**大量设备在线状态监控及快速检索设计方案**

**场景**

一家物联网公司，其业务主务涉及百万设备在线状态监控。其组织架构如下：



总公司下在有多个分公司，各个分公司下面有设备，各个分公司管理自己的设备。

设备有在线，离线状态，异常状态（这时设备是在线的，不过设备不正常工作），设备与通信服务器每分钟进行一次心跳检测，并上报设备健康状态。

设备有很多种，如温度传感器，湿度传感器，PM2.5传感器，风速传感器，光传感器等等。

设备控制台界面可以查询哪些类型的设备在线，哪些类型的设备离线，哪些类型的设备异常，并且可以实现分页查询。如下图，状态，类型，客户等条件可以随意组合查询，查询条件要求2s内返回单页100条数据查询。



**分析**

针对上面的需求，我们仔细分析一下有这么几个棘手的问题要解决：

1.设备健康状态的存储和上报要尽可能的高效，由于设备有上百万台，如果我们将设备状态存储在数据库中，我们可以大体计算一下，1分钟一次状态上报，100万台设备，平均每秒钟就是100万/60=1.7万次，每一次状态上报我们就要去update一次库，数据库直接挂掉，即使我们做一下优化，批量去更新，每批更新100条，那每秒钟也要170次批量更新，对数据库来说，也是无法承受的。同时也带来另外一个重大问题，update会产生大量排它锁，导致其它更新业务无法进行。

2.针对 第1点 所产生的问题，有人提出使用Redis来进行解决，把所有设备的状态存储在一个Redis 哈希表中,key是设备Id，值是设备状态，分别用1,2,3来表示设备正常，离线 ，异常。这样是可以解决第1点中设备状态更新的问题，redis强大的性能完全可以满足每秒钟1.7万条数据的更新，不过这虽然解决了第1点中的问题，减小了数据库的压力，但是同时也带来另外一个同样棘手的问题：怎么实现不同条件的组合查询，例如我要分页查询公司1下面在线的湿度传感器。

3.针对第2点引出来的问题，我们分析一下，如果按第2点设计我们通常是怎么查的，有两个思路，我们看一下是否可行：

1.首先从redis中查找100条在线的设备，然后拿100设备的Id到数据库中查询这100条设备详情，但是这100条设备并不全是公司1的，也并不一定全部是湿度传感器，也有可能是风速传感器，如果这一次查询只筛选出10条符合要求的记录，那你岂不是又要去Redis里面去继续再查询100条在线设备，然后再去 Mysql中查询出设备信息再比较，依此不断循环，直到找到100条符合要求的数据出来。这个方式充满了不确定性，最坏的情况就是只有一台设备在线，而且该设备还在缓存的最后一个，如果这样岂不是要查询100万次才能查到最后一台设备。有人说我可以在Redis存储结构中做一下改进，我把公司Id，设备类型单独拿出来组成一个类似于下图映射关系，即将Redis中的大表拆成小表，拆法如下：



在上图中我们使用了两个hash表来存储不同公司的数据 如果我要查询公司1中的数据，我就从公司1的hash表中查找，如果我要查询公司1下的在线设备。那查找方式还是和上面的操作一样，无非就是之前所有公司在一块，现在只有一个公司，筛选方式还是那么愚蠢。如果总公司想查看所有公司在线湿度传感器的前100条，那我岂不是还要将公司1中和公司2中的在线湿度传感器查出来再合并起来，然后再重复筛选动作，有人说......总之就是一堆不靠谱的设计。

2.先从mysql中查询某个公司下面的某种类型的设备。然后再去redis验证在线状态，直到找到100条。这种方式跟上面的方式如出一辙，无非是先查redis还是先查Mysql问题，最终解决不了的就是怎么精确的取出这100条。

**解决方式**

上面的方式都不可用，难道真的就没有好的解决办法了吗？当然有！Java 有一个类叫做 BitSet,该类是解决此类问题的灵丹妙药。具体BitSet的原理请自行百度。下面是给出本例的状态矩阵图，说白了实际上就跟我们Oracle的位图索引功能一样。



