技术，其最早在一个叫distributed hash tables (DHTs)的系统中进行使用。那些类Dynamo的应用，比如Cassandra、Voldemort和Riak，基本上都使用了一致性hash算法。



Hash环图

一致性hash算法的工作原理如下：首先我们有一个hash函数H，可以通过数据的key值计算出一个数字型的hash值。然后我们将整个hash环的范围定义为［1，L］这个区间，我们将刚才算出的hash值对L进行取余，就能算出一个key值在这个环上的位置。而每一台真实服务器结点就会负责［1-L］之间的某个区间的数据。如上图，就是一个五个结点的hash环。

上面hash环的L值为1000，然后我们对ABCDE 5个点分别进行hash运算，H(A) mod L = 7, H(B) mod L = 234, H(C) mod L = 447, H(D) mod L = 660, and H(E) mod L = 875 ，这样，hash值在7-233之间的所有数据，我们都让它保存在A节点上。在实际动作中，我们对数据进行hash，算出其应该在哪个节点存储即可，例：H('employee30') mod L = 899 那么它应该在E节点上，H('employee31') mod L = 234 那么这个数据应该在B节点上。

备份数据

一致性hash下的数据备份通常采用下面的方法：将数据冗余的存在其归属的节点的顺序往下的节点，例如你的冗余系数为3（即数据会在不同节点中保存三份），那么如果通过hash计算你的数据在A区间［7，233］，你的数据会被同时保存在A，B，C三个节点上。这样如果A节点出现故障，那么B，C节点就能处理这部分数据的请求了。而某些设计会使E节点将自己的范围扩大到A233，以接受对出故障的A节点的请求。

优化的数据分配策略

虽然hash算法能够产生相对均匀的hash值。而且通常是节点数量越多，hash算法会越平均的分配key值。然而通常在项目初期不会有太多的数据，当然也不需要那么多的机器节点，这时候就会造成数据分配不平均的问题。比如上面的5个节点，其中A节点需要负责的hash区间范围大小为227，而E节点负责的区间范围为132。同时在这种情况下，出故障后数据请求转移到相邻节点的策略也可能不好实施了。

为了解决由于节点比较少导致数据分配不均的问题，很多DHT系统都实现了一种叫做虚拟节点的技术。例如4个虚拟节点的系统中，A节点可能被虚拟化成A_1，A_2，A_3，A_4这四个虚拟节点，然后对这四个虚拟节点再进行hash运算，A节点负责的key值区间就比较分散了。Voldemort使用了与上面类似的策略，它允许对虚拟节点数进行配置，通常这个节点数会大于真实节点数，这样每个真实节点实际上是负责了N个虚拟节点上的数据。

Cassandra 并没有使用虚拟节点到真实节点映射的方法。这导致它的数据分配是不均匀的。为了解决这种不平衡，Cassandra 利用一个异步的进程根据各节点的历史负载情况来调节数据的分

