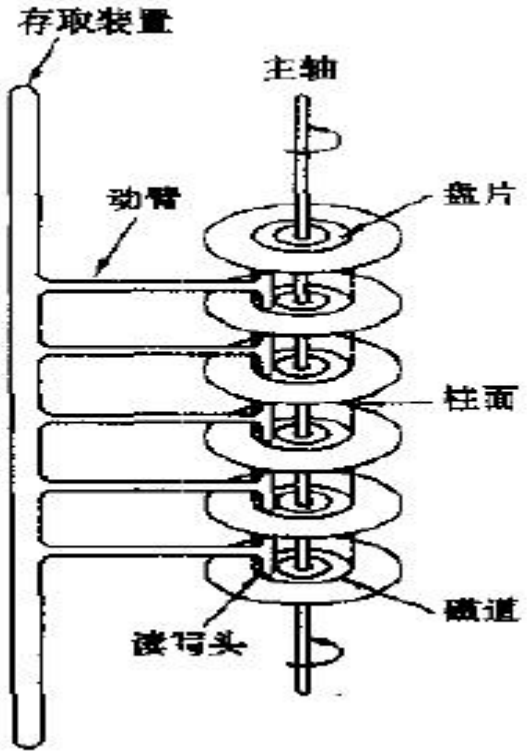
CoolHash数据库引擎架构与设计分享

彭渊

思路提纲

1、存储硬件技术的发展：机械硬盘、固态硬盘的原理
2、数据库软件技术的发展：基于裸设备、b/b+树、跳表的的数据库实现原理
3、业界典型NOSQL数据库架构分析（Aerospike，KyotoCabinet、Couchbase、Redis）
4、Redis解决方案、瓶颈和优化
5、CoolHash的数据逻辑结构
6、CoolHash的并行架构设计+简单事务处理
7、CoolHash的性能分析

机械硬盘存储的伟大发明



机械硬盘是在金属片上涂以磁性材料，通过对磁性材料磁化后的剩磁状态来存储二进制信息。

1、寻找柱面（毫秒级）
2、寻找盘面
3、寻找扇区

操作系统的文件系统实现将连续数据尽量存放同一柱面(不连续即随机读写)
每个文件至少存放一个簇（相邻的扇区），目录区存储文件元数据，数据区存储文件内容

固态硬盘的存储原理

1、主体pcb板
2、主控芯片：调配闪存芯片数据负荷和中转。
3、缓存芯片：辅助主控芯片进行数据处理。
4、闪存芯片（Flash）：存储介质，以充电、放电方式写入擦除数据。不用磁头，通过电路传输信号，寻道时间几乎为0，随机读写速度快。（500m/s VS 100m/s，<1毫秒）

无机械不发热无噪音抗低温抗地震，但是价格贵有擦写次数寿命限制。国内厂商一般购买主控芯片和闪存芯片组装。

数据库软件存储技术的发展趋势

数据库作为企业信息系统的最基础软件，面临着分布式存储、nosql、k/v、并行数据库等创新技术的冲击

- 结构化存储往非结构化存储发展
- 集中存储往分布式存储发展
- sql数据库往nosql发展
- 关系数据库往kv数据库发展
- 单进程数据往并行数据库发展
- 随着软硬件技术的发展，缓存和持久化存储的性能越来越接近

基于Raw device的数据库实现

1、Raw device，没有经过格式化，不被操作系统直接管理的设备，不通过操作系统文件系统来操作。

2、使用应用程序直接操作Raw device，不经过文件系统的缓冲。

3、由于绕开操作系统和其文件系统，直接操作I/O，控制得当可以提高效率。

4、读写很频繁，磁盘I/O成为瓶颈情况下适合操作Raw device。

5、ORACLE、SQLSERVER等数据库支持使用Raw device作为存储介质。

开发方式：

1、常用c开发操作打开dev/sdb...

2、java开发问题（不太适合）：

//读可以打开

```
RandomAccessFile file = new RandomAccessFile("\\\\.\ \PhysicalDrive1","r");
```

//写抛出异常FileNotFoundExceptions

```
RandomAccessFile file = new RandomAccessFile("\\\\.\ \PhysicalDrive1","rw");
```

