​今天有人问我有没有数据存储的相关资料，我想了想。。虽然在这个领域内也算有点积累，以前讲课的ppt有200多页，但毕竟ppt的信息量有限。所以在这里将这个系列的部分内容在这里进行重新编排。这个内容会在我的轻博客和这里同步龟速更新。

主要将涉及到：

1. 数据库原理

   关系代数

   事务l

   sql

   k-v存储的使用

   数据存储模型

   数据写入模式性能和安全性分析

2. 倒排索引

3. 分布式kv系统

   数据的切分

   数据的管理和扩容

   数据安全性

   读写可用性

4. 硬件存储在淘宝的测试数据和分析

5. 淘宝在线数据存储检索经验介绍

那么 在走进海量存储与检索的世界之前，我们先看一看目前似乎觉得最落伍的数据库系统。丑话先说。。我也没在这个领域沉浸几年，所以其实里面很多的概念也是有可能模糊的，所以在这里写出来，一是希望我能把以前的积累再次重新的梳理一次，查缺补漏。二也是在这世界留下点记录。。表明我曾经来到过这世界，学到过这些东西。。

之所以从这里开始，一部分的原因是我本身是从这里开始接触存储这个领域的，另外一部分原因是因为，从我的私心来说，我认为数据库是生产力工具的开始，也将是生产力工具最终的归宿。

一个数据库，我们可以抽象的认为由下面的一个逻辑结构组成，刨除意义不大的视图，存储过程，外键限制等之后，我们就剩下了下面的这张图：



从API来说，也就是SQL,结构化查询语言，这个东东我们后面再去细说，先来看看这个关系代数模型。

之所以要从这里开始，主要的原因是因为，这是最受到关注的一个部分，自大从一开始做分布式数据层开始，被人问得最多的问题就是：1. 切分以后如何做join。2.如何进行分布式事务。。

可惜，现在我也没有一个方法能做到100%让您满意。。因为，没有银弹，只有取舍。

取舍的原则，也就是要根据，1） 你能做什么。2）你需要做什么。3）你能放弃什么。来决定上层系统的整体架构。传统的ACID的关系代数模型，在新的环境中，很难复用。但，原理没有变，做法上可以做微调

不知道各位在想起关系代数的时候，会想到什么？

<http://www.bitscn.com/os/windows/200604/4364.html> 我给大家一些解释（当然我没仔细看过:）

不爱看上面解释的童鞋们呢，也不用纠结，给大家一些简单的例子。

1. select \* from tab where user = ?

这就是最简单的关系代数的例子，我们对他进行抽象来说，其实就是将大量数据中的一小批数据，按照某个要求查出来的过程。

2. select ename,dname from emp,deptwhere emp.deptno=dept.deptno and emp.deptno=30

这是另外一个例子，本质来说就是个∩的操作。

3. select count(\*) from tab.

这是关系代数的第二个主要的用处，就是进行统计和计算，这类的函数有个专用的名字，叫做aggragate function.

恩，基本上有这个概念就可以了。

那么这里请联想，你所接触到的什么地方会有碰到有这样的一些计算呢？

数据库？对。不过不是废话么。。

还有就是hadoop的平台也会用到，hive pig.

那么，我们对这类关系代数进行一下简单的概念上的小结。

也就是用于处理数据的一类方法的抽象，最主要的作用是，按照某个条件选出一批数据，然后再进行一些简单的统计计算的功能。如是而已:)

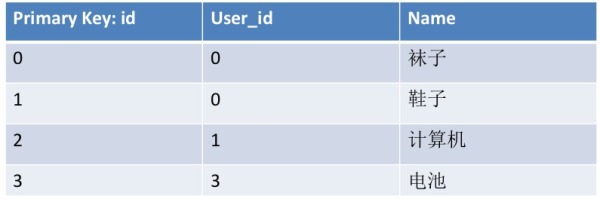
那么，下面就有个疑问了，下层的数据库或存储，是如何进行这类的计算的呢？

那么请看下一个章节。

我们一起探寻一下，处理这类关系代数计算的方法有哪些

在上一篇里面，我们对数据库的抽象的组成原理进行了简单的描述。在这一篇里面，我们一起来看看，如何能够使用kv这样的工具。来完成关系代数运算。

那么，让我们先来热热身:



这是一组数据，以pk作为主键，user\_id和Name是外key.

那么，如果我要运行查询:Select \* from tab where id = ?

应该如何进行呢？

这里需要一些额外的知识，在数据结构中，有那么一种结构，可以用于处理按照某个key找到value的过程，抽象来看，一种方法是二分查找法，一种方法是hash.

如果各位是java用户，那么二分查找的实现可以认为是个TreeMap的实现，而Hash的方法则可以认为是hashMap的实现。如果是个c/cpp的用户，那么就二分查找就对应map实现。而hash实现则对应stl里面的hash\_map。

那么，这里的这个问题，我们就很容易可以解决了

以id作为map的key,以其他数据作为value，把所有数据都放入到map里面，然后再使用id=1作为key,从map中找到对应的value返回即可。（这一个部分，我们在后面的章节里面还会介绍，现在大家只需要有个大概的印象即可）

怎么样？是不是很简单？那么，我们来讨论更进一步的问题：

如果我想找到符合Select \* from tab where user\_id = 0的所有结果，应该如何去作？

仔细想想。那么第一种做法一定是这样。

把整个集合内的所有数据，都拿出来，然后找到user\_id的数字，如果user\_id=0，那么就认为是符合要求的记录，直接返回。

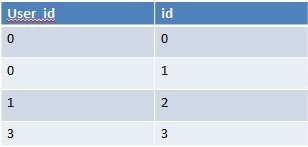
如果不是user\_id=0，那么不匹配，丢弃这条记录即可。

这样一定可以找到所有符合要求的记录。

然而，这样作，带来的问题是，我有多少条记录，就需要进行多少次这样的匹配，那么，假设有100000000000000000条记录，就需要匹配这样多次，才能找到符合要求的记录。这是个悲剧。。

那么，怎么解决这个悲剧呢？

于是有些聪明人就又想起了map结构，hash或tree，不都可以按照k找到value么。那我们这里也可以利用这个map结构嘛。。



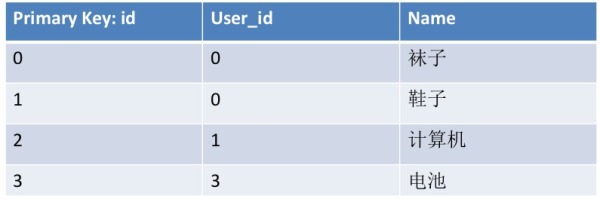
也就是说，以user\_id作为key,id作为value，构建一个Map.不就又能进行快速查询了么。

于是，就有了数据库最重要的一个结构“索引” 这种以外键作为key,主键作为value的东西，有个专有的名字，叫做二级索引。

有了二级索引，我们的所有查询，都可以以接近O(LogN)(有序数据)，或O(1)的效率找到我们需要的数据。是不是很爽？

但这不是银弹，你付出了空间成本，本质来说就是空间换时间的过程。同时，也会降低写入的效率。

怎么样？理解了没？如果自认为对这些都了解了，那么我们再来看一个问题：



如果我要找的是：Select …where user\_id = ? And name = ‘袜子’

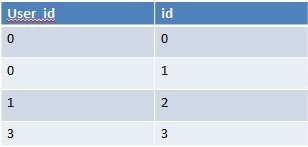
应该怎么做呢？

估计很多人都立刻又会想起那个Map，对的，但在这里，我想给出以下的几种查询的模式：

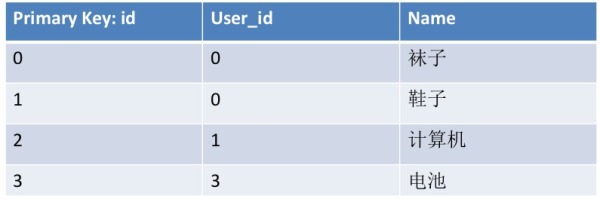
1. 遍历所有数据，取出一条以后，查看user\_id = 0 and name=’袜子’是否符合要求，如果符合，则返回数据。