布。

13.4.4 连续范围分区

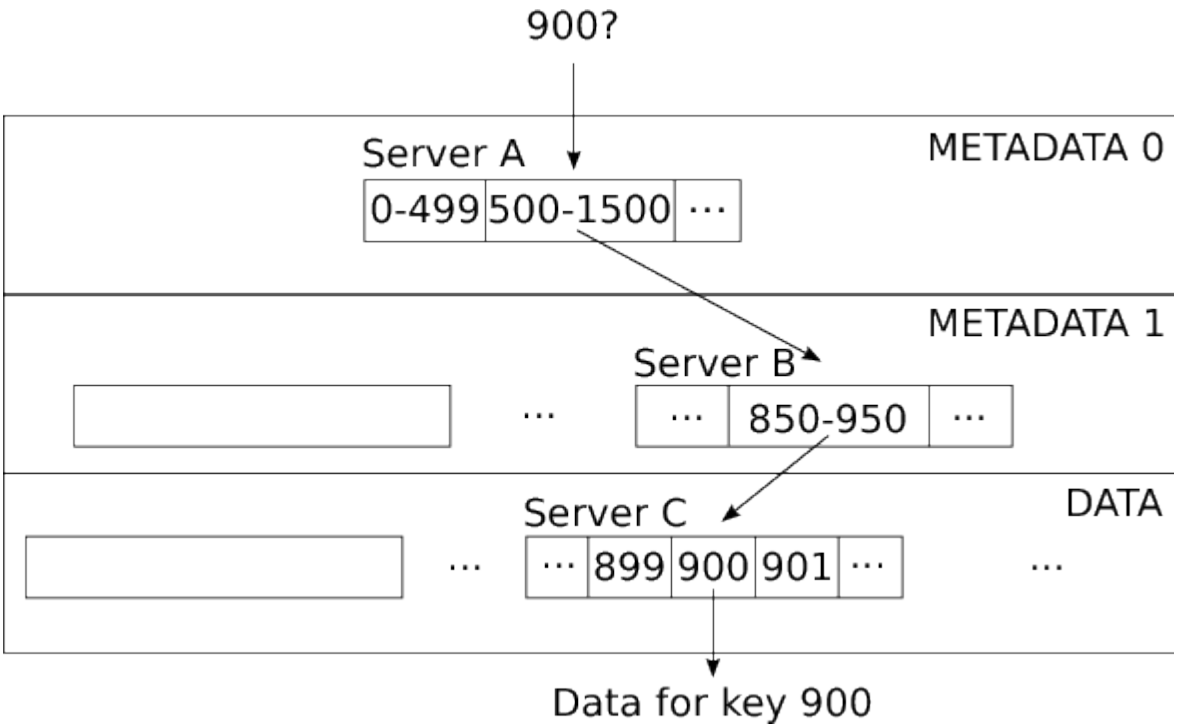
使用连续范围分区的方法进行数据分片，需要我们保存一份映射关系表，标明哪一段key值对应存在哪台机器上。和一致性hash类似，连续范围分区会把key值按连续的范围分段，每段数据会被指定保存在某个节点上，然后会被冗余备份到其它的节点。和一致性hash不同的是，连续范围分区使得key值上相邻的两个数据在存储上也基本上是在同一个数据段。这样数据路由表只需记录某段数据的开始和结束点 [start，end] 就可以了。

通过动态调整数据段到机器结点的映射关系，可以更精确的平衡各节点机器负载。如果某个区段的数据负载比较大，那么负载控制器就可以通过缩短其所在节点负责的数据段，或者直接减少其负责的数据分片数目。通过添加这样一个监控和路由模块，使我们能够更好的对数据节点进行负载均衡。

BigTable的处理方式

Google BigTable 论文中描述了一种范围分区方式，它将数据切分成一个个的tablet数据块。每个tablet保存一定数量的键值对。然后每个Tablet 服务器会存储多个tablet块，具体每个Tablet服务器保存的tablet数据块数，则是由服务器压力来决定的。

每个tablet大概100-200MB大。如果tablet的尺寸变小，那么两个tablet可能会合并成一个tablet，同样的如果一个tablet过大，它也会被分裂成两个tablet，以保持每个tablet的大小在一定范围内。在整个系统中有一个master机器，会根据tablet的大小、负载情况以及机器的负载能力等因素动态地调整tablet在各个机器上的分布。



master服务器会把 tablet 的归属关系存在元数据表里。当数据量非常大时，这个元数据表实际也会变得非常大，所以归属关系表实际上也是被切分成一个个的tablet保存在tablet服务器中的。这样整个数据存储就被分成了如上图的三层模型。

下面我们解释一下上面图中的例子。当某个客户端要从BigTable系统中获取key值为900的数据时，首先他会到第一级元数据服务器A（METADATA0）去查询，第一级元数据服务器查询自己的元数据表，500-1500这个区间中的所有元数据都存在B服务器中，于是会返回客户端说去B服务器查询，客户端再到B服务器中进行查询，B服务器判断到850-950这个区间中的数据都存在tablet服务器C中，于是会告知客户端到具体的tablet服务器C去查询。然后客户端再发起一次向C服务器的请求，就能获取到900对应的数据了。然后，客户端会把这个查询结果进行缓存，以避免对元数据服务器的频繁请求。在这样的三层架构下，只需要128MB的元数据存储，就能定位234 个tablet数据块了（按128MB一个数据块算，是261bytes的数据）。