Raw device的限制

1、sqlserver、oracle安装必须在文件系统上，支持在Raw device创建数据库

2、数据库没有物理名字，只有磁盘驱动号，每个raw分区只能有一个数据文件

3、复制删除重命名等操作无法使用

4、备份和管理比较麻烦，linux使用DD底层命令

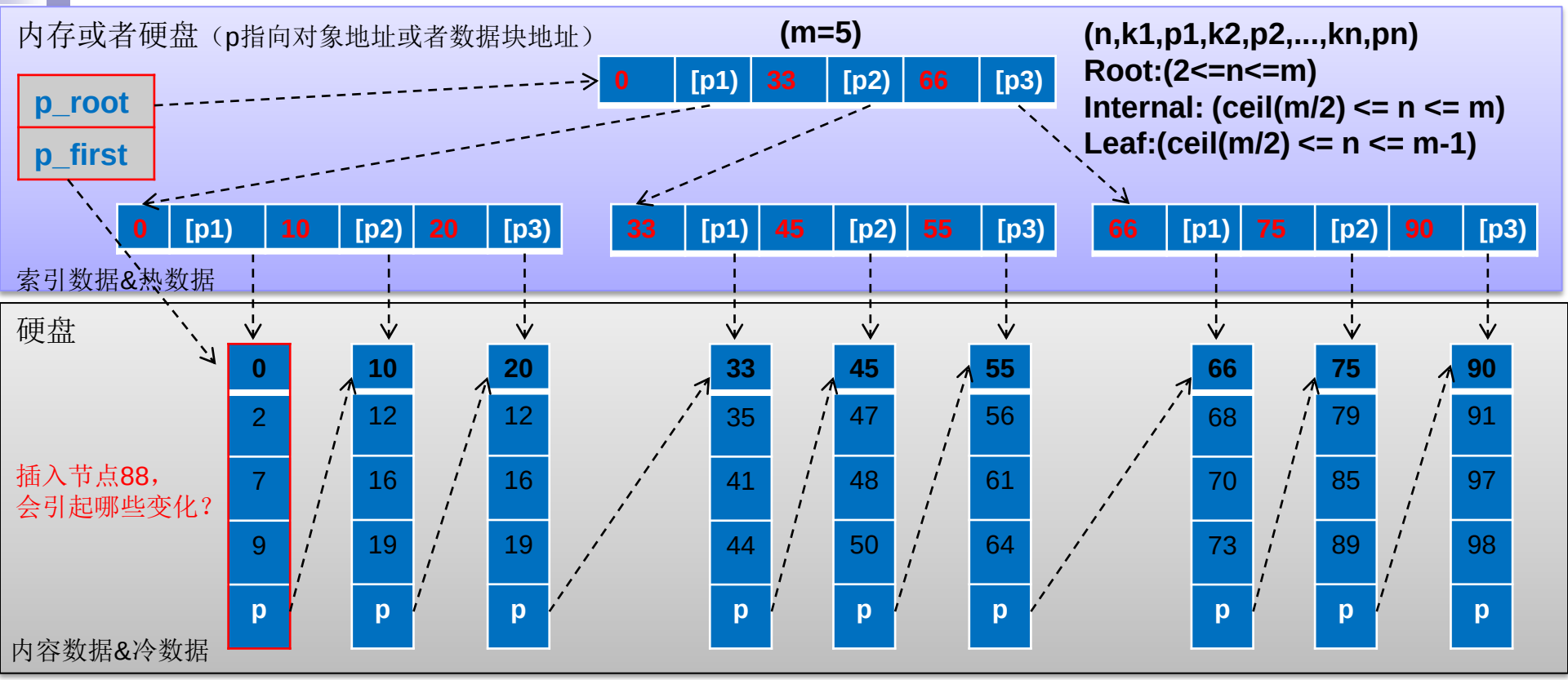
5、数据文件分区大小无法更改

6、磁盘整理坏块替换等系统维护无法使用到Raw device

7、sqlserver推荐使用NTFS或者FAT建立数据库，Oracle 11后不再支持Raw device（包括OUI,DBCA,命令行）

8、硬件专家的经验，一定程度可以提高效率，但是不突出

基于B/B+ Tree实现数据库

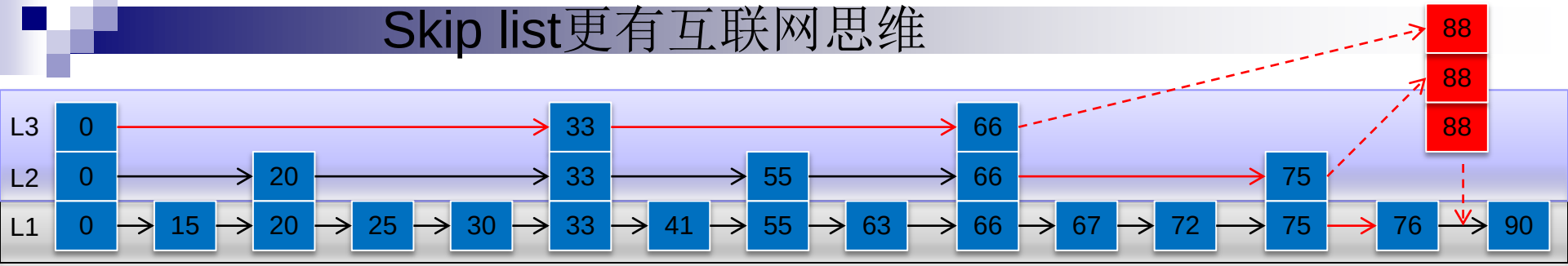


好处：查找、范围查找（锁定两头范围）、遍历方便；每个节点有关键字最大范围，可以控制好数据块存储大小（page），方便做计算；1000阶的3层b+树可以最大存储接近10亿个节点。

对一个懒惰的程序员，用b+树实现数据库麻烦在哪里：

维持平衡的树结构的开销大，为了动态调整树子节点在最大最小范围内，需要不断的分裂树（插入）、合并树（删除），并容易引起向上左右连锁分裂或合并，指针切换和数据块迁移频繁，编程复杂，招式太多容易影响性能；牵一发动全身，插入删除节点引起周边节点关联变化难以事先预估影响范围，对于多条数据插入和更新，难以分解出各自独立任务，并行化困难，很多实现只能单点写。

Skip list更有互联网思维



存储方式：
1、节点每层存储，每个节点含有向右向下指针，除第一层外的索引节点放内存，第一层放硬盘。
2、一个节点只存储一份，用一个指针数组指向多层链接，适合内存数据库。

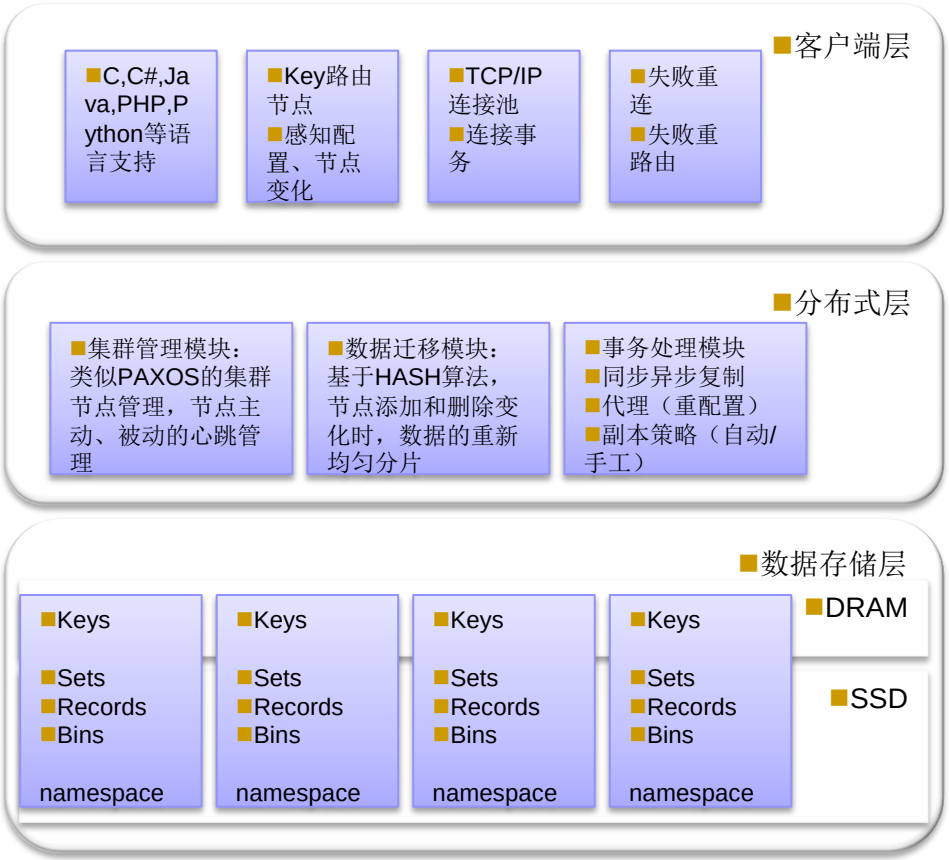
查找方式：从最上层阶梯下降方式最先找到后返回。
插入方式：先随机得到插入层，按查找方式找到该层位置后插入，再继续向下层查找插入直到最后一层，修改节点各层指针。
删除方式：先按查找方式找到后删除，再继续向下层查找删除直到最后一层，修改节点各层指针。
硬币随机：抛硬币直到获取到一个正面的总次数为层数，可约定最大层数限制，不约定需要动态扩充层数组。
每个节点做双向链表数组，每层两个指针，分别指向本层下一个节点和上一个节点，那么删除会更方便，也可以支持从后向前查找，但是要牺牲每节点存储空间。

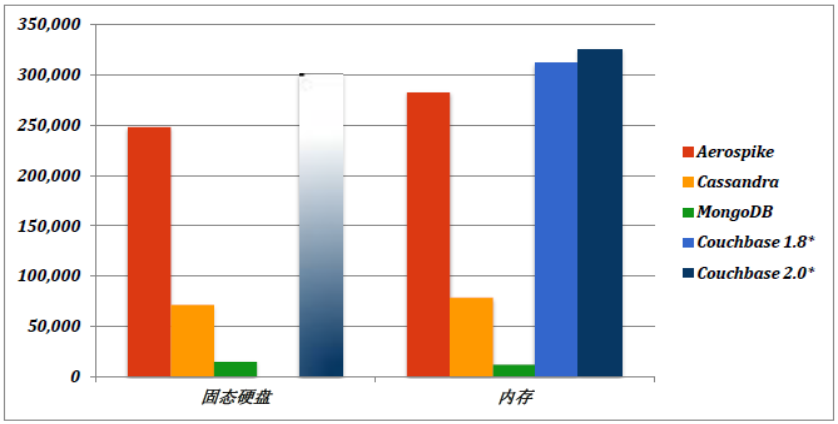
使用场景：跳表适合排序不重复的存储结构，如key/value存储（leveldb），排序set存储(redis zset)

“Skip lists are a data structure that can be used in place of balanced trees. Skip lists use probabilistic balancing rather than strictly enforced balancing and as a result the algorithms for insertion and deletion in skip lists are much simpler and significantly faster than equivalent algorithms for balanced trees.”