这是个合理的策略，空间最为节省，但带来的损耗是要遍历所有的数据。

2. 如果有个user\_id -> pk的索引



，那么我们可以先按照user\_id，找到一组符合要求的pk list.然后再根据pk list，再回到



取出符合要求的数据后，判断name=‘袜子’这个条件，如果符合，就返回，不符合，就丢弃。

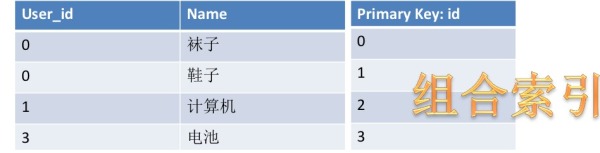
这是个折衷策略，在空间和性能中，尽可能的找到个合理的区间的策略。

题外话，这个“根据pk list，再回到pk=>整个数据的kv表中，找出符合要求的数据后，判断name=‘袜子’这个条件，如果符合，就返回，不符合，就丢弃”的策略，在数据库有个专有名词，叫回表。

3. 组合索引

这是个新名词儿，但其实也是个很简单的概念。

直接上图：



:-) ，其实就是个很简单的策略，先比较user\_id进行排序，如果user\_id相同，那么比较name排序。

这样，假定我们有100000条记录，属于100个用户，那么平均来看，每个用户就只有1000条记录了。

原来要回表1000条记录才能找到符合要求的数据，而如果使用组合索引，这1000条，也可以使用O(log2N)或者O(1)的策略进行检索啦。

在很多场景中，都能够提升效率和速度。但付出的是更多的存储空间。

好啦，这篇就介绍到这里，留个题目给大家：



假设有这么一组数据，性别有4种，user\_id是一对多的关系，如果我想查询

select \* from tab where user\_id in (?,?,?,?) and 性别=’不明’

如何进行索引构建能够获得比较好的效果呢？

首先是回答上次的问题。



假设有这么一组数据，性别有4种，user\_id是一对多的关系，如果我想查询

select \* from tabwhere user\_id in (?,?,?,?) and 性别=’不明’

如何进行索引构建能够获得比较好的效果呢？

我个人认为，应该建立的是以user\_id作为前导列，性别作为辅助列的索引，在大量单值查询时会有优势。

理由如下

1. 假定总数据量为N,user\_id的区分度为N/10000 而性别的区分度为N/4

那么如果以user\_id作为前导列，性别作为后列，那么查询的复杂度为O(logN+log(N/10000))。也就是说，第一次二分查找之后，下一次是在第一次的二分查找的基础上再去查找。而如果以性别作为前导，user\_id作为后列，那么复杂度为

O(logN+log(N/4));

效率略差。

然后进入本次正题。上次介绍了关系模型，那么这次我们来介绍一下事务。

在一切之前，我想先给自己解嘲一下。。事务我自己也没有办法完全融汇贯通，因为每一个小的选择，都会导致效果的完全不同，所以有错请在后面一起探讨。

那么我们在这里，主要以单机事务作为入手点，然后阐述一下多机事务相关的知识点。我在这里只是想做一个引导，让大家能够对整个的知识体系有一个基本的认识，对于细节问题，会给出一些资料，而不会直接去进行讲解，因为篇幅所限.

一般来说，我们一提起事务，就会想到数据库，确实，事务是数据库的最重要的一个属性。但这似乎不是事务的本源，那么，让我们从更深层次去对事务进行一次思考吧：

事务，本质来说就是一组由一个人（或机器）发起的连续的逻辑操作，共同的完成一件事情，在完成整个事情之前，其所有的改动，都不应该对其他人可见和影响。而在事务结束之后，其一切的改动，都必须“全部”“立刻”对其他的人（或机器）可见。

然后，人们为了描述这一运作,使用了四个词汇，这也是很多面试的同学们折戟沉沙之处。J 不过这个以前我也不会，后来发现，理解了以后，确实有点用，所以这里也费一些笔墨吧。

原子性(Atomicity)：也就是说，一组操作，要不就都成功，要不就都失败。不存在中间状态。

一致性(Consistency)：一致性，也就是说，这个事务在提交或回滚的时候，对其他人（或机器）来说，数据的状态是同一的，不会出现中间的状态。最理想的状态下，就是说，数据提交后，所有的更改立刻同时生效，可惜，在计算机领域，这个做不到。。因为cpu运算，磁盘写入，内存写入，都是要时间的，内部一定是个顺序化的过程，所以不可能做到绝对的立刻同时生效。

所以一致性一般来说指代的是逻辑上的同时生效，比如，我要改A,B两行数据，那么，最简单的一致性保证就是，对A,B加锁，改A,B，然后对A，B解锁。

这样下一个人读到的一定是A,B的最新值啦。

（但这块有很多种解释，一般来说这是个最不明确的词汇）。

隔离性(Isolation)：隔离，这是面试最容易挂的一个问题，其实我认为不怪我们，而是因为本身这个隔离性，是依托锁来进行设计的。

我们所知道的锁，主要有以下几种，1.读写锁，2. 排他锁

那么这四种级别其实就和这两种锁的实现有关，之所以要定义四个级别，其实原因也是因为，锁的范围越大，并行效率越低。而范围越小，那么并行的效率就越高。

读未提交： 其实就是什么锁也没有，所以数据的中间状态，是可能被其他人读到的。

读已提交：就是读写锁实现，读锁在查询之后会被释放掉，所以这样其他人可能会更改那些被释放了读锁的数据，这样当前事务再去读取的时候，就可能读取到被别人修改过的数据了，所以一个人在事务中读取到的某个数据，可能下次读取就变成别的数据啦。这就是不可重复读的意思。。

可重复读：也是个读写锁实现，读锁会阻塞其他人（或机器）的写，于是，只要是事务中读取到得数据，都被加了锁，其他人没办法改他们，于是就实现了可重复读咯。

最后是序列化，就是所有都顺序，一个大锁全部锁住J

持久性(Durability)：持久性就是，事务执行后，就丢不了了，就算是整个中国被淹了，机器都没了，数据也不应该丢掉（不过基本做不到这个，也就是一个机器挂了不会丢数据而已。。）所有机房没了那数据也就没了。。

对于这块，给大家一些参考资料：

<http://zh.wikipedia.org/wiki/%E4%BA%8B%E5%8B%99%E9%9A%94%E9%9B%A2>

<http://www.cnblogs.com/wangiqngpei557/archive/2011/11/19/2255132.html>

这些讲的不错，浅显易懂是我的最爱.

好啦，为了保证我写的东西不会被”qing”这个大怪兽再次吃掉。。我先发这些。

在下一个章节，我们继续在事务这个领域徜徉，给大家介绍一下，在单机上面，事务是如何进行的。

其实在上面介绍ACID的时候

我们已经提到了一种最简单的实现方式，就是锁的实现方式。

从原理来看，事务是个变态而复杂的事情。其实如果是序列化的话呢，那么实现起来一定是非常简单的。

但问题就在于，这样性能实在比较低，于是，就有了非常多的方案，为了能哪怕减少一个地方的锁，或者降低一个地方的锁的级别，就付出大量的时间和代码加以实现。

那么，让我们以崇敬的心情，去拜读一下他们的劳动成果吧~

——————————————————————————–

在上一篇中，我们谈了事务管理的四个核心要素，其中有两个要素是和性能紧密相关的，其实也就是需要涉及到锁的，一个是隔离性，一个是一致性。

一致性问题和隔离性问题，我们都可以归结为一个问题，他们都用于定义，什么时候数据可被共享，什么时候数据必须被独占。而这些决策，就最终决定了整个数据库系统的并行度，也就直接的决定了多线程并发时的性能指标。

如果要改一大批数据，又必须保证这些数据要么都出现，要么都不出现，这时候就有个难题了：因为这些数据不可能在同一个时间被选出，更不可能在同一个时间被更改。

于是就必须想个办法来假装达到这个状态，于是我们就需要一种方法，使得针对不同数据的更改，不同人（或机器）不打架。而如果出现对相同数据的更改，则要将更新进行排队。

这个排队可供选择的方法，就我知道的有：1，排他锁。2. 读写锁。3. Copy on write（MVCC） .4. 队列。5. 内存事务。这些方式。

从性能来说，排他锁最慢，而读写因为读可以并发，所以效率稍高，但写和读不能同时进行。3. Copy on write(MVCC) 则读取和写入之间可以互相不影响，所以效率更高。队列这种方式，内存时效果很好，省去中断上下文切换的时间。内存事务，目前还在研究阶段，具备很大潜力的东西。

排他锁，队列和内存事务，在目前的数据库中用的相对较少，我们就不在这里说了。

这里主要说两种实现，一种是读写锁，一种是MVCC.

