VictoriaMetrics阅读笔记 - 知乎

前言

由于组内搭建了一套VictoriaMtrics(简称为vm)的时序数据存储系统，好奇存储内部是如何实现的，所以打算看下源码，目前在github（https://github.com/VictoriaMetrics/VictoriaMetrics）上有两个版本，一个是单点版本（vminsert，vmstorage，vmselect都是部署在单个节点上的），一个cluster版本（vminsert，vmstorage，vmselect分开部署）。cluster版本要切为cluster分支（原本看单点版本看的一脸懵逼==，好险最后发现其实代码逻辑都是差不多的）

cluster版本的vminsert

vminsert的作用是负责把上游的过来的数据（http）做一些解析/过滤之类的活之后通过rpc把数据发送给vmstorage

1. 接受客户端的请求
2. 根据lables哈希得到存储节点（这里用了一致性哈希）
3. 然后flush内存把buffer发到对应节点

cluster版本的vmstorage

vmstorage是负责存储数据的节点，接受vminsert和vmselect的请求，进行存储/查询

因为从vminsert接受的数据里并没有打上TSID之类的信息，所以vmstorage需要先把对应的TSID才能找到对应应该存储的地方。TSID是一条时间序列的唯一标识。TSID的注入会涉及到查询（相同的label的曲线需要有相同的TSID），同时vmstorage为了加速TSID的查找配置了内存缓存等一些加速措施。

```
type TSID struct {
    // 项目和账户，cluster版的vmstorage支持多租户的存储。
    AccountID uint32
    ProjectID uint32
    // 有相同label keys的数据会有相同的MetricGroupID
    MetricGroupID uint64
    JobID uint32
    InstanceID uint32
    // 自增ID，这个自增是单节点的自增
    MetricID uint64
}
```

1. openstorage会加载/data目录下的表
2. data表有个addRow方法
3. data表中有很多个partition，每个partition的区分是时间范围
4. 把row插入的时候要看看是否有对应时间的partition
5. 有的话就执行partitionwrapper.partition.addRow()（最理想的情况就是所有的row都在一个partition里面）
6. 第二理想的情况就是rows虽然没有都在一个partition里面，但是都能在已有的partition里面找到对应的partition，相比最理想的情况多了一个split的步骤
7. 最不理想的情况下就是没有找到对应的partition，需要创建新的partittion（新的partition有small和big目录），创建新的partition会开一些协程（存储的时长是咋确定的，应该是根据配置参数来确定？）

```
pt.startMergeWorkers()
pt.startRawRowsFlusher()
pt.startInmemoryPartsFlusher()
pt.startStalePartsRemover()
```

1. 有了对应的partition之后就可以Addrow了，Add到哪里呢？victoriametircs的存储引擎是类似与LSM树的，所以不会直接写到磁

盘上，而是会先写到内存中。每个partition都维护一个内存的rawRows，现在这个数据就是加到这里面的。

2. partition的rawRow是什么呢？是一个内存shards，类名叫做rawRowsShards，名字就可以看出来有很多分片，很多分片要怎么加呢？自然就是hash，不过这里并没有用到一致性hash，就是普通的取模hash足以，毕竟shard的数量是参数指定的，并不会出现节点离开的情况。
3. 对应的partition.rawRowShard加入新的rows数据之后，会执行一次flush，也就是flushRowsToParts这个函数，这个函数是挂载在partition上的，而不是rawRowShard上，这个函数顾名思义就是把内存中的rows数据刷到part上，接下来我们看看是怎么刷的
4. 首先会先把row刷到inMemoryPart上，如何把rows数据转为一个part数据呢？
    1. 先对rows数据根据(TSID,timestamp)的顺序进行排序
    2. 然后遍历加到rrm（rawRowsMarshaler）中
    3. 然后调用`tmpBlock.Init(tsid, rrm.auxTimestamps, rrm.auxValues, scale, precisionBits)`得到一个实例化的tmpBlock，这个block里面存的timestamp和value都是int64，value是放大后的结果，还存了个scale来表示放大的倍数。
    4. 有了block之后，接着执行`rrm.bsw.WriteExternalBlock`（说实话，我还没找到这个bsw是在哪实例化的，有点奇怪..）bsw也就是blockstreamwriter。先是更新一波block的原数据，blockOffset、blockSize之类的，然后再把block中的数据序列化为二进制。
    5. 转换为二进制之后，这个有个优化点，如果前后两个块的timestampdata是一样的，timestampdata就不会再存一份了，这里可以剩下一些空间。（这里你可能会有疑问，为啥可以这样做呢？不是一个时间戳对应一个value吗？实际上是因为出了value和timestamp数据之外，vm还存了一份indexData数据，数据里面记录了timestampblockOffset，当我们查询的时候就可以拿到对应block的offset了）
    6. 最后就是io.writer，至于写到哪里去其实我也不知道，正如上面说的bsw不知道是咋实例化的。(姑且先当作是写到页缓存中吧)，然后inMemeoryPart就已经写完了。

```
type inmemoryPart struct {
  ph partHeader

  timestampsData bytesutil.ByteBuffer
  valuesData     bytesutil.ByteBuffer
  indexData      bytesutil.ByteBuffer
  metaindexData  bytesutil.ByteBuffer

  creationTime uint64
}
```