——William Pugh

Aerospike架构





1、Aerospike占据了美国RTB市场的商业数据库，优势就是实时响应，国外几乎一线广告科技公司都在用Aerospike。

2、混合存储架构。索引存储在 RAM 中，而数据存储在闪存/固态硬盘（SSD）上，针对采用多核处理器和多处理器机器的现代硬件进行优化

3、唯一的针对闪存优化的In-Memory NoSQL数据库，直接绕过文件系统来利用低层ssd的读写模式（区别常用SSD模式），Aerospike并解决了ssd的使用寿命问题。

4、按行存储，每行存储大小限制在1M。

5、99% 的响应可在 1 毫秒内实现，99.9% 的响应可在 5 毫秒内实现，写TPS在20万-30万（SSD）。

6、成本方面，采用英特尔固态盘DC S3700系列每GB大约为2美元；采用英特尔固态盘DC S3500 系列每GB大约为1美元。

7、没有高效的查询检索设计和实现。

Tokyo Cabinet &KyotoCabinet

1、起源于Kenneth Thompson 开发的DBM（写入慢，检索快），1被inux/unix自带的小型数据库。后续出现一系列的DBM-like产品，比如GDBM（当前linux自带），BerkeleyDB等。

2、日本人平林幹雄设计了QDBM取代GDBM，然后又设计了Tokyo Cabinet取代QDBM， Tokyo Cabinet在日本最大的SNS网站mixi.jp获得了很大成功。

3、Tokyo Cabinet长时间没有更新版本，平林幹雄重新设计了KyotoCabinet，并鼓励用户使用， KyotoCabinet使用c++设计，支持hash和B+树两种实现，提供串行和脏读两种事务

——HashDB：数组+链表实现，数组做mmap，MurMurHash 2.0哈希函数（key长度大）

——B+TreeDB：基于hashDB，性能低点，支持范围查询， Deflate压缩（7-zip实现）

4、Tokyo Tyrant 是由同一作者开发的 Tokyo Cabinet 数据库网络接口。它拥有Memcached兼容协议，也可以通过HTTP协议进行数据交换Tokyo Tyrant 加上 Tokyo Cabinet，构成了一款支持高并发的分布式持久存储系统，对任何原有Memcached客户端来讲，可以将Tokyo Tyrant看成是一个Memcached，但是，它的数据是可以持久存储的。

5、Tokyo Tyrant支持双机互为主辅模式，主辅库均可读写，而其他MySQL等数据库支持主辅库同步的方式实现读写分离，支持“主服务器可读写、辅助服务器只读”模式。

6、对应Tokyo Tyrant，Kyoto Cabinet的接口层组件叫做Kyoto Tyrant。支持网络高并发，多线程访问KC。

Tokyo Cabinet & KyotoCabinet性能

平均性能：

- 写入100万 records / 0.8 sec = 1,250,000 TPS
- 查询100万 records / 0.7 sec = 1,428,571 QPS

测试数据：

写入100万条：tchtest write test.tch 1000000，时间：0.732秒 速度：1366120条/秒

写入200万条：tchtest write test.tch 2000000，时间：1.718秒 速度：1164144条/秒

写入500万条：tchtest write test.tch 5000000，时间：21.529秒 速度：232244条/秒

插入数据超过一定数量性能下降的原因是MMAP内存大小，默认值为67108864，也就是64M，如果数据库文件超过64M，则只有前部分会映射在内存中，所以写入性能会下降。

一些限制：

1、在32位操作系统下，作为 Tokyo Tyrant 后端存储的 Tokyo Cabinet 数据库单个文件不能超过2G，而64位操作系统则不受这一限制。所以，如果使用 Tokyo Tyrant，推荐在64位CPU、操作系统上安装运行。

2、有用户反映： kyoto cabinet超出内存大小后HashDB和TreeDB的入库太慢。

3、GPL开源协议，商用受限制。

CouchDB&Couchbase

1、CouchDB最初由c++开发，后来采用Erlang开发（面向并发语言），使用B树存储结构，Apache 2.0 协议开源。

2、 CouchDB是Apache发布面向文档类型的Nosql数据库，使用JSON格式去保存数据。CouchDB的字段只有三个：文档ID、文档版本号和内容。内容字段可以看到是一个text类型的文本，里面可以随意定义数据，而不用关注数据类型，但 数据必须以json的形式表示并存放。

4、CouchDB以RESTful API的格式提供服务，可以很方便地开发各种语言的客户端。

5、 CouchDB 中没有锁机制，它使用的是多版本并发性控制（Multiversion concurrency controlMVCC）— 它向每个客户机提供数据库的最新版本的快照。

6、CouchDB优势在于：JSON存储格式，方便移值到移动设备上去，方便同步到各个分布式结点。

7、 CouchDB缺点：不支持动态查询，需要事先建立视图（View），使用javascript Map/Reduce 计算这些视图的结果（MongoDB也支持JSON格式和javascript 操作）；不停追加版本，数据膨胀不压缩；局部更新限制。

8、 Damien Katz创建Apache CouchDB文档数据库并成立CouchOne公司，后与Membase公司（用于社交游戏）合并成立Couchbase公司，并综合两家公司的数据库所长。继承CouchDB的面向文档、索引、查询功能，并将Membase的可伸缩性、速度、集群与缓存特性添加到合并后的Couchbase产品中。

RestFul Api提供服务

建立文档数据库并写入json数据的例子：

- 请求 curl http://127.0.0.1:5984/ 返回：{"couchdb":"Welcome","version":"0.8.1-incubating"}
- 请求：curl -X GET http://127.0.0.1:5984/_all_dbs 返回：{}
- 请求：curl -X PUT http://127.0.0.1:5984/wiki 返回：{"ok": true}
- 请求：curl http://127.0.0.1:5984/wiki 返回：{"db_name": "wiki","doc_count": 0,.....}
- 请求：curl -X POST -H “Content-Type: application/json” --data \ ‘{ “text” : “Wikipedia on CouchDB”, “rating” : 5 }’ \ http://127.0.0.1:5984/wiki 返回{"ok": true,"id": "123BAC",...}
- 请求：curl -X DELETE http://127.0.0.1:5984/wiki 返回：{"ok": true}

通过写map/reduce查询

一个查询文档是否有附件的例子：

- 创建视图在数据库中聚合、连接和报告文档，通过JavaScript使用MapReduce方式实现。跟hadoop map/reduce类似。

```
map: function(doc) {
  if (doc._attachments) {
    emit("with attachment", 1);
  }
  else {
    emit("without attachment", 1);
  }
}
reduce: function(keys, values) {
   return sum(values);
}
```