先说读写锁，也是隔离性中“读已提交，可重复读”两种实现中最重要的底层实现方式。

简单来说，就是如果一个人在事务中，那么他所有写过的数据，所有读过的数据，都给他来个锁，让其他小样儿都只能等在外面，直到数据库能确定所有更改已经全部完成了，没有剩下什么半拉子状态的时候，就解开所有的锁，让其他人可以读取和写入。Hoho,就是这个了。

那么MVCC呢，其实是对读写锁的一个改进，有一批大牛们，说你们这读写锁，写的时候不能读，读的时候不能写，并行度太低了，我要做个更牛B的，写不阻塞读，读不阻塞写的东西来超越你们。

于是他们想起了copy-on-write.鼓捣了个MVCC数据库出来。。。

题外话，现在的甲骨文，之所以能在数据库领域保持优势地位，有个很重要的原因也是因为他们是很早就在商业数据库系统中实现了MVCC的数据写入引擎。

所以他们的Thomas Kyte 技术副总裁也就有了在他们的最牛逼的oracle专家编程里面有了吹嘘的资本 XD .

这里我们要着重的介绍一下MVCC，因为这东西看起来非常的精妙而美丽。。。现在大量的分布式类存储中，也都在借鉴这套模式中的很多部分来增加自己的并行度，以提升性能。比如megaStore.比如percolator。

我们在读写锁的实现中，提到了写读的相互阻塞问题，MVCC则使用copy-on-write来解决这个问题。

如果一个人在事务中，会先申请一个事务ID,这个ID是自增的，每个事务都有他自己的唯一的ID，那么他写过的数据，都会被转变为一次带有当前事务ID的新数据，在读取的时候，则只会读取小于等于自己事务ID的数据。这样实现的东东，语义上来说，与可重复读就一样了。而如果读小于等于全局ID的数据，那么这样的实现，就是读已提交了。

一般来说，MVCC只实现了四个级别中的第二级和第三级，其他的就没有啦，不过这两个是我们最常见的级别。所以也就大家同乐，同乐了~

有了这个东西，我们的一致性也就很容易保证了，因为一个事物和他对应的版本号对应，又有更改后的数据和更改前的数据，如果要提交，那么就只需要很简单的让更改后的数据生效可见即可，这样我们可以将大量的更新中要做的事情，都在事务过程中进行，这样，比原有的基于读写锁的必须在commit时候一起做掉来说，commit这个操作就轻量化了很多，于是，就可以支持更多的人（或机器）持有事务状态了。

很美妙吧？

我一致认为这是oracle当年的核心竞争力，不过现在基本上是个数据库就用了这一套，我们就不在多嘴啦~

解决了一致性和隔离性，剩下的是原子性和持久性，原子性么，一般来说就是要么都成功，也就是新版本数据都让他生效，要么就都失败，也就是让和自己事务ID对应的所有修改都无效即可。也很好就解决掉了。持久性。这个就是后面我们要在写入模型里面介绍的东西了，基本上来说就是写磁盘策略的事情。

到这里，我们单机ACID的实现大概思路，就给大家介绍过了。下一个章节，我们还要用很多的文字，来向大家介绍在分布式场景中我们面临的事务的难题，以及“我所知道的”百花齐放的解决方法。

在上一章节，我们一起浏览了如何进行单机事务操作。下面我们来看一下分布式场景中我们碰到的问题吧。

需要说明的一点是，这里涉及到的权衡点非常的多。就我短短的工作经验里面，也只是能够简单的涉猎一部分，因为在事务这个领域，目前大家都在尝试提出各种各样的不同的方法，而在taobao，我们目前也没有完美的解决这个问题，更多的是在权衡，在金钱和开发成本之间，做出选择。

那么，我们就先从问题开始，来看一下原来的事务出了什么问题。

在事务中，有ACID四种属性。（见上篇文章）

在分布式场景中，我们看引入了什么因素，导致了什么样的新问题：

1.      延迟因素:光是我们所知最快的信息载体了，各位可能都会从潜意识里面认为光传输信息不就是一眨眼的事情而已。那我们做个简单的计算吧（感谢@淘宝叔度，第一次在分享中让我对这个问题有了个数值化的印象。）：

北京到杭州，往返距离2600km ，光在真空中的传输速度是30wkm/s。在玻璃中的速度是真空的2/3。算下来，最小的请求和响应，之间的延迟就有13ms。并且，因为光在管子里走的不是直线，又有信号干扰等问题，一般来说要乘以2~3倍的因子值。

所以一次最小的请求和响应，时间就差不多有30ms左右了。

再想想TCP的时间窗口的移动策略，相信大家都能意识到，实际上延迟是不可忽略的，尤其在传输较多数据的时候，延迟是个重要的因素，不能不加以考虑。

并且，延迟 不是 带宽，带宽可以随便增加，千兆网卡换成万兆，但延迟却很难降低。而我们最需要的，是带宽，更是延迟的降低。因为他直接决定了我们的可用性。

2.      灾备因素：单机的情况下，人们一般不会去追求说一个机器物理上被水冲走了的时候，我的数据要保证不丢（因为没办法的嘛。。）。但在分布式场景下，这种追求就成为了可能，而互联网行业，对这类需求更是非常看重，恨不能所有的机器都必须是冗余的，可随意替换的。这样才能保证7\*24小时的正常服务。这无疑增加了复杂度的因素。

3.      Scale out的问题： 单机总是有瓶颈的，于是，人们的追求就一定是：不管任何一种角色的机器，都应该可以通过简单的增加新机器的方式来提升整个集群中任何一个角色的性能，容量等指标。这也是互联网行业的不懈追求。

4.      性能：更快的响应速度，更低的延迟，就是更好的用户体验。（所以google用了个“可怜”到家的简单input框来提升用户体验，笑）。

说道这里，大概大家都应该对在分布式场景下的广大人民群众的目标有了一个粗略的认识了。

那么我们来看一下原有ACID的问题吧。

在上次的章节中，我们也提到了ACID中，A和D相对的，比较容易达到。但C和I都涉及到锁实现，也就和性能紧密的相关了。

然后，人们就开始了纠结，发掘这个C和I，似乎不是那么容易了。

上次，我们谈到，目前主流的实现一次更新大量数据的时候，不同人（或机器）修改数据相互之间不会打架的方法有以下几种：

1.      排他锁

2.      读写锁

3.      Copy-on-write

4.      队列

5.      内存事务

排他锁和读写锁，本身都是锁的实现，单机的锁实现，相对而言是非常简单的事情，但如果涉及到分布式锁，那么消耗就很高了，原因是，锁要在两边都达到一致，需要多次机器之间的交互过程，这个交互的过程，再考虑到延迟的因素，基本上一次加锁请求就要100~200+毫秒的时间了，那么去锁又要这样的时间。而要知道，我们在单机做内存锁操作，最慢也不过10毫秒。。

于是，有一批人就说了，既然这么难，我们不做了！~来个理论证明他很难就行了~。于是就有了CAP和BASE.

