**LevelDB Bloom Filter实现**

# 1.   RFC

如下内容是Sanjay发表在Google Groups [leveldb](https://groups.google.com/forum/?fromgroups#!forum/leveldb)上的初始设计方案。实际实现可能与此不同。对于bloom filter的支持是在最新的1.4版本中加入的，在此之前的版本中并无此支持。

人们希望可以在LevelDB中加入bloom filter的支持。目前针对一次查询，LevelDB可能需要在每个level上进行一次磁盘随机访问。通过使用bloom filter可以大大减少所需要的随机访问操作次数。比如，假设调用者正在查找一个值为”Foo”的key，LevelDB会从每个level下选择相应的SSTable文件(那些range包含了该key的文件)，之后会在这些SSTable文件上进行随机读。如果每个SSTable都有一个对应的bloom filter，那么查找时就可以很容易地通过检查bloom filter跳过那些不包含该key的SSTable文件。

下面的内容会描述下如何为LevelDB添加bloom filter支持。事实上，我们提供了更通用的bloom filter支持，允许应用进行定制，为不同类型的查询减少磁盘随机访问，并不仅限于单个key的查询。

为了将为一系列key构建summary以及根据summary判断某个key是否存在的机制进行封装，我们会增加一个新的接口类型。

class Summary {  
   public:  
    // Return a summary of the contents of key\_set  
    virtual std::string Construct(key\_set /\* exact type TBD \*/);

    // Returns true if key may potentially match one of the keys that  
    // generated summary.  
    virtual bool MatchKey(const Slice& summary, Slice key);  
  };

应用程序可以在打开数据库时提供Summary的一个实例，LevelDB会使用该对象来减少随机读取。

LevelDB的发布版中将包含一个默认的基于bloom filter的Summary实现，很多应用程序只需要使用这个实现就可以了： Construct()方法会返回key\_set中的所有key的bloom filter。如果key与存储在summary中bloom filter匹配，MatchKey()方法就会返回true。

 在内部，LevelDB实现将会为每个SSTable data block调用Construct()，同时将结果存储到指向给data block的index block pointer附近。在读取一个data block之前，LevelDB将会读取对应的summary，同时如果MatchKey()返回false，该data block的读取就会被跳过。

 应用程序实例：

    // Open the database  
    leveldb::Options options;  
    options.summarizer = leveldb::BloomFilterSummary();  
    leveldb::DB::Open(options, ...);

    // The following lookup will use bloom filters to reduce disk reads  
    leveldb::Get(..., key, &value);

上述变更会加速Get()调用的执行。但是我们也希望在使用iterator查找某些pattern时也可以利用上summary。为了做到这一点，需要允许ReadOptions包含一个”query”参数同时为Summary增加一个接口MatchQuery()，该函数会以通过ReadOptions传递的query为参数。

 比如，考虑一个使用LevelDB的特殊应用程序的常用操作：搜索那些大于某个长度的key。应用程序可以声明一个以key\_set中的key的最大长度为summary的Summary类。query参数将会包含查询操作正在寻找的key的最小长度M。如果编码在query中的长度M大于编码在summary中的长度，MatchQuery()会返回fasle。

   class LengthSummary : public leveldb::Summary {  
    public:  
     virtual std::string Construct(key\_set) {  
       int mlength = 0;       // Use length of longest key as summary  
       for each key in key\_set {  
         mlength = max(mlength, key.size());  
       }  
       return Encode(mlength);  
     }  
     virtual bool MatchQuery(const Slice& summary, const Slice& query) {  
       // Return true iff the length encoded in \*query is <= the length  
       // encoded in the summary.  
       return Decode(query) <= Decode(summary);  
     }  
   };

  // Open the database with a custom summarizer  
   LengthSummary summary;  
   leveldb::Options options;  
   options.summarizer = &summary;  
   leveldb::DB::Open(options, ...);

   // Find entries with keys >= 10000 bytes in length  
   leveldb::ReadOptions read;  
   read.query = Encode(10000);  
   leveldb::Iterator\* iter = db->NewIterator(read);  
   for (iter->SeekToFirst(); iter->Valid(); iter->Next()) {  
     ...  
   }