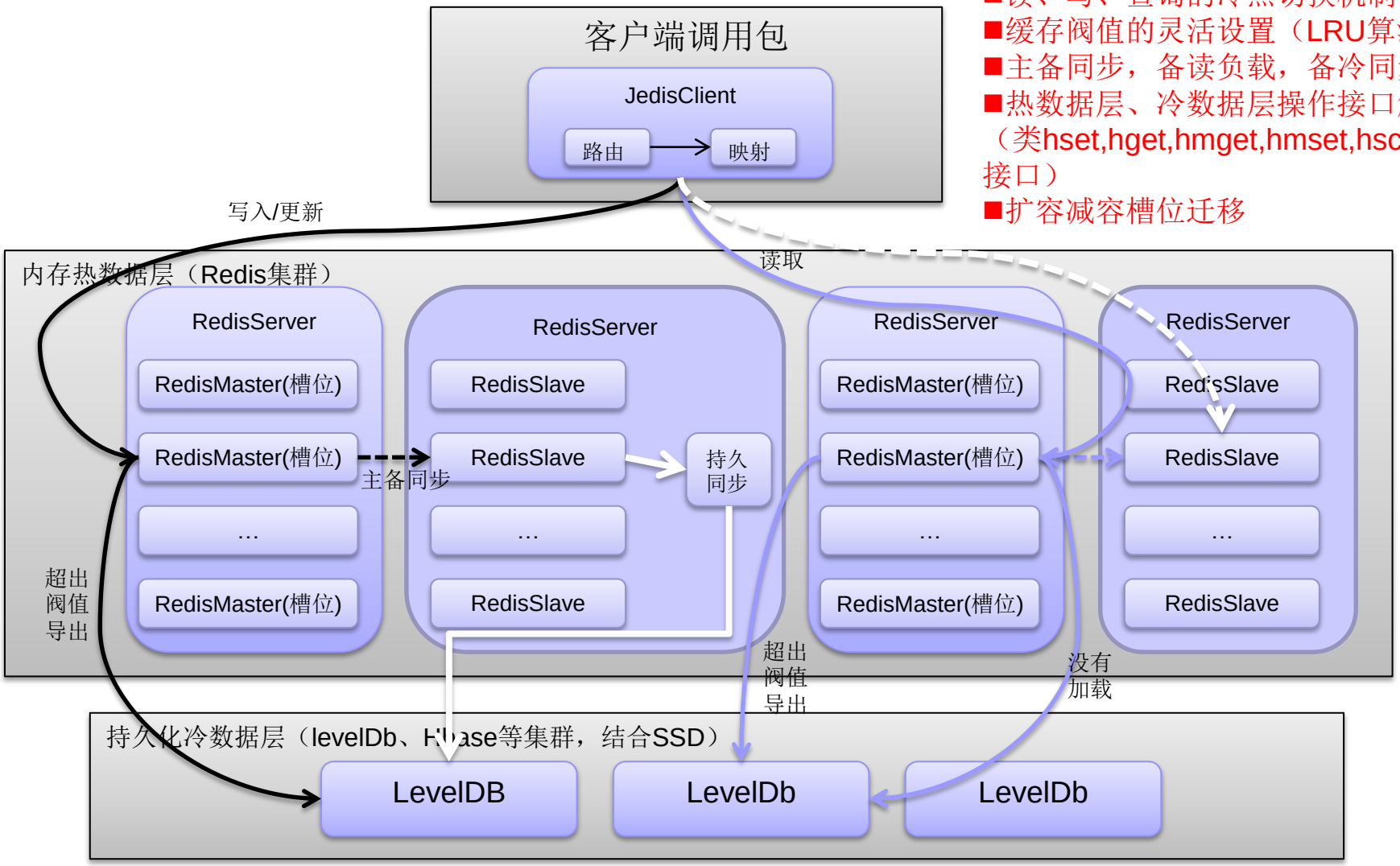
Redis代表的10万tps的kv存储性能突破

在单实例场景下，并发200，每并发5000条数据，客户端与服务端同机测试，不考虑网卡IO								
持久化方式	RDB				AOF			
数据大小（byte）	2	512	1024	10k	2	512	1k	10k
SET	137419.42	137045.4	131141.98	59829.11	123244.05	101990.28	81185.27	10270.85
INCR	141896.72	141279.92	142323.43	139525.11	125245.83	123408.68	117134.08	109475.37
LPUSH	140778.11	130798.72	121885.2	51340.97	124670.72	84322.47	58017.26	6503.5
SADD	144389.46	144218.14	144440.79	135211.3	145805.81	147316.51	132965.49	110068.52
磁盘写io（kB/s）	1515	7870	10804	24894	5287.33	106034.67	117708	142080

Cpu: Intel(R) Xeon(R) CPU E5530 @2.40GHz 8核	内存：48G	网卡： 1GEbit	硬盘： SATA					
单机8个redis实例	字节数	set	get	incr	lpush	lpop	sadd	spop
前提：关闭rdb和aof	2	691322.26	664494.07	644914.21	649081.65	638711.86	716315.95	803859.71
客户端：benchmark	64	655980.59	659405.65	646261.7	612469.29	631248.31	670238.93	754420.09
并发数：800	128	503854.24	599338.2	635299.8	558716.37	617490.15	665729.46	730435.92
单机8个redis实例	字节数	set	get	incr	lpush	lpop	sadd	spop
前提：开启rdb和aof	2	616692.08	664793.78	596612.89	590405.76	604063.58	665631.98	757115.32
客户端：benchmark	64	581107.46	657430.49	585783.91	529799.36	542416.83	648812.44	746168.36
并发数：800	128	503961.58	606372.62	577345.41	434216.58	478175.91	611693.27	729675.59

Redis缓存+持久化架构方案

- ■读、写、查询的冷热切换机制
- ■缓存阀值的灵活设置（LRU算法）
- ■主备同步，备读负载，备冷同步
- ■热数据层、冷数据层操作接口解耦（类hset,hget,hmget,hmset,hscan接口）
- ■扩容减容槽位迁移



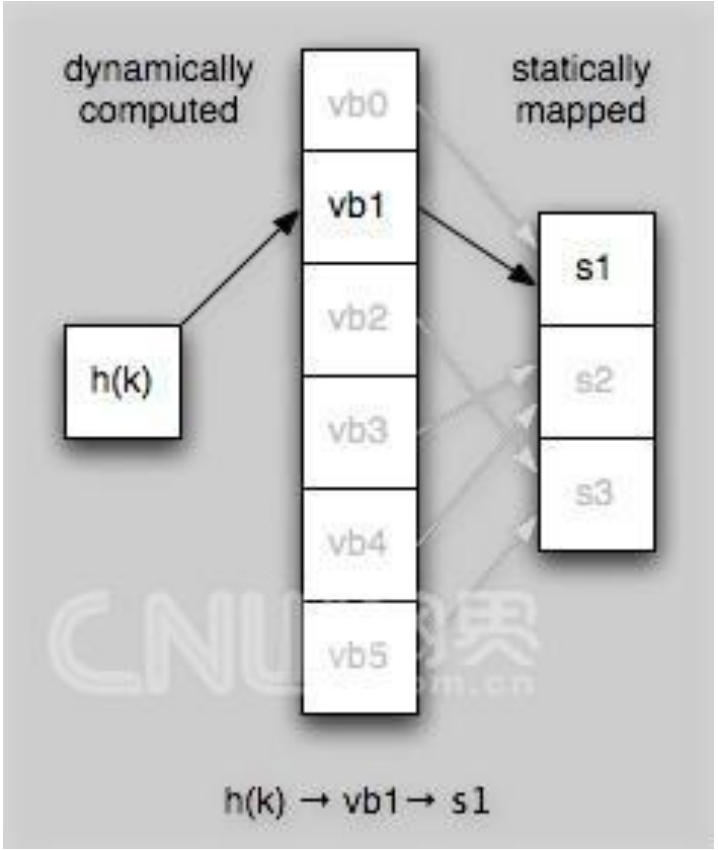
Redis本身有volatile-lru等6种随机抽样数据淘汰策略，但是删除丢掉，不放回冷持久存储层。Redis2.8以前版本有基于虚拟内存的冷热切换功能，现已废弃。

Redis hash slot算法原理

1、vBucket实现redis hash slot

2、按服务器取模：
servers = ['server1:11211', 'server2:11211', 'server3:11211']
server_for_key(key) = servers[hash(key) % servers.length]
服务器为单位粒度太粗，跟服务器耦合

3、按vbuckets取模
servers = ['server1:11211', 'server2:11211', 'server3:11211']
vbuckets = [0, 0, 1, 1, 2, 2]
server_for_key(key) = servers[vbuckets[hash(key) % vbuckets.length]]
细粒度，方便备份和迁移，管理好客户端和服务器的存储映射配置



Redis的性能瓶颈

redis在单机上性能能否突破10万tps，达到百万tps？

单实例的性能瓶颈：主要在于cpu的单核处理能力

经过单实例redis单客户端和高并发客户端的测试观察，内存、网卡（几十m字节收发每秒）都不是瓶颈，单纯的网络收发包返回不做处理就已经耗光单核cpu，但是其他核空闲，单实例仍然有很大优化空间，同样的物理配置，程序上多核设计可以突破10万tps，达到30-40万。所有物理指标数据都是程序设计产生的，如果单实例能突破到几十万的tps，那么多实例很容易就能过百万。

多实例的性能瓶颈：多实例的瓶颈在于cpu、网卡、和客户端并发数

提升cpu使用率的方式在单机上搭集群，加大redis实例数，发现实例上去了并发数上不去，并发上去了网卡上不去，导致CPU利用率不够。

如果一台在2288服务器物理机上，只有3个实例的集群，在200以内的客户端并发下，它的tps没有明显提升。加大到6个个实例，可以达到20万。如果加大到24个实例，在4台物理机上模拟将近2000多客户端并发，tps能提升到40-50万，但是cpu只有12核的利用率。如果减少到12个实例，效果是一样的，也就是一台物理机redis部署的实例数目前只能达到小于核数的利用，达不到线性增长。