所谓CAP，我个人的理解是描述了一种： 在数据存了多份的前提下，一致性和响应时间，读写可用性不可兼得的“现象”而已。

在我这里来看CAP的证明过程就是个扯淡的玩意儿，他只是描述了一种现象而已。原因还是网络延迟，因为延迟，所以如果要做到数据同时出现或消失，那么按照锁的方式原来可能只需要10ms以内完成的操作，现在要200~400ms才能完成，那自然不能接受了。所谓CAP就是这个现象的英文简称，笑。

BASE呢，这个理论似乎更老，其实也是个现象，就是基本可用，软状态，最终一致的简称，也没个证明，其实就是告诉咱：要权衡一下，原来的ACID不太容易实现啦，我们得适当放弃一些啦。但请各位注意，ACID实际上是能够指导我们在什么情况下做什么样的事情能够获取什么样的结果的。而BASE则不行，这也说明BASE不是个经典的理论。

好啦。废话了这么多，其实就是想说，分布式场景没有银弹啦，你们自己权衡去吧。我们大牛们救不了你们啦的意思。。

既然大牛救不了咱，咱就只能自救了。。。

上次我们讲到，单机事务个我们面临的问题，下面我们来说一些我所知的解决的方法。

在我开始做淘宝数据层的时候，被问得最多的无非也就是：如何做事务，如何做join.至今仍然如此，我一般都会简单而明确的跟对方说：没有高效的实现方法。

虽然没有高效的实现，但实现还是有的。作为引子，我们先来介绍一下这种实现的方式。

我们仍然以上一次讲到的bob和smith为例子来说明好了。

开始的时候。Bob要给smith100块，那么实际上事务中要做的事情是

事务开始时查询bob有多少钱。如果有足够多的钱让bob的账户 -100 ,然后给smith 的账户+100 。最后事务结束。

如果这个事情在单机，那么事情可以使用锁的方式加以解决。

但如果bob在一台机器，smith在另外一台机器，我们应该怎么做呢？

第一种最常被人想起的方法，就是两段提交协议。

两段提交协议从原理上来说是非常简单的一套协议。

Prepare(bob-100) at 机器A->prepare (smith+100) at 机器b ->commit(bob) ->commit(smith)

事务结束。

两段提交的核心，是在prepare的阶段，会对所有该操作所影响的数据加锁，这样就可以阻止其他人（或机器）对他的访问。题外话，问个问题: ）如果这时有其他节点，用相反的方向，进行更新，也就是先更新smith,然后更新bob.会有可能发生什么事情呢？

两段提交协议是被我们在大部分场景下放弃的一个模型，原因主要是因为

1.      Tm本身需要记录事务进行的过程，log要保证安全和可信，性能非常低。

2.      锁的利用率和并行性较低。

3.      网络开销较大

4.      可见性要求实际上就等于让快的操作等慢的。

所以从性能角度来说，这类需求不多也不常见。

既然这样的模型不行，有没有其他模型可以使用呢？

有的。

在事务的过程中，细心的读者不难发现，实际上事务中并不需要这么强的一致可见性。

Bob是需要强一致的，因为他的操作仰赖于他有多少钱，如果他的钱不够100，那么是不能让他的账户变为负数的。但smith却不需要，smith不需要判断他的账户有多少钱，只需要把钱加到他的账户里，不少给他，到账时间尽可能短就可以。

Smith不需要chech账户的钱数，这个前提非常重要，这也是我们能使用最终一致性的关键因素。

下面，我们来看一下另外的选择吧。

Bob的账号在机器A上，smith的账号在机器b上。

首先，我们在机器A上做以下操作:

1.      本地事务开始

2.      读取bob的账户

3.      判断是否有充足余额

4.      更新bob的账号，将bob的钱减少100

5.      将需要给smith加100块这个操作，以事务的形式插入到同机(A)的一张log表中，并自动生成一个唯一的transactionID。

6.      事务关闭

然后，异步的发送一个通知，给一个消费者。

消费者接到通知后，从bob的机器上读取到需要给smith+100这个操作，以及该操作所对应的transactionID。

然后，按照如下方法进行运作

1.      查看在去重表内是否有对应的transactionID.如果没有，则

2.      开启本地事务

3.      将smith的账户+100

4.      将transactionID 插入去重表

5.      事务结束

这样，我们也可以完成一个交易的核心流程了。在交易类过程中的大量事务操作，都是以这样的方式完成的。

下面，我们针对上面的这个流程的一些抉择的点进行一些探讨。

首先，是bob这个机器，这里涉及第一个抉择点。

如果bob是个消费大户，短时间内进行了大量购买，那么可能会造成的问题是，bob所在的那个机器会成为热点，如果在某个突发的情况下，某个账户突然成为热点，那么这些有状态的数据很难快速的反应并加以处理，会造成事务数在某个单节点大量堆积。造成挂掉。

可能的解决方法是：

1.      利用两段提交协议来让原来的” 将需要给smith加100块这个操作，以事务的形式插入到同机(A)的一张log表中，并自动生成一个唯一的transactionID”这步操作放在另外的一台机器上进行。

这样做的的好处是，无论bob怎么是热点，都可以通过水平的加log机器的方式来防止这种热点的产生。

坏处则有：

1方案复杂度高

2额外的网络开销

3消息基于网络发送后，会可能得到三个可能的反馈：1. 成功 2. 失败 3. 无反馈。最麻烦的就是这个无反馈，他可能成功，也可能失败。所以是不确定的状态，需要进行事务的两边进行第二次确认，来确保这个事务的参与方是否都做了该做的事情，如果有一方做了类似commit的操作，那么另外的一方应该commit.如果两方都没做commit操作，那么应该回滚。

2.      让bob的库余量更高，并按照访问压力进行数据的切分，按照热度进行数据划分，放弃原有的简单取mod的策略。来兼容这种不均匀特性。

其次，如果有80个系统都关注着smith加了100这个操作的log，要做对应的处理（比如一些人要针对这个加钱操作做个打款短信推送，有些要做个数据分析等等），那么这里就有另外一个问题，这些系统对bob所在的库的读取就会让该机器成为悲剧的存在。

所以，可以考虑的方式是，增加一个队列，使用，推，拉，或推拉结合的方式将smith加100这个操作加以分发。这样就可以减轻主机的压力。

坏处则是：

1方案进一步复杂

2如何保证log到数据分发服务器之间的数据同步是安全的和准确的？

3如何保证分发服务器的可靠和冗余？

4如何保证写入分发服务器的数据的安全和可靠？

再次，smith这边也有问题，为什么要使用一张去重表呢？其实是因为，在发送端，也就是队列将数据发送到目标机器后，也可能从目标机获取到三种不同的反馈，一类是成功（这个占了大多数）。一类是失败。还有一类是。。。没反馈。

当然，最麻烦的还是这个没反馈的情况，没人知道这时候到底对方是做成功了呢？还是没做成功，为了保证最大的吞吐量，又不能其他人都不做事儿了，就等对方的反馈。所以这里就有另外的权衡了。

一般的模型有两类，一类是用分布式事务来完成。

一类是使用努力送达的模型，说叫努力送达，顾名思义，就是只有得到成功的反馈，才停止投递，而其他时候则重复投递消息，直到对方反馈成功为止。

两种模型比较，显然应该追求速度而放弃方便性，于是我们主要来说说这个努力送达以后所带来的影响。

影响一 : 会有重复的投递，也就是说，这个消息可能会投多次，这对于update set version=version+1 这类的操作来说，是个比较毁灭性的打击。

影响二：如果需要重复投递的消息过多，会导致log分发的机器消耗大量资源来进行重复投递。这会影响server的稳定性

影响三：如果大量堆积消息，那么会造成消息的严重delay。smith发现自己在1个月后收到了bob的钱，你说他会不会去K咱一顿: ) .