故障处理

在BigTable中，master机器是一个故障单点，不过系统可以容忍短时间的master故障。另一方面，如果tablet 服务器故障，那么master可以把对其上tablet的所有请求分配到其它机器节点。

为了监测和处理节点故障，BigTable实现了一个叫Chubby的模块，Chubby是一个分布式的锁系统，用于管理集群成员及检测各成员是否存活。ZooKeeper是Chubby的一个开源实现，有很多基于 Hadoop 的项目都使用它来进行二级master和tablet节点的调度。

基于范围分区的NoSQL项目

HBase 借鉴了BigTable的分层理论来实现范围分区策略。tablet相关的数据存在HDFS里。HDFS会处理数据的冗余备份，并负责保证各备份的一致性。而像处理数据请求，修改存储结构或者执行tablet的分裂和合并这种事，是具体的tablet服务器来负责的。

MongoDB也用了类似于BigTable的方案来实现范围分区。他用几台配置机器组成集群来管理数据在节点上的分布。这几台机器保存着一样的配置信息，他们采用 two-phase commit 协议来保证数据的一致性。这些配置节点实际上同时扮演了BigTable中的master的路由角色，及Chubby 的高可用性调度器的角色。而MongoDB具体的数据存储节点是通过其Replica Sets方案来实现数据冗余备份的。

Cassandra 提供了一个有序的分区表，使你可以快速对数据进行范围查询。Cassandra也使用了一致性hash算法进行数据分配，但是不同的是，它不是直接按单条数据进行hash，而是对一段范围内的数据进行hash，也就是说20号数据和21号数据基本上会被分配在同一台机器节点上。

Twitter的Gizzard框架也是通过使用范围分区来管理数据在多个节点间的备份与分配。路由服务器可以部署成多层，任何一层只负责对key值进行按范围的分配到下层的不同节点。也就是说路由服务器的下层既可以是真实的数据存储节点，也可能是下层的路由节点。Gizzard的数据备份机制是通过将写操作在多台机器上执行多次来实现的。Gizzard的路由节点处理失败的写操作的方式和其它NoSQL不太一样，他要求所有更新都是幂等的（意思是可以重复执行而不会出错）。于是当一个节点故障后，其上层的路由节点会把当前的写操作cache起来并且重复地让这个节点执行，直到其恢复正常。

13.4.5 选择哪种分区策略

上面我们说到了Hash分区和范围分区两种策略，哪种更好呢？这要看情况了，如果你需要经常做范围查询，需要按顺序对key值进行操作，那么你选择范围分区会比较好。因为如果选择hash分区的话，要查询一个范围的数据可能就需要跨好几个节点来进行了。

那如果我不会进行范围查询或者顺序查询呢？这时候hash分区相对来说可能更方便一点，而且hash分区时可能通过虚拟结点的设置来解决hash不均的问题。在hash分区中，基本上只要在客户端执行相应的hash函数就能知道对应的数据存在哪个节点上了。而如果考虑到节点故障后的数据转移情况，可能获取到数据存放节点就会麻烦一些了。

范围分区要求在查询数据前对配置节点还要进行一次查询，如果没有特别好的高可用容灾方案，配置节点将会是一个危险的故障单点。当然，你可以把配置节点再进行一层负载均衡来减轻负载。而范围分区时如果某个节点故障了，它上面的数据可以被分配到多个节点上，而不像在一致性hash时，只能迁移到其顺序的后一个节点，造成下一个节点的负载飙升。