一个原因就是网卡已经到极限，这时网卡已经接近128m的瓶颈，发包数也接近110万极限。

1、双网卡多网卡方案；2、增加物理机

另一个原因在实例节点数小的情况下，按数据随机的集群算法分配不一定完全均匀，导致多个实例中只有部分实例会处于频繁操作的状态，出现有的实例cpu核空闲。但是在实例节点数多的时候会趋于均匀。

Redis性能优化

1、运维调优：关闭节能、调整网卡参数、关闭超线程、绑定核数等，效果有限。

2、升级硬件：升级为Infiniband RDMA（100g，高带宽，低延迟），升级为2699服务器的3.6g主频的cpu，可进一步提升整体性能。

3、改进redis：服务端改成多核处理（这是最有效提升性能方式）；将多条命令打包节省网络交互开销，基于pipeline方式采用多key读写和查询接口，这在一定程度上能优化性能，但是也面临服务端处理能力、客户端并发数、集群支持等问题；利用LUA脚本动作聚合读写操作（集群不支持）。

总结和探索

redis的优势在于快速的缓存读写、丰富的数据类型支持，以及完善的主备复制和集群实现，劣势在于单线程模式、内存限制、查询检索等方面。 KyotoCabinet、MongoDB、levelDb等新型k/v存储都采用了缓存+虚拟内存+持久化技术，速度上能达到缓存效果，部分采用多线程设计，持久化能突破内存限制，并且一般采用b+树、hash等标准数据库存储结构设计，可以很好支持范围查询。持久化方式有利于结合ssd硬件优势，能很好满足更广泛的k/v存储需求。

那么，能否在一台普通物理机(24cpu(2*6*2)256g内存SAS/SATA)实现百万tps的吞吐量呢？
从物理特性上是可以做到的，但是需要改变数据库引擎软件设计。
下面是coolhash和redis的读写对比

100万每批写读	1客户端	20并发客户端*100万
Redis Pipeline	写7秒，约14.3万/秒 读6秒，约16.7万/秒	写97秒，约20.6万/秒 读114秒，约17.5万/秒
CoolHash (1工人)	写9秒，约11.1万/秒 读7秒，约14.2万/秒	写42秒，约47.6万/秒 读23秒，约86.9万/秒
CoolHash (24工人)	写3秒，约33.3万/秒 读2.5秒，约40万/秒	写14秒，约142.8万/秒 读11秒，约181.8万/秒

服务端的多核处理能力和客户端的批量写入能力是提升性能的关键因素，也是nosql k/v数据库的技术发展趋势。

CoolHash是什么

CoolHash是原创数据库引擎，高度产品化，易用性强，容易嵌入使用和复制传播（200k大小），采用apache2.0开源协议，使用java实现(jdk1.7)，对外提供java接口，同时支持windows和linux（unix-like）。

CoolHash做什么

只做数据库最基础核心的引擎部分，支持大部分数据类型的“插入、获取、更新、删除、批量插入、批量获取、批量更新、批量删除、高效查询（精确查、模糊查、按树节点查、按key查、按value查）、分页，排序、and操作、or操作、事务处理、key指针插入和查询、缓存持久互转”等操作和远程操作。

CoolHash不做什么：

“监控、管理、安全、备份、命令行操作、运维工具”等外围特性都剥离出去不做，分布式扩容、分库分表等集群特性也不做。

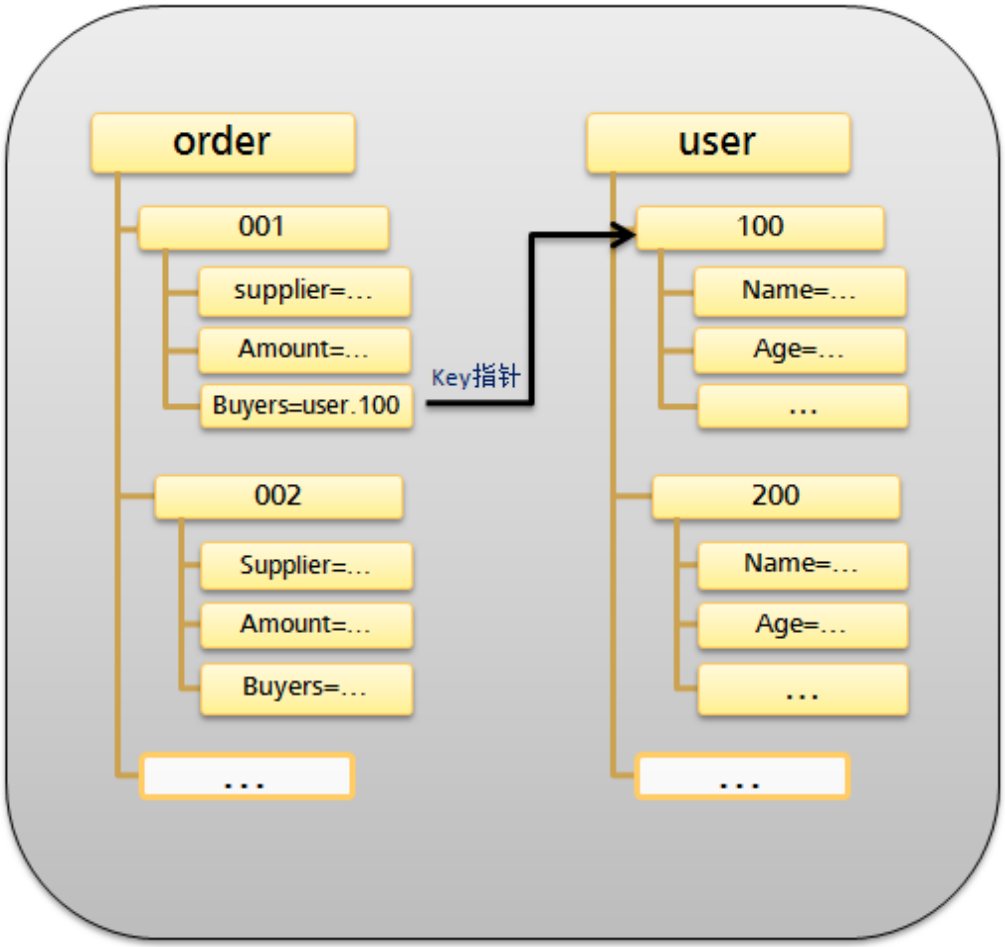
1、CoolHash是一个数据库引擎
2、CoolHash是一个k/v数据库
3、CoolHash是一个并行数据库（mpp）
4、CoolHash是一个nosql数据库
5、CoolHash实现了简单事务处理
6、CoolHash是一个数据库Server

CoolHash数据逻辑结构

order表	supplier	amount	buyer
001	…	…	…
002	…	…	…

关系表结构转换成CoolHash

user表	name	age	…
100	…	…	…
200	…	…	…



以树型结构设计key，比如：<db>.<table>.<id>.<prop>，用key point解决关联关系：

用户001：user.001.age=10，user.001.name=tom，user.001.order=order.001(key point)，...

用户002：user.002.age=25，user.002.name=rose，user.002.children0.age=1，...