最后，额外记的这两次log其实在某些场景下也是可以省去的。

以上，就是我在尝试还原淘宝的消息和事务系统时所能大概想到的一些非常需要权衡和注意的问题点。

小小总结一下，整个问题的核心其实是幂等，说白了就是要能够理解数据基于网络的同步过程中，无反馈是一个经常发生的现象，在这种现象中，重复投递比傻傻等待要有效率的多。所以，重复作为一个side affect也就被默认的存在于系统中，所有的工程师都需要认识到这个问题的客观存在，并采取方法去解决之。

在基于网络的数据同步过程中，一致性是第一个被放弃的。然后数据和消息不会出现重复，是第二个被放弃指标。

使用这种模型，我们可以放弃原来快得等慢的的模式，让整体的吞吐量和性能不会受制于锁的限制，所以淘宝和支付宝才能够支持如此大的交易量。完成大量交易订单。

PS,广告下，如果各位对以上的这些权衡点感兴趣，希望能够了解，知道他们在淘宝的实际运作情况以及走过的经验教训，欢迎私信给我简历哦~

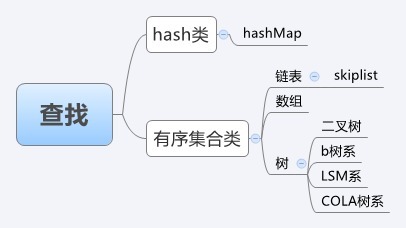
在上一个章节，我们阐述了分布式场景下，事务的问题和一些可能的处理方式后，我们来到了下一章节

Key-value存储

这一章，我们将进入k-v场景，其实，在大部分场景下，如果某个产品宣称自己的写读tps超过其他存储n倍，一般来说都是从k-v这个角度入手进行优化的，主要入手的点是树的数据结构优化和锁的细化，一般都能在一些特定的场景获得5-10倍的性能提升。由此可见key-value存储对于整个数据存储模型是多么的重要。

好吧，那么我们来进入这个章节，用最简单和浅显的话语，阐述这些看起来很高深的理论吧 : )

在未来的几篇中，我们将大概的介绍和分析如下几种比较有特点的数据结构，并探讨其优势劣势以及适用的场景。



让我们先从映射入手吧，所谓映射，就是按照key找到value的过程，这个过程几乎就是我们处理数据的最核心数据结构了。

如何能够根据一个key找到对应的value呢？

一类是hash map.最简单的实现就是算一个key数据的hashCode.然后按照桶的大小取mod.塞到其中的一个桶里面去。如果出现冲突怎么办呢？append到这个桶内链表的尾部就行了。

还有一类呢，我们可以抽象的认为是一个有序结构。之所以把它归类到有序结构原因也很简单，因为只有有序才能做二分查找。。。举些有序结构的例子吧： 1. 数组 2. 各类平衡二叉树 3. B-树类族 4. 链表

这些数据结构如果想进行快速查找，都需要先让他们有序。然后再去做log2N的二分查找找到对应的key。

从原教旨上来说，这就是我们要用的key-value的主要结构了。

那么，hash和有序结构，他们之间有什么样的差别呢？我们来进行一下简单的比较



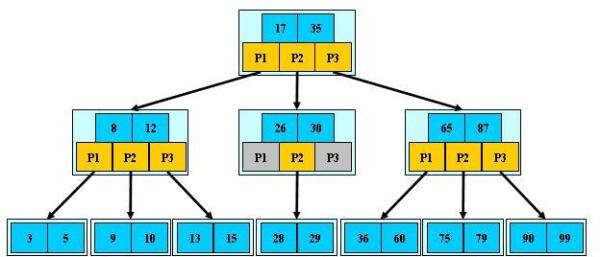
基本上来说，核心区别就是上面的这点，hash单次查询效率较高，但为了保证O(1)效率，对空间也有一定要求。而有序结构，查询效率基本是O(log2N)这个级别。但有序结构可以支持范围查找，而hash则很难支持。

所以，一般来说我们主要在使用的是有序结构来进行索引构建，因为经常需要查询范围。

不过，所有数据库几乎都支持hash索引，如果你的查询基本都是单值的，那么可以找一找稳定的hash索引，他们能从一定程度上提升查询的效率。

在这里，我们主要讨论有序结构，对于数据库或nosql来说，有序结构主要就是指b-tree或b-tree变种。那么我们先来介绍一下什么叫b-tree作为讨论磁盘结构的入门吧。

先上图(copy的，这是个b+tree。版权方请找我)



首先进行词汇科普：b-tree只有两类，一类叫b-tree，就是btree，还有一类是b+tree，但b-tree不是b”减”树的意思。这个大家不要再跟当年的我犯同样的错误哟 :\_\_0

那么b树的核心是几个关键词

1.      树高:一般来说，树的高度比较低。三到五层

2.      数组:每一个node，都是一个“数组”，数组是很关键的决定性因素，我们后面写入和读取分析的时候会讲到。

没了呵呵

然后我们进行一下读取和写入的模拟。

读取来说：如果我要查找28这个数据对应的value是多少，路径大概是：首先走root节点，取出root node后，对该数组进行二分查找，发现35>28>17,所以进入branch节点中的第二个节点，取出该节点后再进行二分查找。发现30>28>26，所以进入branch节点的p2 value，取出该节点，对该三个值的数组进行二分查找，从而定位到28这个数据的对应value。

而写入删除则涉及到分裂和合并这两个btree最重要的操作，比如，要写入37，那么会先找到36所应该被插入的数组[36,60]这个数组，然后判断其是否有空，如果有空，则对该数组进行重新排序。而如果没有空，则必须要进行分裂。分裂的缘由是因为组成b-tree的每一个node,都是一个数组，数组最大的特性是，数组内元素个数是固定的。因此必须要把原有已经满掉的数组里面的一半的数据拿出来，放到新的一个新建立的空数组中，然后把要写入的数据写入到老或新的这两个数组里面的一个里面去。

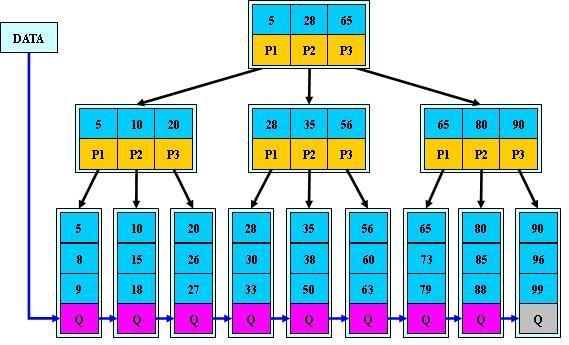
【这里要留个问题给大家了，我想问一下，为什么b-tree要使用数组来存储数据呢？为什么不选择链表等结构呢？】

对于上面的这个小的b-tree sample里面呢，因为数组[35,60]，数组已经满了，所以要进行分裂。于是数组在插入了新值以后，变成了两个[35,36] 和[60] ，然后再改变父节点的指针并依次传导上去即可。

当出现删除的时候，会可能需要进行合并的工作，也就是写入这个操作的反向过程。在一些场景中，因为不断地插入新的id,删除老的id，会造成b-tree的右倾，这时候需要有后台进程对这种倾向进行不断地调整。

基本上，这就是b-tree的运转过程了。

B+tree



B+tree 其实就是在原有b-tree的基础上。增加两条新的规则

1.      Branch节点不能直接查到数据后返回，所有数据必须读穿或写穿到leaf节点后才能返回成功

2.      子叶节点的最后一个元素是到下一个leaf节点的指针。

这样做的原因是，更方便做范围查询，在b+树种，如果要查询20~56.只需要找到20这个起始节点，然后顺序遍历，不再需要不断重复的访问branch和root节点了。

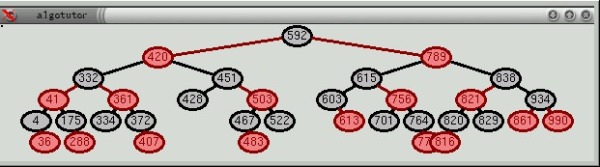
发现每一种数据结构都需要去进行简介才能够比较方便的了解到他们的特性，所以在后续的章节还会介绍几种有代表性的树的结构都会针对性的加以介绍。

首先来回答一个问题：为什么在磁盘中要使用b+树来进行文件存储呢？

原因还是因为树的高度低得缘故，磁盘本身是一个顺序读写快，随机读写慢的系统，那么如果想高效的从磁盘中找到数据，势必需要满足一个最重要的条件：减少寻道次数。

我们以平衡树为例进行对比，就会发现问题所在了：

先上个图



这是个平衡树，可以看到基本上一个元素下只有两个子叶节点

抽象的来看，树想要达成有效查找，势必需要维持如下一种结构：

树的子叶节点中，左子树一定小于等于当前节点，而当前节点的右子树则一定大于当前节点。只有这样，才能够维持全局有序，才能够进行查询。

这也就决定了只有取得某一个子叶节点后，才能够根据这个节点知道他的子树的具体的值情况。这点非常之重要，因为二叉平衡树，只有两个子叶节点，所以如果想找到某个数据，他必须重复更多次“拿到一个节点的两个子节点，判断大小，再从其中一个子节点取出他的两个子节点，判断大小。”这一过程。