13.5 一致性

上面我们讲到了通过将数据冗余存储到不同的节点来保证数据安全和减轻负载，下面我们来看看这样做引发的一个问题：保证数据在多个节点间的一致性是非常困难的。在实际应用中我们会遇到很多困难，同步节点可能会故障，甚至会无法恢复，网络可能会有延迟或者丢包，网络原因导致集群中的机器被分隔成两个不能互通的子域等等。在NoSQL中，通常有两个层次的一致性：第一种是强一致性，既集群中的所有机器状态同步保持一致。第二种是最终一致性，既可以允许短暂的数据不一致，但数据最终会保持一致。我们先来讲一下，在分布式集群中，为什么最终一致性通常是更合理的选择，然后再来讨论两种一致性的具体实现结节。

13.5.1 关于CAP理论

为什么我们会考虑削弱数据的一致性呢？其实这背后有一个关于分布式系统的理论依据。这个理论最早被 Eric Brewer 提出，称为CAP理论，尔后Gilbert 和 Lynch 对CAP进行了理论证明。这一理论首先把分布式系统中的三个特性进行了如下归纳：

- 一致性（C）：在分布式系统中的所有数据备份，在同一时刻是否同样的值。
- 可用性（A）：在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。
- 分区容忍性（P）：集群中的某些节点在无法联系后，集群整体是否还能继续进行服务。

而CAP理论就是说在分布式存储系统中，最多只能实现上面的两点。而由于当前的网络硬件肯定会出现延迟丢包等问题，所以分区容忍性是我们必须需要实现的。所以我们只能在一致性和可用性之间进行权衡，没有NoSQL系统能同时保证这三点。

要保证数据一致性，最简单的方法是令写操作在所有数据节点上都执行成功才能返回成功。而这时如果某个结点出现故障，那么写操作就成功不了了，需要一直等到这个节点恢复。也就是说，如果要保证强一致性，那么就无法提供7×24的高可用性。

而要保证可用性的话，就意味着节点在响应请求时，不用完全考虑整个集群中的数据是否一致。

只需要以自己当前的状态进行请求响应。由于并不保证写操作在所有节点都写成功，这可能会导致各个节点的数据状态不一致。

CAP理论导致了最终一致性和强一致性两种选择。当然，事实上还有其它的选择，比如在Yahoo!的PNUTS中，采用的就是松散的一致性和弱可用性结合的方法。但是我们讨论的NoSQL系统没有类似的实现，所以我们在后续不会对其进行讨论。

13.5.2 强一致性

强一致性的保证，要求所有数据节点对同一个key值在同一时刻有同样的value值。虽然实际上可能某些节点存储的值是不一样的，但是作为一个整体，当客户端发起对某个key的数据请求时，整个集群对这个key对应的数据会达成一致。下面就举例说明这种一致性是如何实现的。

假设在我们的集群中，一个数据会被备份到N个结点。这N个节点中的某一个可能会扮演协调器的作用。它会保证每一个数据写操作会在成功同步到W个节点后才向客户端返回成功。而当客户端读取数据时，需要至少R个节点返回同样的数据才能返回读操作成功。而NWR之间必须要满足下面关系：R + W>N

下面举个实在的例子。比如我们设定N = 3（数据会备份到A、B、C三个结点）。比如值employee30:salary 当前的值是20000，我们想将其修改为30000。我们设定W = 2，下面我们会对A、B、C三个节点发起写操作（employee30:salary, 30000），当A、B两个节点返回写成功后，协调器就会返回给客户端说写成功了。至于节点C，我们可以假设它从来没有收到这个写请求，他保存的依然是20000那个值。之后，当一个协调器执行一个对employee30:salary的读操作时，他还是会发三个请求给A、B、C三个节点：

- 如果设定R = 1，那么当C节点先返回了20000这个值时，那我们客户端实际得到了一个错误的值。
- 如果设定R = 2，则当协调器收到20000和30000两个值时，它会发现数据不太正确，并且会在收到第三个节点的30000的值后判断20000这个值是错误的。

所以如果要保证强一致性，在上面的应用场景中，我们需要设定R = 2，W = 2

如果写操作不能收到W个节点的成功返回，或者写操作不能得到R个一致的结果。那么协调器可能会在某个设定的过期时间之后向客户端返回操作失败，或者是等到系统慢慢调整到一致。这可能就导致系统暂时处于不可用状态。