订单001：order.001.amount=1000，order.001.buyer=user.001(key point)，…

即可独立获取也可以范围查询：

以user.001.*的方式只查询某个父节点，获取用户001所有属性/值，并可匹配条件模糊查

以user.*.name的方式只查询所有子节点，获取所有用户的名称，并可匹配条件模糊查。

读写常规操作

key和value的数据格式：CoolHash的key只能是字符串，默认最大长度为256字节。value能支持非常广泛的数据类型，基本数据类型“String（变长字符）、short（短整形）、int（整型）、long（长整形）、 double（双精度浮点型）、 float（浮点型）、Date（日期型）”， 高级数据类型的大部分的java集合都能支持（List、Map、Set等），以及任意可序列化的自定义java类型，底层数据类型也可以支持二进制型。

不需要安装配置，服务端启动CoolHashServer，指定好ip和端口，客户端编程步骤：

```
CoolHashClient chc = BeanContext.getCoolHashClient("localhost",2014);//连接CoolHashServer
chc.put("user.001.name","zhang");//写入字符
chc.put("user.001.age",20);//写入整数
chc.put("user.001.weight",50.55f);//写入浮点数
chc.put("user.001.pet",new ArrayList());//写入集合对象

String name = (String)chc.get("user.001.name");//读取字符
int age = (int)chc.get("user.001.age");//读取整数
float weight = (float)chc.get("user.001.weight");//读取浮点数
ArrayList pet = (ArrayList)chc.get("user.001.pet");//读取集合对象
```

以上是单条操作，但是建议一次读写一个HashMap的集合数据，批量操作更能发挥并行计算优势和节省网络开销。

模糊查询操作

基本数据类型的value可支持按内容查询，高级数据类型和二进制类型不支持按内容查询。

```
CoolKeyResult keyresult = chc.findKey("user.001.*");//查找用户001的所有属性
CoolKeySet ks = keyresult.nextBatchKey(4);//分页获取前4条结果
System.out.println(ks);//输出[user.001.weight, user.001.age, user.001.name, user.001.pet]

CoolHashResult mapresult = chc.find("user.*.age", ValueFilter.greater(18));//查找年龄大于18岁的用户
CoolHashMap hm = mapresult.nextBatch(10);
System.out.println(hm);//输出[user.001.age=20, user.002.age=25]
```

……

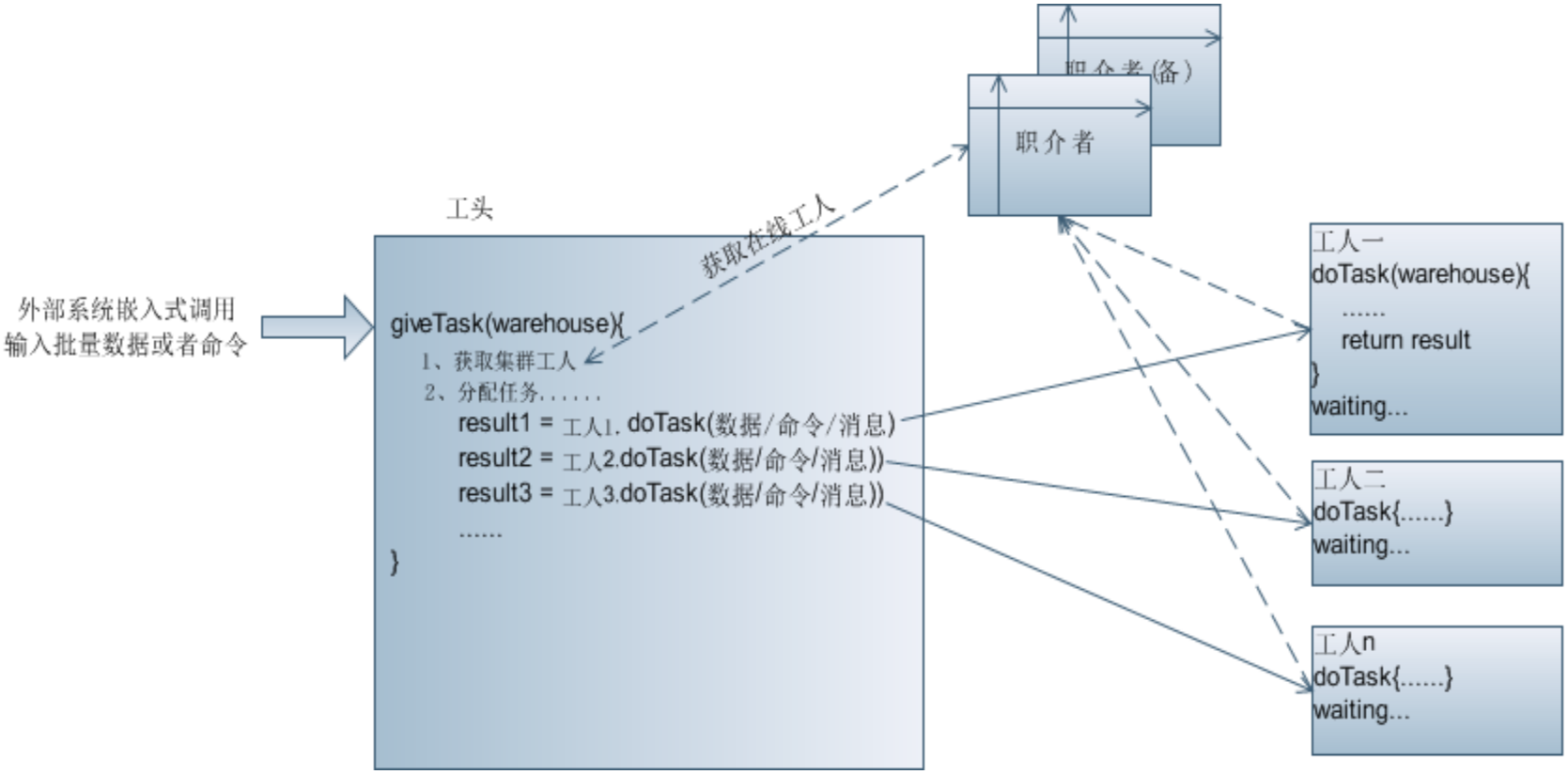
更多的功能使用请去参考开发包里的demo

CoolHash的测试性能

读写吞吐量超过百万，千万级别查询1秒完成，连续48小时打满CPU强压力运行稳定

- CoolHash的单条写入和读取速度都在毫秒级别，写和读差别不大，读略快于写。
- CoolHash的批量写入和读取速度都控制在秒级别，100万数据写入基准测试，普通台式机或者笔记本（4核4g）需要5-6秒，标准pc server(24核256g)需要2-3秒；批量读、批量删除和批量写入的速度差不多。
- CoolHash写入缓存和写入持久的速度差别不大，100万数据写入缓存基准测试，普通台式机或者笔记本（4核4g）需要5秒左右，标准pc server(24核256g)需要2秒左右。如果是10万级别的数据读写，缓存和持久的速度大致接近等同。
- CoolHash的查询速度控制在秒级别，100万数据的模糊查询（如like%str%）在没有构建索引情况下，普通台式机或者笔记本（4核4g）需要2-3秒，标准pc server(24核256g)需要1-2秒；如果是重复查询，由于CoolHash内部做了数据内存映射，第二次以后只需要毫秒级完成。
- 高并发多客户端的吞吐量总体速度要快于单客户端，但是受服务器cpu、内存、io等性能限制，会倾向于一个平衡值。