这个过程重复的次数，就是树的高度。那么既然每个子树只有两个节点，那么N个数据的树的高度也就很容易可以算出了。

平衡二叉树这种结构的好处是，没有空间浪费，不会存在空余的空间，但坏处是需要取出多个节点，且无法预测下一个节点的位置。这种取出的操作，在内存内进行的时候，速度很快，但如果到磁盘，那么就意味着大量随机寻道。基本磁盘就被查死了。

而b树，因为其构建过程中引入了有序数组，从而有效的降低了树的高度，一次取出一个连续的数组，这个操作在磁盘上比取出与数组相同数量的离散数据，要便宜的多。因此磁盘上基本都是b树结构。

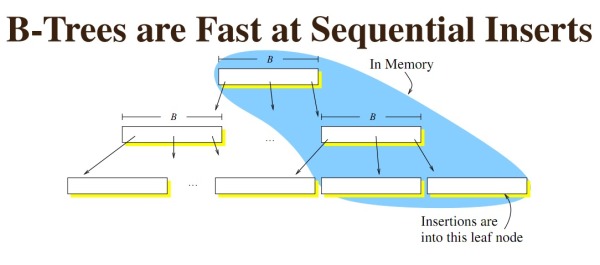
不过，b树结构也不是完美的，与二叉树相比，他会耗费更多的空间。在最恶劣的情况下，要有几乎是元数据两倍的格子才能装得下整个数据集（当树的所有节点都进行了分裂后）。

以上，我们就对二叉树和b树进行了简要的分析，当然里面还有非常多的知识我这里没有提到，我希望我的这个系列能够成为让大家入门的材料，如果感兴趣可以知道从哪里着手即可。如果您通过我的文章发现对这些原来枯燥的数据结构有了兴趣，那么我的目标就达到了: )

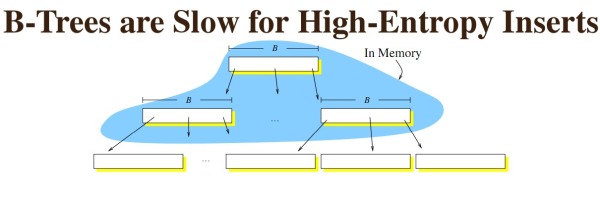
在这章中，我们还将对b数的问题进行一下剖析，然后给出几个解决的方向

其实toku DB的网站上有个非常不错的对b树问题的说明，我在这里就再次侵权一下，将他们的图作为说明b树问题的图谱吧，因为真的非常清晰。

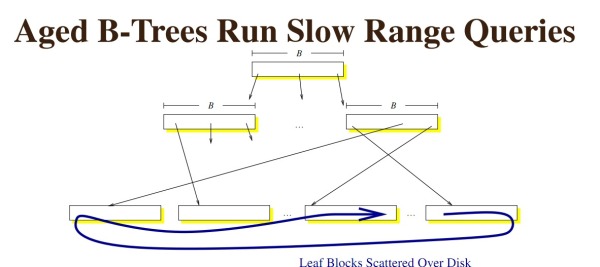
<http://tokutek.com/downloads/mysqluc-2010-fractal-trees.pdf>



B树在插入的时候，如果是最后一个node,那么速度非常快，因为是顺序写。



但如果有更新插入删除等综合写入，最后因为需要循环利用磁盘块，所以会出现较多的随机io.大量时间消耗在磁盘寻道时间上。



如果是一个运行时间很长的b树，那么几乎所有的请求，都是随机io。因为磁盘块本身已经不再连续，很难保证可以顺序读取。

以上就是b树在磁盘结构中最大的问题了。

那么如何能够解决这个问题呢？

目前主流的思路有以下几种

1.      放弃部分读性能，使用更加面向顺序写的树的结构来提升写性能。

这个类别里面，从数据结构来说，就我所知并比较流行的是两类，

一类是COLA（Cache-Oblivious Look ahead Array）（代表应用自然是tokuDB）。

一类是LSM tree(Log-structured merge Tree)或SSTABLE

(代表的数据集是cassandra,hbase,bdb java editon,levelDB etc.).

2.      使用ssd，让寻道成为往事。

我们在这个系列里，主要还是讲LSM tree吧，因为这个东西几乎要一桶浆糊了。几乎所有的nosql都在使用，然后利用这个宣称自己比mysql的innodb快多少多少倍。。我对此表示比较无语。因为nosql本身似乎应该是以省去解析和事务锁的方式来提升效能。怎么最后却改了底层数据结构，然后宣称这是nosql比mysql快的原因呢？

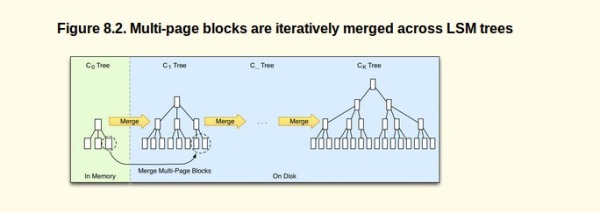
毕竟Mysql又不是不能挂接LSM tree的引擎。。。

好吧，牢骚我不多说，毕竟还是要感谢nosql运动，让数据库团队都重新审视了一下数据库这个产品的本身。

那么下面，我们就来介绍一下LSM Tree的核心思想吧。

首先来分析一下为什么b+树会慢。

从原理来说，b+树在查询过程中应该是不会慢的，但如果数据插入比较无序的时候，比如先插入5 然后10000然后3然后800 这样跨度很大的数据的时候，就需要先“找到这个数据应该被插入的位置”，然后插入数据。这个查找到位置的过程，如果非常离散，那么就意味着每次查找的时候，他的子叶节点都不在内存中，这时候就必须使用磁盘寻道时间来进行查找了。更新基本与插入是相同的



那么，LSM Tree采取了什么样的方式来优化这个问题呢？

简单来说，就是放弃磁盘读性能来换取写的顺序性。

乍一看，似乎会认为读应该是大部分系统最应该保证的特性，所以用读换写似乎不是个好买卖。但别急，听我分析之。

1.      内存的速度远超磁盘，1000倍以上。而读取的性能提升，主要还是依靠内存命中率而非磁盘读的次数

2.      写入不占用磁盘的io，读取就能获取更长时间的磁盘io使用权，从而也可以提升读取效率。

因此，虽然SSTable降低了了读的性能，但如果数据的读取命中率有保障的前提下，因为读取能够获得更多的磁盘io机会，因此读取性能基本没有降低，甚至还会有提升。

而写入的性能则会获得较大幅度的提升，基本上是5~10倍左右。

下面来看一下细节

其实从本质来说，k-v存储要解决的问题就是这么一个：尽可能快得写入，以及尽可能快的读取。

让我从写入最快的极端开始说起，阐述一下k-v存储的核心之一—树这个组件吧。

我们假设要写入一个1000个节点的key是随机数的数据。

对磁盘来说，最快的写入方式一定是顺序的将每一次写入都直接写入到磁盘中即可。

但这样带来的问题是，我没办法查询，因为每次查询一个值都需要遍历整个数据才能找到，这个读性能就太悲剧了。。

那么如果我想获取磁盘读性能最高，应该怎么做呢？把数据全部排序就行了，b树就是这样的结构。

那么，b树的写太烂了，我需要提升写，可以放弃部分磁盘读性能，怎么办呢？

简单，那就弄很多个小的有序结构，比如每m个数据，在内存里排序一次，下面100个数据，再排序一次……这样依次做下去，我就可以获得N/m个有序的小的有序结构。

在查询的时候，因为不知道这个数据到底是在哪里，所以就从最新的一个小的有序结构里做二分查找，找得到就返回，找不到就继续找下一个小有序结构，一直到找到为止。

很容易可以看出，这样的模式，读取的时间复杂度是(N/m)\*log2N 。读取效率是会下降的。

这就是最本来意义上的LSM tree的思路。

那么这样做，性能还是比较慢的，于是需要再做些事情来提升，怎么做才好呢？

于是引入了以下的几个东西来改进它

1.      Bloom filter : 就是个带随即概率的bitmap,可以快速的告诉你，某一个小的有序结构里有没有指定的那个数据的。于是我就可以不用二分查找，而只需简单的计算几次就能知道数据是否在某个小集合里啦。效率得到了提升，但付出的是空间代价。