对于R和W的不同设定，会导致系统在进行不同操作时需要不同数量的机器节点可用。比如你设定在所有备份节点上都写入才算写成功，既W = N，那么只要有一个备份节点故障，写操作就失

败了。一般设定是R + W = N+1，这是保证强一致性的最小设定了。一些强一致性的系统设定W = N，R = 1，这样就根本不用考虑各个节点数据可能不一致的情况了。

HBase是借助其底层的HDFS来实现其数据冗余备份的。HDFS采用的就是强一致性保证。在数据没有完全同步到N个节点前，写操作是不会返回成功的。也就是说它的W = N，而读操作只需要读到一个值即可，也就是说它R = 1。为了不至于让写操作太慢，对多个节点的写操作是并发异步进行的。在直到所有的节点都收到了新的数据后，会自动执行一个swap操作将新数据写入。这个操作是原子性和一致性的。保证了数据在所有节点有一致的值。

13.5.3 最终一致性

像Voldemort，Cassandra和Riak这些类Dynamo的系统，通常都允许用户按需要设置N，R，W三个值，即使是设置成W + R<= N也是可以的。也就是说他允许用户在强一致性和最终一致性之间自由选择。而在用户选择了最终一致性，或者是W<N的强一致性时，则总会出现一段各个节点数据不同步导致系统处理不一致的时间。为了提供最终一致性的支持，这些系统会提供一些工具来使数据更新被最终同步到所有相关节点。

下面我们会先讲一讲如何判断数据是最新的还是陈旧的，然后我们再讨论一下如何进行数据同步，最后我们再列举一些Dynamo里使用的加速同步过程的巧妙方法。

版本控制与冲突

由于同一份数据在不同的节点可能存在不同值，对数据的版本控制和冲突监测就变得尤为重要。类Dynamo的系统通常都使用了一种叫vector clock（向量时钟）的版本控制机制。一个vector clock可以理解成是一个向量，它包含了这个值在每一个备份节点中修改的次数。比如说有的数据会备份到A，B，C三个节点，那么这些值的vector clock值就是类似（NA，NB，NC），而其初始值为（0，0，0）。

每次一个key的值被修改，其vector clock相应的值就会加1。比如有一个key值当前的vector clock值为（39，1，5），然后他在B节点被修改了一次，那么它的cector clock值就会相应的变成（39，2，5）。而当另一个节点C在收到B节点的同步请求时，他会先用自己保存的vector clock值与B传来的vector clock值进行对比，如果自己的vector clock值的每一项都小于等于B传来的这个值，那么说明这次的修改值是在自己保存的值上的修改，不会有冲突，直接进行相应的修改，并把自己的vector clock值更新。但是如果C发现自己的vector clock有些项比B大，而某些项比B小，比如B的是（39，2，5）C的是（39，1，6），那么这时候说明B的这次修改并不是在C的基础上改的，数据出现冲突了。

冲突解决

不同的系统有不同的冲突解决策略。Dynamo选择把冲突留给应用层来解决。如果是两个节点保存的购物车信息冲突了，可以选择简单的通过把两个数据合并进行解决。但如果是对同一份文档进行的修改冲突了，可能就需要人工来解决冲突了（译者：像我们在SVN中做的一样）。Voldemort就是采用的后者，它在发现冲突后，会把有冲突的几份数据一起返回给应用层，把冲突解决留给应用层来做。

Cassandra通过为每一个操作保存一个时间戳的方法来解决冲突，在冲突的几个版本里，最后修改的一个会获胜成为新的值。相对于上面的方式，它减少了通过应用层解决冲突时需要的网络访问，同时也简化了客户端的操作API。但这种策略并不适合用来处理一些需要合并冲突的场合，比如上面的购物车的例子，或者是分布式的计数器这样的应用。而Riak把Voldemort和Cassandra 的策略都实现了。CouchDB会把冲突的key进行标识，以便应用层可以主动进行人工修复，在修复完成前，客户端的请求是无法得到确定值的。