基于Fourinone并行计算



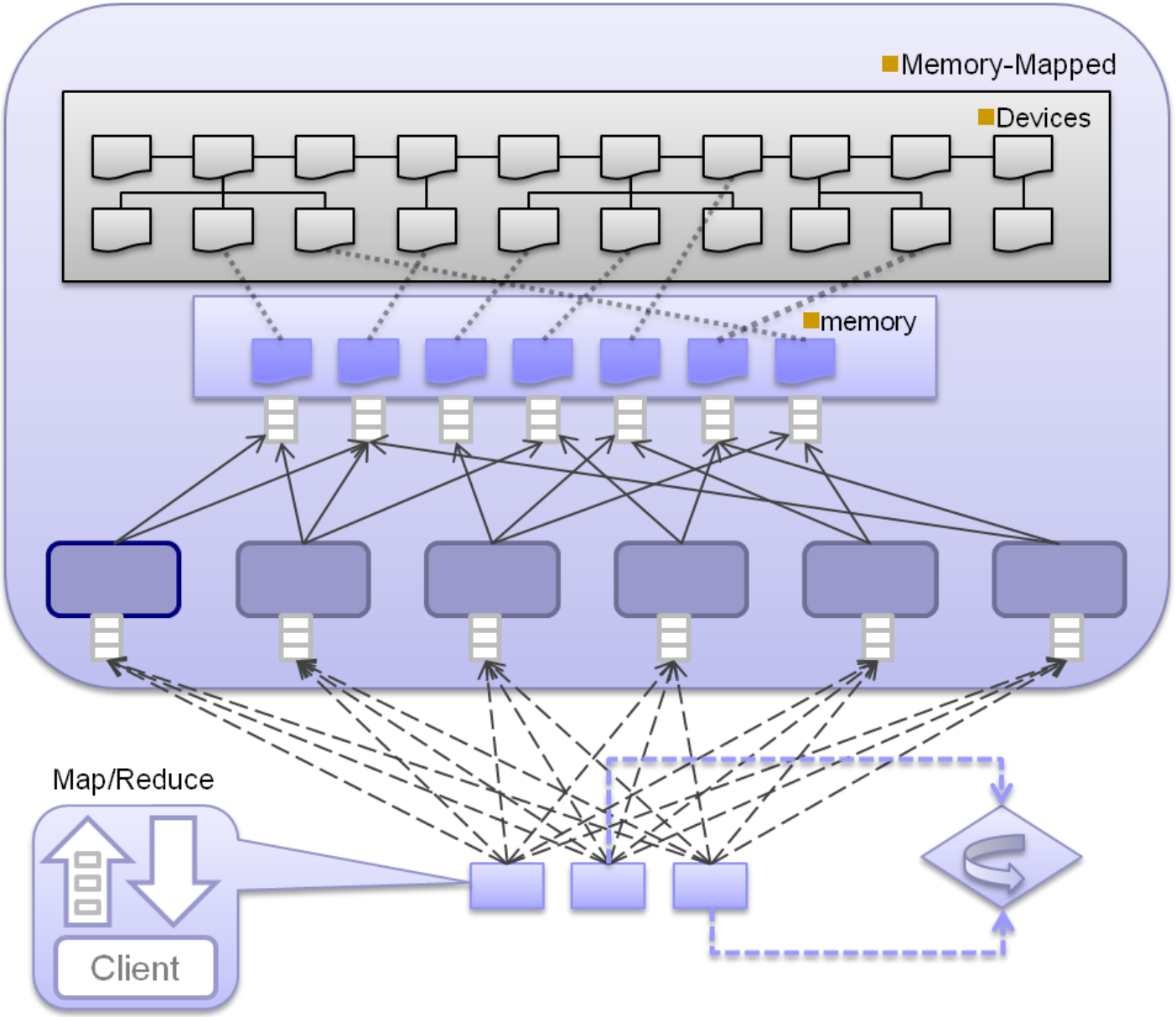
实现一个简化的分布式并行计算结构（抽象出几个角色+几种接口去满足大部分并行计算需求）

包工头去服务化，嵌入式，负责分配任务，开发者实现分配任务接口 如何容灾？

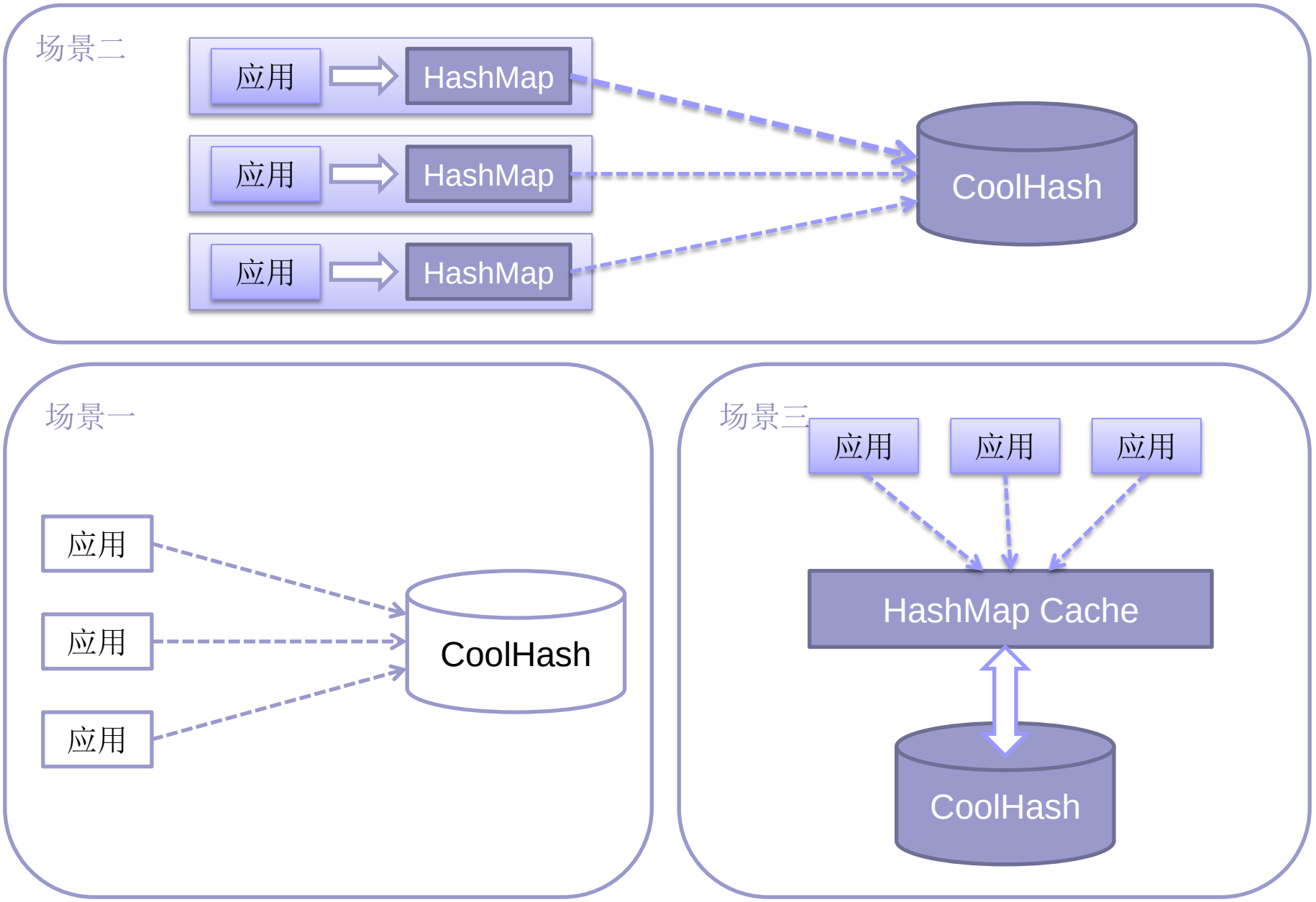
农民工负责执行任务，开发者实现任务执行接口 如何容灾？

职介者负责协同一致性等处理（登记，介绍，保持联系） 如何容灾？

CoolHash并行架构示意图



CoolHash使用场景



linux内存映射

1、文件映射是虚存的中心概念，用户像操作内存一样操作文件

2、传统i/o非常低效，open系统调用打开文件，然后使用read，write调用进行操作，每次I/O操作都需要一次系统调用。多个进程操作同一个文件，每个进程都要维持副本浪费内存空间(不适合并行化)。文件内存映射方式能很好的利用内存空间，并且能将多个进程的随机操作合并成顺序读写大大提升效率。