定制化的summary函数可能比较有用的另一个场景是在那些将多个feature编码成单个key，但是只想基于key的一个前缀进行查找的应用程序中。这样的应用程序可以使用一个在生成/查询bloom filter时只依赖于key的前缀的Summary实现即可。

# 2.   实现

由于LevelDB在磁盘上的数据组织方式，单个的Get()调用可能引入对磁盘的多个读操作。FilterPolicy机制可以用来减少磁盘操作。

 调用方式：

   leveldb::Options options;

   options.filter\_policy = NewBloomFilter(10);

   leveldb::DB\* db;

   leveldb::DB::Open(options, "/tmp/testdb", &db);

   ... use the database ...

   delete db;

   delete options.filter\_policy;

上面的代码为将一个基于Bloom Filter的过滤策略关联到数据库。基于Bloom Filter的过滤依赖于内存中针对key的一定bit的数据(在上述的例子中每个key对应10bit的数据)。这个filter大概可以将Get()所需要的磁盘读取降低100倍。随着每个key所对应的数据bit的增加会带来更大的降低，与此同时内存使用也会随之上升。我们推荐那些工作集合无法完全放入内存或者涉及很多随机读操作的应用设置一个过滤策略。

 如果你使用了一个自己定制的比较器，需要保证你正在使用的过滤策略与该比较器是兼容的。比如，对于一个在比较key时会忽略末尾空格的比较器，NewBloomFilter就不能与这样的比较器配合使用。相反地是应用程序应该提供一个同样忽略末尾空格的过滤策略。比如：

  class CustomFilterPolicy : public leveldb::FilterPolicy {

   private:

    FilterPolicy\* builtin\_policy\_;

   public:

    CustomFilterPolicy() : builtin\_policy\_(NewBloomFilter(10)) { }

    ~CustomFilterPolicy() { delete builtin\_policy\_; }

    const char\* Name() const { return "IgnoreTrailingSpacesFilter"; }

    void CreateFilter(const Slice\* keys, int n, std::string\* dst) const {

      // Use builtin bloom filter code after removing trailing spaces

      std::vector<Slice> trimmed(n);

      for (int i = 0; i < n; i++) {

        trimmed[i] = RemoveTrailingSpaces(keys[i]);

      }

      return builtin\_policy\_->CreateFilter(&trimmed[i], n, dst);//{?bug?why not trimmed[0]?}

    }

    bool KeyMayMatch(const Slice& key, const Slice& filter) const {

      // Use builtin bloom filter code after removing trailing spaces

      return builtin\_policy\_->KeyMayMatch(RemoveTrailingSpaces(key), filter);

    }

  };

 在更高级的应用中，可能会提供一个不是使用bloom filter对key求summary的过滤机制。具体可参考：leveldb/filter\_policy.h。

## 2.1.  文件格式

为了支持Filter机制，LevelDB为SSTable增加了一个新的Meta Block类型：”filter” Meta Block。如果数据库在打开时，指定了FilterPolicy，metaindex block内将会包含一条记录用来将"filter.<N>"映射到针对该filter meta block的block handle。此处”<N>”代表了由filter policy的Name()方法返回的字符串{!也就是说在metaindex block内将会有一个KeyValue对用来寻址filter summary数据组成的meta block，对于该KeyValue对来说，Key就是"filter.<N>"，value就是filter meta block的handle}。

 Filter Block存储了一系列的filters。对于这些filters来说，filter i包含了以offset落在[ i\*base ... (i+1)\*base-1 ]的那个block的所有key为输入的FilterPolicy::CreateFilter()函数的输出结果{!因此实际中一个filter可能对应不止一个block，但是肯定是整数个block，一个block不会跨越两个filter，这样可以快速地计算出一个block对应的filter，因为data index block中存储了data block的offset，直接根据offset和base取值就可以计算出该data block对应了第几个filter。只要知道了是第几个filter，根据Filter Block自身的结构，就可以直接访问该filter数据了}。

 目前，base的值是2KB。因此， 比如blocks X和Y的起始位置都是位于[0KB,2KB-1]，X和Y中的所有key将会通过调用FilterPolicy::CreateFilter()转变为一个filter，同时最终计算出的filter将会作为filter block的第一个filter进行存储。Filter Block的格式如下：

     [filter 0]

     [filter 1]

     [filter 2]

     ...