2.      小树合并为大树： 也就是大家经常看到的compact的过程，因为小树他性能有问题，所以要有个进程不断地将小树合并到大树上，这样大部分的老数据查询也可以直接使用log2N的方式找到，不需要再进行(N/m)\*log2n的查询了。

这就是LSMTree的核心思路和优化方式。

不过，LSMTree也有个隐含的条件，就是他实现数据库的insert语义时性能不会很高，原因是，insert的含义是： 事务中，先查找该插入的数据，如果存在，则抛出异常，如果不存在则写入。这个“查找”的过程，会拖慢整个写入。

这样，我们就又介绍了一种k-v写入的模型啦。在下一次，我们将再去看看另外一种使用了类似思路，但方法完全不同的b树优化方式 COLA树系。敬请期待 ~

终于来到了COLA树系，这套东西目前来看呢，确实不如LSM火，不过作为可选方案，也是个值得了解的尝试，不过这块因为只有一组MIT的人搞了个东西出来，所以其实真正的方案也语焉不详的。从性能来说，tokuDB的写入性能很高，但更新似乎不是很给力，查询较好，占用较少的内存。

<http://www.mysqlperformanceblog.com/2009/04/28/detailed-review-of-tokutek-storage-engine/>

这里有一些性能上的指标和分析性文字。确实看起来很心动，不过这东西只适合磁盘结构，到了SSD似乎就挂了。原因不详，因为没有实际的看过他们的代码，所以一切都是推测，如果有问题，请告知我。

先说原理，上ppt <http://tokutek.com/presentations/bender-Scalperf-9-09.pdf>，简单来说，就是一帮MIT的小子们，分析了一下为什么磁盘写性能这么慢，读的性能也这么慢，然后一拍脑袋，说：“哎呀，我知道了，对于两级的存储（比如磁盘对应内存，或内存对于缓存，有两个属性是会对整个查询和写入造成影响的，一个是容量空间小但速度更快的存储的size，另外一个则是一次传输的block的size.而我们要做的事情，就是尽可能让每次的操作传输尽可能少的数据块。

传输的越少，那么查询的性能就越好。

进而，有人提出了更多种的解决方案。

•B-tree [Bayer, McCreight 72]

• cache-oblivious B-tree [Bender, Demaine, Farach-Colton 00]

• buffer tree [Arge 95]

• buffered-repositorytree[Buchsbaum,Goldwasser,Venkatasubramanian,Westbrook 00]

• Bε

tree[Brodal, Fagerberg 03]

• log-structured merge tree [O'Neil, Cheng, Gawlick, O'Neil 96]

• string B-tree [Ferragina, Grossi 99]

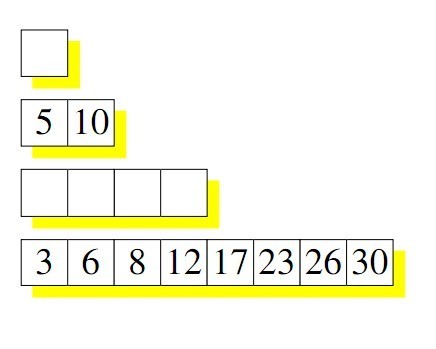
这些结构都是用于解决这样一个问题，在磁盘上能够创建动态的有序查询结构。

在今天，主要想介绍的就是COLA，所谓cache-oblivious 就是说，他不需要知道具体的内存大小和一个块的大小，或者说，无论内存多大，块有多大，都可以使用同一套逻辑进行处理，这无疑是具有优势的，因为内存大小虽然可以知道，但内存是随时可能被临时的占用去做其他事情的，这时候，CO就非常有用了。

其他我就不多说了，看一下细节吧~再说这个我自己都快绕进去了。

众所周知的，磁盘需要的是顺序写入，下一个问题就是，怎么能够保证数据的顺序写。

我们假定有这样一个空的数据集合

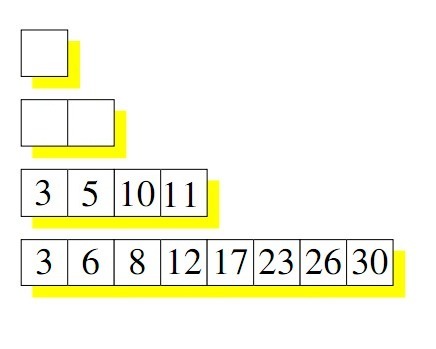


可以认为树的高度是log2N。

每行要么就是空的，要么就是满的，每行数据都是排序后的数据。

如果再写一个值的时候，会写在第一行，比如写了3。

再写一个值11的时候，因为第一行已经写满了，所以将3取出来，和11做排序，尝试写第二行。又因为第二行也满了，所以将第二行的5和10也取出，对3,11,5,10 进行排序。写入第四行



这就是COLA的写入过程。

可以很清楚的看出，COLA的核心其实和LSM类似，每次“将数据从上一层取出，与外部数据进行归并排序后写入新的array”的这个操作，对sas磁盘非常友好。因此，写入性能就会有非常大的提升。

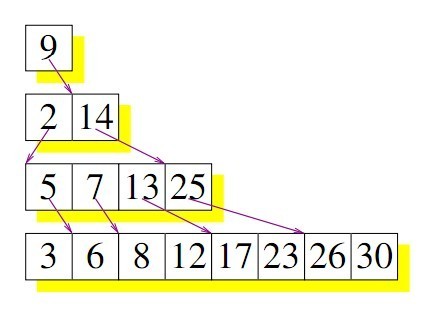
并且因为数据结构简单，没有维持太多额外的指针，所以相对的比较节省空间。

但这样查询会需要针对每个array都进行一次二分查找。

性能似乎还不是很高，所以，他们想到了下面这种方式，把它的命名为fractal tree,分形树。

用更简单的方法来说的话呢，就是在merge的时候，上层持有下层数据的一个额外的指针。

来协助进行二分查找。



这样，利用空间换时间，他的查询速度就又回到了log2N这个级别了。

到此，又一个有序结构被我囫囵吞枣了。

嘿嘿

在下一篇，我们将进入大家期待的分布式k-V场景，也就是noSQL的范畴了，让我们拨开nosql的神秘面纱，看看这东西到底意味着什么。

**上一次，我们介绍了几种常见的kv存储模型，下面我们就正式进入到分布式存储的场景里去看看这套东西在分布式场景下的运作方式吧。**

**在分布式key-value中，很多原来的知识是可以继续复用的。因为k-v解决的问题实在是非常的简单，只不过是根据一个key找到value的过程，所以原来的知识，现在也继续的可以用。**

**但有两个额外的因素需要考虑**

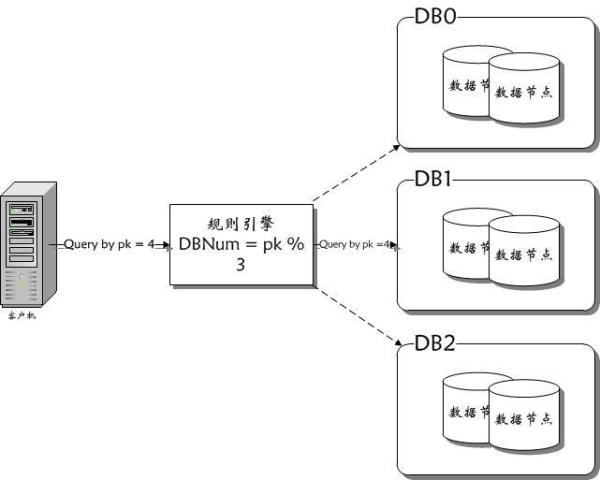
**网络延迟**

**TCP/IP –公用网络，ip跳转慢，tcp包头大**

**可能出现不可达问题**

**这其实是状态机同步中最难的一个问题，也就是，A给B消息，B可能给A的反馈可能是:1. 成功 2. 失败 3. 无响应。最难处理的是这个无响应的问题。以前的文章中我们讨论过这个问题，以后还会碰到。这里暂且hold住。**

**先上图一张，在未来的几周内，我们都会依托这张图来解释分布式K-V系统**

****

**可以看到，在客户机到服务器端，有这么几个东西**

**一，规则引擎**

**二，数据节点间的同步**

**抽象的来看，分布式K-V系统和传统的单机k-v系统的差别，也就只在于上面的两个地方的选择。**

**今天先来谈规则引擎**

**抽象的来看，规则引擎面向的场景应该被这样的描述：对于有状态的数据，需要一套机制以保证其针对同一个数据的多次请求，应该物理上被发送到同一个逻辑区块内的同一个数据上。**

**举例子来说，一个人进行了三笔交易，每笔交易都是这个人给其他人100元。那么，这三比交易的更新(update set money = money – 100 where userid = ?) 必须被发到同一台机器上执行，才能拿到正确的结果。【不考虑读性能的gossip模型除外】**

