Victoria Metrics 索引写入流程

关注

TSID 介绍

索引建立的时机

- vm通过 `TSID` 将 `metricName`, `tags` 与`timestamp`, `values` 相关联。在时序数据写入时, 需要查找时序数据的`metricNameRow` 的 `TSID` ,如果未找到则插入一个新的索引。
- 在合理使用时序数据库的场景下, 大多是时候写入的时序数据都是可以命中索引直接找到 TSID 的，建立索引的操作是比较低频的。

TSID的结构

```
// storage/tsid.go
type TSID struct {
        // metricName, 指标名对应的id
        MetricGroupID uint64

        JobID uint32
        InstanceID uint32

  // metricNameRaw,  指标名+tags对应的id
        MetricID uint64
}
复制代码
```

一个`TSID`的结构有四个字段,分别是`MetricGroupID`,`JobId`,`InstanceId`,`MetricID`。其中`JobId`和`InstanceId`是为了兼容prometheus的协议而添加的，这里我们不讨论。

`MetricGroupId`的生成方法是对指标名做hash,方法如下:

```
// storage/index_db.go:generateTSID
dst.MetricGroupID = xxhash.Sum64(mn.MetricGroup)
复制代码
```

`MetricID`通过`generateUniqueMetricID()`生成,在重启时,`nextUniqueMetricID`被赋值为当时的时间戳,随后每次新的`TSID`的创建都会在此基础之上+1。

```
// storage/index_db.go:generateTSID
dst.MetricID = generateUniqueMetricID()
```

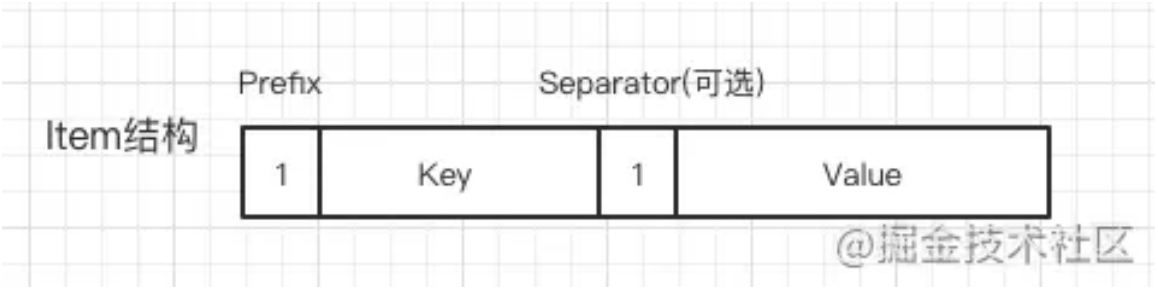
```
// storage/index_db.go
var nextUniqueMetricID = uint64(time.Now().UnixNano())

// storage/index_db.go:generateUniqueMetricID
func generateUniqueMetricID() uint64 {
        return atomic.AddUint64(&nextUniqueMetricID, 1)
}
复制代码
```

倒排索引

倒排索引的结构

在创建完 TSID 后,需要建立一系列的索引供查找时使用。在vm中,不同类型的索引都是通过 KV 关系来描述，在代码中称为Item,Item 的结构如下:



其中 Prefix 一个字节表示索引的类型;Separator在无法区分 Key, Value 时用于分割KV；在可以区分KV的情况下(如key定长以及key中本身含有分隔符),不使用该字段。

具体的索引类型如下:

```
// storage/index_db.go
const (
        // Prefix for MetricName->TSID entries.
        nsPrefixMetricNameToTSID = 0
```

```
        // Prefix for Tag->MetricID entries.
        nsPrefixTagToMetricIDs = 1

        // Prefix for MetricID->TSID entries.
        nsPrefixMetricIDToTSID = 2

        // Prefix for MetricID->MetricName entries.
        nsPrefixMetricIDToMetricName = 3

        // Prefix for deleted MetricID entries.
        nsPrefixDeletedMetricID = 4

        // Prefix for Date->MetricID entries.
        nsPrefixDateToMetricID = 5

        // Prefix for (Date,Tag)->MetricID entries.
        nsPrefixDateTagToMetricIDs = 6
)
复制代码
```

在 `storage/index_db.go:createIndexes` 函数中,分别建立各个索引,生成 `items`

> 对于 **INSERT cpu,host=serverA,region=us_west value=0.64** 的时序指标，假设生成的 `TSID` 为
>
> ```
> {
> "MetricGroupID": 10000,
> "MetricID": 20000
> }
> 复制代码
> ```
>
> 则生成的`items`的逻辑结构分别为:

```
// nsPrefixMetricNameToTSID
00 | cpu,host=serverA,region=us_west -> 10000,20000
// nsPrefixMetricIDToMetricName
03 | 20000 -> cpu,host=serverA,region=us_west
// nsPrefixMetricIDToTSID
02 | 20000 -> 10,20000
// nsPrefixTagToMetricIDs
01 | 01 |  cpu -> 20000
01 | host | 01 | serverA -> 20000
01 | region | 01 | us_west -> 20000
复制代码
```