     [filter N-1]

     [offset of filter 0]                  : 4 bytes

     [offset of filter 1]                  : 4 bytes

     [offset of filter 2]                  : 4 bytes

     ...

     [offset of filter N-1]                : 4 bytes

     [offset of beginning of offset array] : 4 bytes

     lg(base)                              : 1 byte

 通过Filter Block底部的offset数组可以很容易地将一个data block的offset映射到它所对应的filter{!根据前面的解释，给定data block我们很容易得出它对应的是第几个filter。知道了第几个filter，再根据此处的offset数组很容易就计算出filter的offset了，知道了offset再知道size，就可以取出该filter对应的数据了，而size又可以根据下一个filter的offset-当前offset计算得出。这里的offset是指在filter block内的偏移，而不是文件内的。另注意lg(base)存储的是base对2取对数的值，也就是说比如base值为2KB，那么存储的就是lg(2\*1024)=11}。

## 2.2.  接口设计

目前的实现与RFC中的设计有一些不同，比如增加了一个Name()接口，该接口用来返回该FilterPolicy的名称，该名称一方面会用于metaindex block的KeyValue对中的Key的命名中："filter.<N>"。另一方面还可以用来校验写入时与读取时采用的FilterPolicy是否一致，类似于Comparator中的Name()函数。如果filter计算方式发生了不兼容地变更时，必须修改该函数，避免误用。

 另外CreateFilter()与Construct()相比，接口也有所改变。CreateFilter()的接口定义可以直接在原字符串上操作，减少内存拷贝次数。此外目前还未提供MatchQuery的支持。

|  |
| --- |
| class FilterPolicy { |
| public: |
| virtual ~FilterPolicy(); |
| virtual const char\* Name() const = 0; |
| virtual void CreateFilter(const Slice\* keys, int n, std::string\* dst) |
| const = 0; |
| virtual bool KeyMayMatch(const Slice& key, const Slice& filter) const = 0; |
| }; |
|  |
|  |
| extern const FilterPolicy\* NewBloomFilterPolicy(int bits\_per\_key); |

 用户可以通过继承实现一个FilterPolicy定义自己的过滤机制。

调用方式：

   leveldb::Options options;

   options.filter\_policy = NewBloomFilter(10);

   leveldb::DB\* db;

   leveldb::DB::Open(options, "/tmp/testdb", &db);

   ... use the database ...

   delete db;

   delete options.filter\_policy;

## 2.3.  代码分析

如果要让filter机制生效，SSTable层和DB层都需要进行一些改动。

### 2.3.1.   默认的bloom filter实现

默认的bloom filter实现代码在util/bloom.cc中。

在该实现中，Name()返回的是"leveldb.BuiltinBloomFilter"，因此metaindex block 中的key就是”filter.leveldb.BuiltinBloomFilter” 。这里的hash函数是通过使用两个hash函数来生成一系列的hash函数[理论分析可参考论文[Kirsch,Mitzenmacher 2006](http://www.eecs.harvard.edu/~kirsch/pubs/bbbf/esa06.pdf)]。此处的hash函数满足如下关系：

Gi(x)=H1(x)+iH2(x)

H2(x)=(H1(x)>>17) | (H1(x)<<15)

其中H1和H2是基本的double-hashing，Gi是实际投入使用的hash函数。{?但是这里有个问题H2和H1不是独立的hash函数，H2是通过H1循环右移得到的}

