

数据转储

Github

1. 问题重述

2. 数据转储

2.1. 数据预处理

2.2. 转储结构

如何确定行键？

如何确定列族/列修饰符

冗余副本duplicate_data

为什么要选择这样的结构

2.3. 转储实现

数据分块

蚂蚁搬大象：向Hbase转储数据分片

2.4. 掉电与宕机容错

3. 数据检索引擎与缓存机制

3.1. Hbase连接、存取数据以及关闭断开连接

连接数据库

获取数据

关闭连接

3.2. 缓存机制

Default:LRU Cache

用户自定义的Cache

4. Summary

Github

branch:lab1

<https://github.com/Liyanhao1209/BDL.git>

1. 问题重述

利用滑动窗口算法进行时间序列分析与流量预测。将每一个滑动窗口的数学表示与后续一个时间点上的label存入Hbase中。

其中，每一个滑动窗口的数学表示是key，而后续一个时间点上的label是value。

2. 数据转储

接下来我们进行数据集的转储，转存到Hbase中。

2.1. 数据预处理

2.2. 转储结构

在Hbase中，我们采用列式存储，每一行都有列族（也就相当于关系型数据库中的主索引），因此我们需要选择一个合适的数据结构来作为KV对的具体形式。

也就是说，必须要明确以下两个问题：

1. 怎样构建每一行的key？
2. 怎样构建每一行的value？

如何确定行键？

换句话说，data中的每一个流量数据，如何唯一地标识？

回顾一下data[i][j][k]的含义：第j个城区的第k个业务在第i个10分钟内的流量数据。也就是说有3个维度标识了每一个y值。而这三个维度中的最大值，不超过9999。

而最终的行键其实是每一个滑动窗口的数学表示。这里我们设滑动窗口的长度为w（假设是一个定长的sliding window），那么对于一个长度为l的时间序列seq，就会产生l-w+1个滑动窗口（前提是w≤l），而由于value是下一个时间点，如果滑窗右侧没有元素，那么就相当于没有value了，因此要舍弃最后一个滑窗。

因此，最终将产生(l-w)个滑窗，而此处l=i，那么将产生(i-w)个滑窗。因为我们总计有10000个地区和5个业务，因此最后将产生(8928-w)*10000*5个KV对。

由是，我设计每一行的形式如下：

行键	i(0≤i≤8927-w)	j(0≤j≤9999)	k(0≤k≤4)
value	data[i+w][j][k]		

不过在Hbase中，行键是一个字节串，例如b'12138'这样的，那么我们就需要使用一些状态压缩的算法，把i,j,k给压缩成一个字节串。

状态压缩

i,j,k的数据范围都非常小，最大的不过9999，也小于 2^{16} 次方，所以我们可以把i,j,k压缩为：

```
(i<<16<<16) | (j<<16) | k
```

这样就把三个整数压缩成一个整数了，然后我们把这个整数转为字节串作为行键就可以了

如何确定列族/列修饰符

这个比较简单了，仅保存窗口后一个点的y值即可。形如col:value=data[i+w][j][k]

那么怎么还原出滑窗内这w个采样点呢？也就是data[i:i+w-1][j][k]的数据？

冗余副本duplicate_data

我选择在Hbase中另起一张表duplicate_data，专门用来存放原始数据集。可以把上述KV对看作是原始数据集的某种意义上的metaData。这样我们在访问由j,k确定的第i个滑窗时，可以直接向duplicate_data中检索w次，其中键为：

行键	i+δ	j	k
----	-----	---	---

其中 $\delta \in [0, w-1]$

随后利用同样的状态压缩算法计算行键，插入到Hbase。这样我们只要查询对应行键就还原出了窗口内的采样点。

为什么要选择这样的结构

无论是什么样的数据库，对于某一条数据，一定要有一个对应的且全局唯一的索引。不然数据库也没法知道用户到底想访问哪条数据。

因此我无法把窗口内的所有采样点的值作为行键，原因如下：

1. 当窗口较大，用户查询时，需要传入相当批量的值组成行键，操作不便。
2. 使用数据点的值组成的行键，会导致重复。比如四个数据点为：

```
[ 1, 1, 1, 1 ]
```