读时修复

在数据读取时，如果有R个节点返回了一致的数据，那么协调器就可以认为这个值是正确的并返回给客户端了。但是在总共返回的N个值中，如果协调器发现有的数据不是最新的。那么它可以通过读时修复机制来对这些节点进行处理。这种方式在Dynamo中有描述，在Voldemort、Cassandra和Riak中都得到了实现。当协调器发现有的节点数据不是最新时，它会在数据不一致的节点间启动一个冲突解决过程。这样主动的修复策略并不会有多大的工作量。因为读取操作时，各个节点都已经把数据返回给协调器了，所以解决冲突越快，实际上可能造成的后续的不一致的可能性也就越小。

Hinted Handoff

Cassandra、Riak和Voldemort都实现了一种叫Hinted Handoff的技术，用来保证在有节点故障后系统的写操作不受太大影响。它的过程是如果负责某个key值的某个节点宕机了，另一个节点会被选择作为其临时切换点，以临时保存在故障节点上面的写操作。这些写操作被单独保存起来，直到故障节点恢复正常，临时节点会把这些写操作重新迁移给刚刚恢复的节点。Dynamo 论文中提到一种叫“sloppy quorum”的方法，它会把通过 Hinted Handoff 写成功的临时节点也计算在成功写入数中。但是Cassandra和Voldemort并不会将临时节点也算在写入成功节点数内，如果写入操作并没有成功写在W个正式节点中，它们会返回写入失败。当然，Hinted Handoff 策略在这些系统中也有使用，不过只是用在加速节点恢复上。

Anti-Entropy

如果一个节点故障时间太长，或者是其 Hinted Handoff 临时替代节点也故障了，那么新恢复的节点就需要从其它节点中同步数据了。（译者：实际上就是要找出经过这段时间造成的数据差异，并将差异部分同步过来）。这种情况下Cassandra和Riak都实现了在Dynamo文档中提到的一种方法，叫做anti-entropy。在anti-entropy过程中，节点间通过交换Merkle Tree来找出那些不一致的部分。Merkle Tree是一个分层的hash校验机制：如果包含某个key值范围的hash值在两个数据集中不相同，那么不同点就在这个key值范围，同理，如果顶层的hash值相同，那么其负责的所有key值范围内的值都认为是相同的。这种方法的好处是，在节点恢复时，不用把所有的值都传一遍来检查哪些值是有变化的。只需要传几个hash值就能找到不一致的数据，重传这个数据即可。

Gossip

当一个分布式系统越来越大，就很难搞清集群中的每个节点的状态了。上面说到的类Dynamo 应用都采用了Dynamo文档中说到的一种古老的方法：Gossip。通过这个方法，节点间能够互相保持联系并能够检测到故障节点。其具体做法是，每隔一段时间（比如一秒），一个节点就会随便找一个曾经有过通信的节点与其交换一下其它节点的健康状态。通过这种方式，节点能够比较快速的了解到集群中哪些节点故障了，从而把这些节点负责的数据分配到其它节点去。（译者：Gossip其实是仿生学的设计，Gossip意思为流言，节点传播其它节点的健康信息，就像一个小村镇里的无聊妇人们互相说别人的闲话一样，基本上谁家谁人出什么事了，都能比较快地被所有人知道）。

13.6 写在最后的话

目前NoSQL系统来处在它的萌芽期，我们上面讨论到的很多NoSQL系统，他们的架构、设计和接口可能都会改变。本章的目的，不在于让你了解这些NoSQL系统目前是如何工作的，而在于让你理解这些系统之所以这样实现的原因。NoSQL系统把更多的设计工作留给了应用开发工作者来做。理解上面这些组件的架构，不仅能让您写出下一个NoSQL系统，更让您对现有系统应用得更好。

13.7 致谢

非常感谢Jackie Carter, Mihir Kedia，以及所有对本章进行校对并且提出宝贵意见的人。没有这些年来NoSQL社区的专注工作，也不会有这一章内容的诞生。各位加油。

相关信息

原文: http://www.aosabook.org/en/nosql.html
原作者：Adam Marcus
译者：iammutex
组织：NoSQLFan
翻译时间：2011年6月