在bloom\_filter的数据的最后一个字节存放的是k\_的值，k\_实际上就是G(x)的个数，也就是计算时采用的hash函数个数。

### 2.3.2.   SSTable层的修改

table目录下的主要改动有：

  新增了filter\_block.h，filter\_block.cc，内有FilterBlockReader,FilterBlockBuilder用于filter block的读写。FilterBlockBuilder有三个接口StartBlock,AddKey,Finish ,这三个接口调用必须按照如下形式(StartBlock AddKey\*)\* Finish。StartBlock函数会判断是否需要将在此block之前的那些block的key组成的集合求一次filter，如果之前的block所对应的filter与当前block对应的不是同一个，则GenerateFilter()。AddKey会把key连续存放到一个string中，在GenerateFilter()时会根据keys\_和start\_构建出一个key的list，然后计算它们对应的summary。最终result\_里保存的就是最终的filter block数据。FilterBlockReader功能主要在于根据现有的filter block数据，判断给定的key及其所在block的offset，判断key是否可能命中。

  Table类中新增InternalGet,ReadMeta,ReadFilter函数，同时新增filter, filter\_data两个成员变量。在Open()时，会调用ReadMeta函数，该函数会读取metaindex block，并根据” filter.<N>”找到filter meta block的handle，然后调用ReadFilter函数将filter meta block数据读出，并设置filter和filter\_data。InternalGet函数，会首先找到key所对应的block handle，这样就得到了该block对应的offset，然后就可以通过filter->KeyMayMatch判断是否需要继续读取，如果不需要就可以直接返回，避免了对data block的读取。

  TableBuilder类中增加filter\_block成员变量，在TableBuilder::Add()调用的同时会通过调用AddKey函数更新filter数据，在TableBuilder::Flush()调用时，调用StartBlock函数，在TableBuilder::Finish()调用时，会首先写入filter block数据，此时会调用Finish函数，然后写入metaindex block，然后是index block，最后是footer。

 另外可以观察到的一点，过滤机制对于实际的数据读取实际上只是锦上添花，并不是不可或缺的，因此对于很多出错情况进行了容错。

### 2.3.3.   DB层的修改

首先Option里要包含filter\_policy这一参数。然后在Option提供的filter\_policy对象的基础上，内部实现了一个InternalFilterPolicy，这样做的原因是需要从internal key中求出user key，然后再针对user key进行filter相关运算{!理解此处需要知道internal key与user key的区别，用户给定的key在LevelDB内部存储时，实际上会被转换为internal key，internal key由三部分组成：user\_key,sequence,type。}。从DB层来看，实际使用的就是InternalFilterPolicy。

 TableCache增加了Get接口，在该函数中会调用Table的InternalGet。而TableCache的Get接口又会被Version::Get调用。Version::Get又会被DBImpl::Get调用。

 DB在查找某个key时，首先会查找内存中的memtable，如果找不到才会去文件中查找，通过Version::Get遍历所有level下的文件，对于level 0来说可能需要读取多个文件，因此在读取前需要对某level下文件根据创建时间进行排序，优先读取最新创建的文件，在读取某文件时会通过table\_cache\_->Get读取给定key。TableCache::Get会调用FindTable在cache中查找是否存在对应table的缓存，如果不存在就重新调用Table::Open，然后将其存入cache。之后调用FindTable所得到的table对象的InternalGet，得到对应key的value，并将其保存到Saver结构中，Saver结构定义在/db/version\_set.cc中。Table::InternalGet会首先利用filter进行判断该key是否match，如果不match就直接返回，无需读取相应的block，否则才需要进一步读取并判断是否是所查找的key。filter就是通过这样的过程在DB的查找中起到作用的。另TableCache::Get,Table::InternalGet的输入参数k都是internal key，执行filter相关操作时，它会被转换为user\_key。

# 3.   参考文献

<https://groups.google.com/forum/?fromgroups#!topic/leveldb/qGDONxfBAf4>

<http://leveldb.googlecode.com/svn/trunk/doc/index.html>

<http://leveldb.googlecode.com/svn/trunk/doc/table_format.txt>

<https://code.google.com/p/leveldb/source/detail?r=85584d497e7b354853b72f450683d59fcf6b9c5c>