**这种根据一个userid 找到其对应的机器的过程，就是规则引擎所要处理的事情。**

**我们对于规则引擎的需求，一般来说也就是要查的快，第二个是要能尽可能的将数据平均的分配到所有的节点中去。第三个，如果新的节点加入进去，希望能够只移动那些需要移动的数据，不需要移动的数据则不要去移动他。**

**那么一想到“根据xxx找到xxx”相信大家第一个想到的一定又是以前说过的K-V了。所以我们就再复习一遍: )**

**Hash**

**O(1)效率**

**不支持范围查询（时间这样的查询条件不要考虑了）**

**不需要频繁调整数据分布**

**顺序**

**主要是B-Tree**

**O(logN)效率**

**支持范围查询**

**需要频繁调整节点指针以适应数据分布**

**这也就是我们最常用的两种分布式k-value所使用的数据结构了**

**首先来看HASH的方法。Hash其实很容易理解，但我跟不少人交流，发现大家可能对一致性hash的理解有一定的误解。下面请允许我给大家做个简单的介绍。**

**简单取模:**

**最简单的HASH 就是取mod,user\_id % 3 。这样，会将数据平均的分配到0,1,2 这三台机器中。**

**这也是我们目前最常用的，最好用的方案。但这套方案也会有一个问题，就是如果id % 3 -> id % 4 总共会有80%的数据需要变动hash桶，想一想，只增加了一台机器，但80%的数据需要从一个机器移动到另外一个机器，这无疑是不够经济的，也是对迁移不友好的方案。**

**不过，虽然增加一台机器，会发生无谓的数据移动，但取模的方案在一些特殊的场景下，也能很好的满足实际的需要，如id % 3 -> id % 6，这种情况下，只需要有50%的数据移动到新的机器上就可以了。这也就是正常的hash取模最合适的扩容方式—–> 倍分扩容。我们一般把这种扩容的方式叫做”N到2N”的扩容方案。**

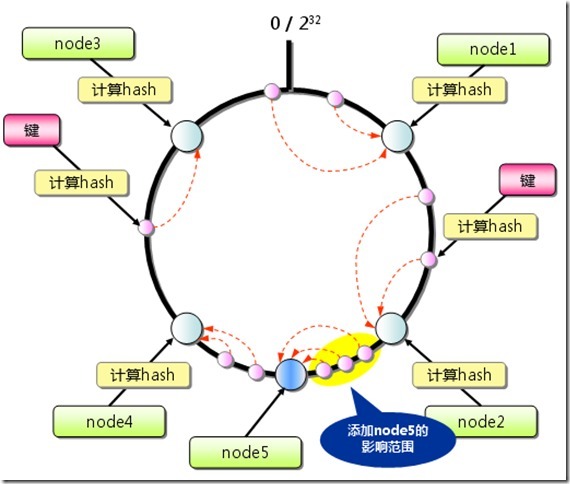
**取模Hash还有个无法解决的问题，就是无法处理热点的问题，假设有一个卖家有N个商品。如果按照卖家ID进行切分，那么就有可能会造成数据不均匀的问题。有些卖家可能有10000000个商品，而有些卖家只会有10个。这种情况下如果有大量商品的卖家针对他的商品做了某种操作，那这样无疑会产生数据热点。如何解决这类问题，也是分布式场景中面临的一个重要的问题。**

**既然简单取模有这么多的问题，那有没有办法解决这些问题呢？**

**首先，我们来介绍第一种解决这个问题的尝试。**

**一致性Hash.**

**先来个图，这套图估计几乎所有对Nosql稍有了解的人都应该看过，在这里我会用另外的方式让大家更容易理解**

****

**上面这个图，用代码来表示，可以认为是这样一套伪码**

**Def idmod = id % 1000 ;**

**If(id >= 0 and id < 250)**

**returndb1;**

**Else if (id >= 250 and id < 500)**

**returndb2;**

**Else if (id >= 500 and id < 750)**

**returndb3;**

**Else**

**returndb4;**

**这个return db1 db2 db3 db4 就对应上面图中的四个浅蓝色的点儿。**

**而如果要加一个node5 ，那么伪码会转变为**

**Def idmod = id % 1000 ;**

**If(id >= 0 and id < 250)**

**returndb1;**

**Else if (id >= 250 and id < 500)**

**returndb2;**

**Else if (id >= 500 and id < 625)**

**returndb5;**

**else if(id >= 625 and id <750)**

**returndb3**

**Else**

**returndb4;**

**从这种结构的变化中，其实就可以解决我们在普通hash时候的面临的两个问题了。**

**1.      可以解决热点问题，只需要对热点的数据，单独的给他更多的计算和存储资源，就能部分的解决问题（但不是全部，因为迁移数据不是无成本的，相反，成本往往比较高昂）**

**2.      部分的能够解决扩容的问题，如果某个点需要加机器，他只会影响一个节点内的数据，只需要将那个节点的数据移动到新节点就可以了。**

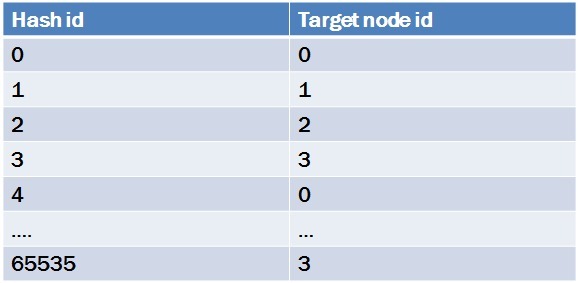
**但一致性hash也会带来问题，如果数据原本分布就非常均匀，那么加一台机器，只能解决临近的一个节点上的热点问题，不会影响其他节点，这样，热点扩容在数据分布均匀的情况下基本等于n->2n方案。因为要在每个环上都加一台机器，才能保证所有节点的数据的一部分迁移到新加入的机器上。**

**这无疑对也会浪费机器。**

**于是，我们又引入了第三套机制:**

**虚拟节点hash**

**Def hashid = Id % 65536**

****

**可以很容易的看出，上面这套虚拟节点的方案，其实与id % 4的结果等价。**

**可以认为一致性hash和普通节点hash，都是虚拟节点hash的特例而已。**

**使用虚拟节点hash,我们就可以很容易的解决几乎所有在扩容上的问题了。**

**碰到热点？只需要调整虚拟节点map中的映射关系就行了**

**碰到扩容？只需要移动一部分节点的映射关系，让其进入新的机器即可**

**可以说是一套非常灵活的方案，但带来的问题是方案有点复杂了。**

**所以，我们一般在使用的方式是，首先使用简单的取模方案，如id % 4。在扩容的时候也是用N->2N的方案进行扩容。但如果碰到需求复杂的场景，我们会“无缝”的将业务方原来的简单取模方案，直接变为使用虚拟节点hash的方案，这样就可以支持更复杂的扩容和切分规则，又不会对业务造成任何影响了。**

**好，到这，我基本上就给大家介绍了如何使用Hash来完成分布式k-value系统的规则引擎构建了。**