1. `p, err := mp.NewPart()`，有了这个inMemoryPart之后会调用NewPart方法，这个part就是可搜索的part了，这个part因为还只是没有经过merge的part，在vm里面是属于small part，这也是为啥可以看到有small和big两个目录，意思也很简单，一个是用来存小part的，一个是用来存大part
2. 每个partition都会维护一个smallparts列表，上面12创建的part就会加到这里面，当smallparts的长度超出阈值之后，就会执行merge，默认的阈值是256
3. `mergeSmallParts`这个是执行merge part的函数，首先会判断是merge为small part还是big part，然后执行`pt.mergeParts(pws, pt.stopCh)`。
4. 这次就是merge到文件中去了，由于merge的时候要保证事务性，所以会用到txn和tmp文件夹，核心的逻辑在`mergeBlockStreamsInternal`这个函数里面
   1. 会先把所有的small part丢到一个堆里去，按照part header的TSID排序
   2. 然后按照顺序merge，merge的流程就是比较pendingBlock和当前要merge进pengdingBlock的block，看看怎么合并到一起，合并的具体逻辑在`mergeBlocks`这个函数
   3. 然后删除old part
   4. 如果merge的结果还是small part就把merge的这个part加到smallparts中去

```
func mergeBlocks(ob, ib1, ib2 *Block, retentionDeadline
int64, rowsDeleted *uint64) {
    // 最快的路径，block1的最大时间小于block2的最小时间，直
接add合并就可以，不用遍历
        if ib1.bh.MaxTimestamp < ib2.bh.MinTimestamp {
                // Fast path - ib1 values have smaller
timestamps than ib2 values.
                appendRows(ob, ib1)
                appendRows(ob, ib2)
                return
    }
    // 第二种情况，如上
        if ib2.bh.MaxTimestamp < ib1.bh.MinTimestamp {
                // Fast path - ib2 values have smaller
timestamps than ib1 values.
                appendRows(ob, ib2)
                appendRows(ob, ib1)
                return
        }
    // 这里是用于非法数据的判别
        if ib1.nextIdx >= len(ib1.timestamps) {
                appendRows(ob, ib2)
                return
        }
        if ib2.nextIdx >= len(ib2.timestamps) {
                appendRows(ob, ib1)
                return
        }
        for {
        // 这里就是常规的两个有序数组的合并过程
                i := ib1.nextIdx
                ts2 := ib2.timestamps[ib2.nextIdx]
                for i < len(ib1.timestamps) &&
ib1.timestamps[i] <= ts2 {
                        i++
```

```
                }
                ob.timestamps = append(ob.timestamps,
ib1.timestamps[ib1.nextIdx:i]...)
                ob.values = append(ob.values,
ib1.values[ib1.nextIdx:i]...)
                ib1.nextIdx = i
                if ib1.nextIdx >= len(ib1.timestamps) {
                        appendRows(ob, ib2)
                        return
                }
                ib1, ib2 = ib2, ib1
        }
}
```

感受

1. 代码里基本上所有类型都使用了sync.Pool来减少gc的负担
2. 变量命名缩写很多，看的时候经常需要回看变量的意思
3. 利用了很多指针赋值，导致无法用ctrl跳转到引用的地方
4. 由于代码里对于block、part、partition的概念没有比较清晰的解释，可能会像常规的理解那样，多个block组成part，实际上block可以理解为一个虚拟的读取和写入的单元，比如mergepart的时候就会用到BlockStreamReader，这个reader会把memory part或者file part读取出来，用于合并。

优点

1. 支持prometheus远端写
2. 良好的并发限制
3. 更低成本的存储，独特的label索引存储
4. 特殊的时序数据存储结构，按metric+timestamp排序，更加符合时序数据按时间范围的查询需求

缺点

1. 缺乏高可用机制（这里的高可用是指raft之类分布式协调机制），这个缺点比较致命，只能通过部署从节点减少宕机的影响
2. vminsert没有对pull的数据源有足够的支持（kafka之类的消息队列）

编辑于 2021-07-31 17:24