如上图，我们用bit 值 0和1来表示设备的状态。例如本例中，设备分为在线，异常。我们按公司，设备类型将设备分别装到不同的BitSet中去，例如，BitSet1表示的是公司\_1下的温度传感器的在线状态，1表示在线，0表示离线，BitSet11表示的是公司\_1下的温度传感器的异常状态，1表示异常 0 表示正常。

bit的下标就是我们对应的设备Id，当然这种设计时，设备表的Id必须是整型。设备Id同时也表示bit位的下标。

如果我们查询公司1，温度传感器和湿度传感器在线的设备，那么就取 BitSet =( BitSet1 OR BitSet2 ) ,再对BitSet遍历，取出bit为1的值，就是在线设备。整个运算都是在内存级别做的，所以速度快。

下面我们用代码模拟一下：

System.***out***.println(**"==============================开始模拟加载数================================================="**);  
  
System.***out***.println(**"......初始化开始......"** + **new** Date().getTime());  
  
*/\*\*  
 \* 模拟初始化加载数据，实际中这里是要从数据库中加载的  
 \*/***for** (**int** j = 0; j < 1; j++) {  
 Map<String, BitSet> m = **new** HashMap<>();  
 BitSet bitSet\_t = **new** BitSet();  
 m.put(**"温度传感器"**, bitSet\_t);  
 BitSet bitSet\_h = **new** BitSet();  
 m.put(**"湿度传感器"**, bitSet\_h);  
 BitSet bitSet\_p = **new** BitSet();  
 m.put(**"PM2.5传感器"**, bitSet\_p);  
 *map*.put(**"公司\_"** + j, m);  
 **for** (**int** i = 0; i < 1000000; i++) {  
 **if** (i % 3 == 0) { *//如果是3的倍数我们就认为它在线* bitSet\_h.set(i);  
 }  
  
 }  
 **for** (**int** i = 1000000; i < 2000000; i++) {  
 **if** (i % 7 == 0) {*//如果是7的倍数我们就认为它在线* bitSet\_t.set(i);  
 }  
  
 }  
 **for** (**int** i = 2000000; i < 3000000; i++) {  
 **if** (i % 19 == 0) {*//如果是19的倍数我们就认为它在线* bitSet\_p.set(i);  
  
 }  
 }  
}  
System.***out***.println(**"bitSet\_h 在线个数:"** + *map*.get(**"公司\_"** + 0).get(**"温度传感器"**).cardinality());  
System.***out***.println(**"bitSet\_t 在线个数:"** + *map*.get(**"公司\_"** + 0).get(**"湿度传感器"**).cardinality());  
System.***out***.println(**"bitSet\_p 在线个数:"** + *map*.get(**"公司\_"** + 0).get(**"PM2.5传感器"**).cardinality());  
  
System.***out***.println(**"......初始化结束......"** + **new** Date().getTime());  
  
  
System.***out***.println(**"==============================开始模拟分页查找================================================="**);  
  
Date searchStart = **new** Date();  
List<Integer> indexEqualsOne = **new** ArrayList<>();  
BitSet bitSet = (BitSet) *map*.get(**"公司\_"** + 0).get(**"温度传感器"**).clone();  
bitSet.or(*map*.get(**"公司\_"** + 0).get(**"温度传感器"**));  
bitSet.or(*map*.get(**"公司\_"** + 0).get(**"湿度传感器"**));  
bitSet.or(*map*.get(**"公司\_"** + 0).get(**"PM2.5传感器"**));  
System.***out***.println(**"总共多少条： :"** + bitSet.cardinality());  
  
  
**int** start = 2000;  
**int** pageSize = 100;  
**int** end = start + pageSize;  
**int** count = 0;  
**for** (**int** i = 0; i < bitSet.size(); i++) {  
  
 **if** (bitSet.get(i)) {  
 count++;  
 }  
 **if** (count > start && count < end) {  
 indexEqualsOne.add(i);  
 }  
 **if** (indexEqualsOne.size() == pageSize) {  
 **break**;  
 }  
 ;  
}  
Date searchEnd = **new** Date();  
  
System.***out***.println(**"查询从第"** + start + **"到第"** + end + **"条记录总共+耗时:"** + (searchEnd.getTime() - searchStart.getTime()) / 1000.0 + **"s"**);  
  
*/\*\*  
 \* 查找我们可以把它封装成方法  
 \*/*System.***out***.println(**"结果为："**+indexEqualsOne);

我们放置了1000000万条数据，然后对其进行分析模拟查找，结果如下：



我们可以看到总耗时只有0.005s 这个速度简直是快到了极致。

我们看一下内存消耗，不到25M。这也是BitSet固有的数据结构决定的。