3、进程的文件内存映射如下（应用程序调用mmap()）：可以看到两个进程的共享内存拥有相同地址

08048000-0804f000 r-xp 00000000 08:01 573748 /sbin/rpc.statd #text

start_end perm offset major:minior inode

start：该区域的起始虚拟地址

end:该区域的结束虚拟地址

perm:读、写和执行权限

offset:被映射部分在文件中的起始地址

major minor :主次设备号

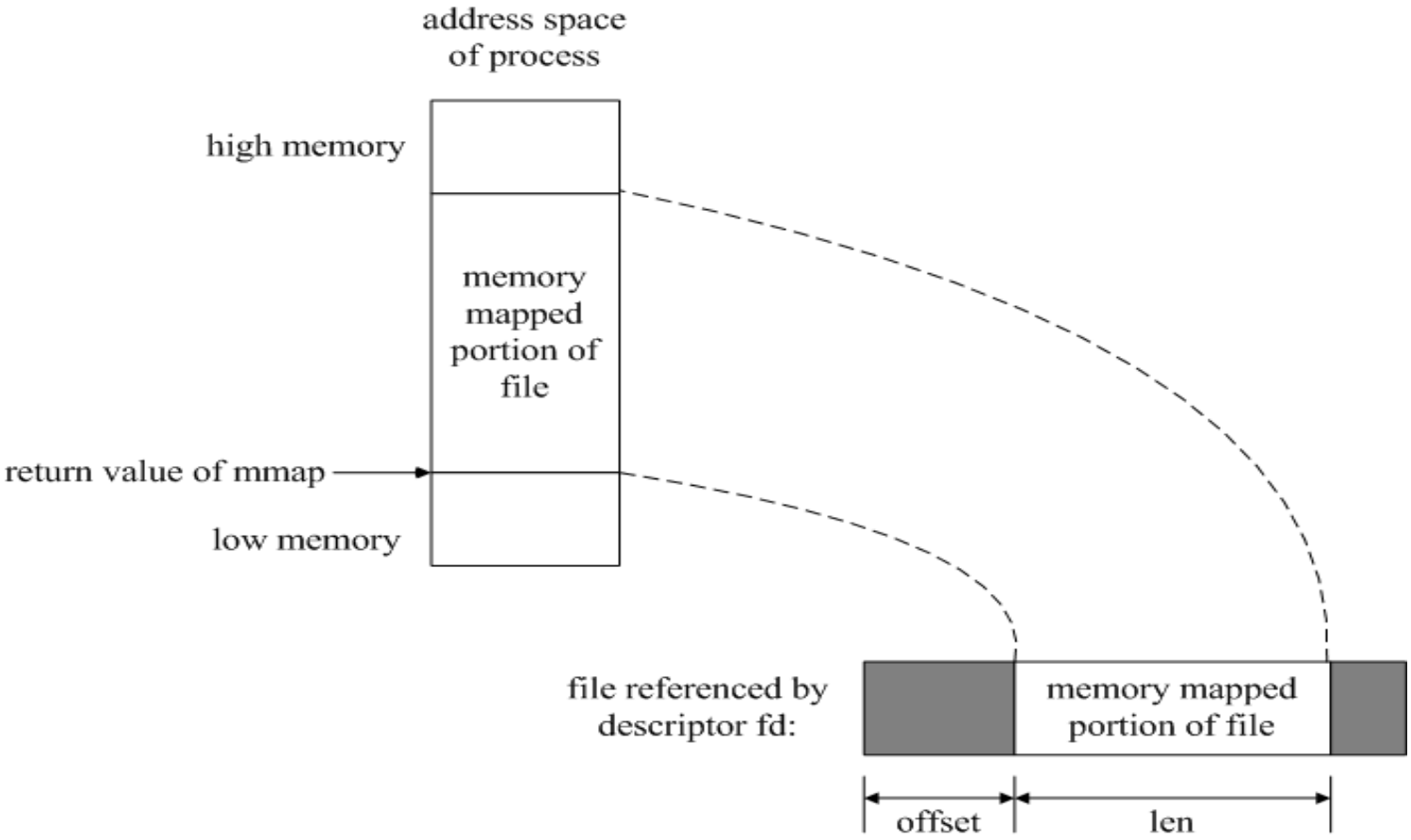
inode:索引结点

linux内存映射

void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);

struct vm_area_struct

1、内核内部的虚拟内存管理系统通过应用程序的调用的mmap()的变量，生产vm_area_struct结构体，从而内存映射，然后初始化该结构体的域，最后才传递的过程。2、应用程序锁调用mmap()函数中使用的多数变量值由vma内核处理为vm_area_struct结构体形式，然后传递到设备驱动程序上。



提升CPU并行化可以获取更高的性能

通过调整“coolhash数据工人数量、客户端并发数量、每客户端读写数量”三个指标项达到一台服务器的最佳吞吐量性能

测试1：x个数据工人，200个客户端并发，每个客户端写入10万数据：

数据工人	1个工人	8个工人	24个工人	32个工人	96个工人
耗时	400秒	40秒	20秒	23秒	25秒
cpu最高峰	10%	20%	75%	85%	90%
TPS	5万/秒	50万/秒	100万/秒	87万/秒	80万/秒

分析：可以清晰的看到并行数据库的优势明显，如果只有一个数据工人，也就是单进程模式，它的TPS是很难超出10万的，如果是8个数据工人并行作业，性能一下子就能从400秒减少到40秒，提升10倍，但也不是数据工人越多性能越好，我们看到24-32个工人是个顶峰，再增加工人数虽然能提升cpu使用率，但是调度开销大，后端硬盘io等跟不上，导致性能反而有所下降。

提升访问并发量可以获取更高的性能

测试2：24个数据工人，x个客户端并发，每个客户端写入10万数据：

客户端数	1并发	10并发	20并发	50并发	100并发	200并发
写入总量	10万	100万	200万	500万	1000万	2000万
耗时	1秒	3秒	5秒	10秒	18秒	20秒
TPS	10万/秒	33万/秒	40万/秒	50万/秒	55万/秒	100万/秒

分析：如果每个客户端写相同数量的数据，随着并发数的提高，总体的吞吐量会高于单客户端呈线性增长趋势，但是受服务器cpu、内存、io等性能限制，不会一直增长，会倾向于一个平衡值。每台服务器并不是能承受无限大的并发数量，如果超出了承受限制，客户端会长时间等待，容易产生socket连接超时。合理的控制并发数量能提升服务器的吞吐性能，下面我们增大每个客户端的写入数量，减少总的并发数，并观察效果。

提升每个请求的写入量可以获取更高的性能

测试3：24个数据工人，20个客户端并发，每个客户端写入x万数据

每客户端写	10万/每	50万/每	100万/每	200万/每	300万/每
写入总量	200万	1000万	2000万	4000万	6000万
耗时	4秒	6秒	10秒	15秒	22秒
写TPS	50万/秒	167万/秒	200万/秒	267万/秒	272万/秒

分析：可以看到同样写入2000万数据，采用20并发*100万比200并发*10万的性能提升了一倍，能达到200万以上TPS。这是因为客户端建立连接后，一次提交100万条数据的写入请求，相比每条数据连接server，能很大节省网络开销和硬盘IO开销。由此我们也能得到，并不是并发连接越多越好，而是控制一定数量的连接池性能会更好。

提升每个请求的读出量可以获取更高的性能

测试4：24个数据工人，20个客户端并发，每个客户端读出x万数据

每客户端读	10万/每	50万/每	100万/每	200万/每	300万/每
读出总量	200万	1000万	2000万	4000万	6000万
耗时	4秒	6秒	11秒	20秒	30秒
读TPS	50万/秒	167万/秒	182万/秒	200万/秒	200万/秒

分析：coolhash是一个读写平衡的数据库引擎，可以看到读和写的性能相差不大，都能达到200万的TPS。

缩小范围并行查询可以获取更高的性能

测试5：24个数据工人，x个客户端模糊查询x万数据

单个客户端模糊查询：					
数据总量	600万	2000万	6000万	1亿	3亿
耗时	0.9秒	1秒	1.7秒	2秒	6秒

多个客户端高并发模糊查询：						
客户端数	1并发	10并发	20并发	50并发	100并发	200并发
600万	0.9秒	2秒	4秒	9秒	14秒	18秒
2000万	1秒	4秒	9秒	21秒	37秒	45秒

分析：对于千万级别的数据，客户端一次任意模糊查询的耗时都是在1秒完成，如果超过1亿需要2秒，3亿需要6秒（在没有额外构建索引情况下）。如果是100个客户端并发模糊查询，按照上面表格里100并发查询2000万数据，平均耗时37秒，每次查询耗时37/100=0.37秒，每秒查询次数100/37=2.7次，每秒查询数据范围大小100*2000万/37秒=5400万。这里测试key是“*”代表所有，如果key根据业务特点进行了分层设计，以“user.*.name”这样形式模糊查询，范围会缩小，速度会更快。

提问/交流

技术博客：
http://fourinone.iteye.com

邮箱：
fourinone@yeah.net

国内oschina code:
https://git.oschina.net/fourinone/fourinone/blob/master/fourinone-4.05.06.zip

国内csdn code：
https://code.csdn.net/fourinone/Fourinone/tree/master/fourinone-4.05.06.zip

Thank You