即一个指标会生成 `n+4`，n为指标 `Tag` 的数量

Item InmemoryBlock的merge流程

在vm中维护着一个 `table` 结构，其中和写入索引相关性比较大的两个字段为 `rawItemsBlocks` 和 `parts`，`rawItemsBlocks` 为预处理 `Items` 使用，把 Items 划分成 64K 对齐的 block。parts主要是存储merge后的blocks，一个part与文件系统上的一个目录对应。

```
// mergeset/table.go
type Table struct {
  // ...
        partsLock sync.Mutex
        parts     []*partWrapper

        rawItemsBlocks        []*inmemoryBlock
        rawItemsLock          sync.Mutex
  // ...
}

// mergeset/encoding.go
```

```
type inmemoryBlock struct {
        commonPrefix []byte
        data         []byte
        items        byteSliceSorter
}
复制代码
```

mergeBlocks 的时机有四个:

1. 当merge产生的block数量超过1024，进行merge操作
2. 每秒由定时器执行刷新一次
3. createSnapshot时
4. 程序退出时

inmemoryPart 结构

inmemoryPart 是 part 读入内存中的结构, 在 inmemoryBlock merge之前，每个inmemoryBlock都会转换成一个 inmemoryPart 的结构(block数量为1)，inmemoryPart中 metaindexData, indexData, itemsData, lensData 数据结构与磁盘对应的文件内容一致。

具体的结构如下:

```
type inmemoryPart struct {
  // partHeader 记录 itemsCount, blocksCount, firstItem, lastItem信息, 最后会序列化到metadata.json
        ph partHeader
  // 压缩后的items和lens数据
        sb storageBlock
  // 当前block的header信息，有commonPrefix, firstItem, marshalType, itemsCount, itemsBlockOffset, lenBlockOffset, itemsBlockSize, lenBlockSize
        bh blockHeader
  // 当前block的metaindex信息,存储了当前blockHeader的
```

```
firstItem, blockHeaderCount, indexBlockOffset,
indexBlockSize
        mr metaindexRow

  // 未压缩的index字节数组
        unpackedIndexBlockBuf []byte
  // 压缩后的index字节数组
        packedIndexBlockBuf   []byte

  // 未压缩的metaindex字节数组
        unpackedMetaindexBuf []byte
  // 压缩后的metaindex字节数组
        packedMetaindexBuf   []byte

  // 用于序列化最后写入内存/磁盘文件使用
        metaindexData bytesutil.ByteBuffer  // ->
metaindex.bin
        indexData     bytesutil.ByteBuffer  // -> index.bin
        itemsData     bytesutil.ByteBuffer  // -> items.bin
        lensData      bytesutil.ByteBuffer  // -> lens.bin
}
复制代码
```

merge inmemoryBlock调用的函数为 `mergeRawItemsBlocks`, 流程如下

1. 每次选取最多 `defaultPartsToMerge:15`数量的blocks, 调用 `tb.mergeInmemoryBlocks(blocksToMerge[:n])`
2. 将所有的 `inmemoryBlock` 转换为 `inmemoryPart` 结构, 为每个 `inmemoryPart` 构造 `blockStreamReader`, 用于迭代读取items
3. 创建一个新的`inmemoryPart`,并构造一个`blockSteamWriter` 用于合并写入的数据

4. 调用 `mergeBlockStreams(&mpDst.ph, bsw, bsrs, tb.prepareBlock, nil, nil, &tb.itemsMerged)` 开始merge数据

5. 把多个 `blockStreamReader` 构造成 `blockStreamMerger` 结构，merger里面主要是一个堆用于维护`bsrs`,用于merge数据时的排序

   > 这里主要解决的问题是多个有序的字节(item)数组(inmemoryPart)，按照字节序排序，合成一个inmemoryPart的过程。
   >
   > 主要的实现代码在`mergeset/merge.go:Merge`
   >
   > 在merge的过程中，每64KB会单独的建立一个blockHeader，用于快速索引该block里面的items

6. 在重复**1-5**步骤后，**n**个`inmemoryBlock`会合并成 *(n-1)/defaultPartsToMerge+1* 个 `inmemoryPart`, 最后再调用 `mergeParts` 完成索引持久化。

分类：