再假设窗口大小为3。那么当用户输入[1,1,1]作为行键时，他想查询的是前面那个窗口，还是后面那个？

因此我选择使用窗口的唯一标识来作为行键，就不会导致行键的重复。但这样也就导致了用户无法访问到具体的每一个数据点，因此我选择创建一个数据集的副本，将其作为查询原始数据的数据库。

2.3. 转储实现

数据分块

我在内存8G,外存128G的Ubuntu22.04机器上试了一下用pycharm执行第一个版本的脚本，进程直接就被杀死了，因为光原始数据就是3个多G，远远超出了一个应用程序可以使用的最大内存空间上限。

因此我的思路就转向了将数据进行**分块**，然后**分片处理**的方向。大致思路是：

1. 把一个数据集拆分成多个小的数据集。
2. python脚本将每个小的数据集的原始数据以及滑窗数据转储到Hbase。
3. 多次执行python脚本，把每个小数据集转储到Hbase。
4. 最终Hbase中就拥有了整个数据集的原始数据以及滑窗数据。

而对数据进行分块的任务，我也是通过脚本来完成的，大致思路如下：

1. 首先在我自己的Windows的机器上读入整个数据集（我的电脑配置高一点，能处理这样的大型数据集）
2. 鉴于整体data的大小为8928*10000*5，我选择将data拆分为893个新的数据集，每一个的大小为10*10000*5，减少Linux机器的负担。
3. 其中前8个文件大小为10*10000*5，最后一个大小为8*10000*5
4. 对于每个新的.h5文件，我采用这样的命名方式：
 - a. 假设这个文件存储的是原数据集中data[start:end,0:9999,0:4]的数据
 - b. 那么文件名为:

```
f"{start}<<16|end}"
```

这样我们就可以通过文件名，还原出这个文件在原数据集中的存储范围了。

还原方式如下(和16位全1串的掩码相与)：

```
# 假设文件名的前缀对应的数字为index
start = (index>>16)&0xFFFF
end = index & 0xFFFF
```

5. 最后，我们把这个文件名保存到一个txt文件中，这样在我们执行自己编写的shell脚本时，就可以根据这个txt文件中的文件名，来找到对应的.h5文件，然后加载子数据集。

代码如下：

```
import os
import sys

import h5py

def split_data_and_save(hdf5_path, target_dir, b_size):
    # 打开HDF5文件
    with h5py.File(hdf5_path, 'r') as f:
        # 读取data和idx数据集
        data = f['data'][:]
        idx = f['idx'][:]

        # 初始化分割参数
        block_size = b_size
        num_full_blocks = data.shape[0] // block_size
        remainder = data.shape[0] % block_size

        # 遍历并分割data和idx
        for i in range(num_full_blocks):
            start_idx = i * block_size
            end_idx = start_idx + block_size
            print("start end", start_idx, end_idx)
```

```

# 分割data和idx
data_block = data[start_idx:end_idx]
idx_block = idx[start_idx:end_idx]

# 计算文件名
begin = start_idx
end = end_idx - 1
filename = f"{begin << 16 | end}.h5"

# 创建输出文件的路径
output_file = os.path.join(target_dir, filename)

# 将分割后的数据写入新的HDF5文件
with h5py.File(output_file, 'w') as outfile:
    outfile.create_dataset('data', data=data_block)
    outfile.create_dataset('idx', data=idx_block)

    # 处理剩余部分
if remainder > 0:
    start_idx = num_full_blocks * block_size
    end_idx = start_idx + remainder

    # 分割data和idx
    data_remainder = data[start_idx:end_idx]
    idx_remainder = idx[start_idx:end_idx]

    # 计算文件名
    begin = start_idx
    end = end_idx - 1
    filename_remainder = f"{begin << 16 | end}.h5"

    # 创建输出文件的路径
    output_file_remainder = os.path.join(target_dir, filename_remainder)

    # 将剩余数据写入新的HDF5文件
    with h5py.File(output_file_remainder, 'w') as outfile_re

```

```

        outfile_remainder.create_dataset('data', data=data_remainder)
        outfile_remainder.create_dataset('idx', data=idx_remainder)

def main(hdf5_path, target_dir, b_size):
    # 确保目标目录存在
    if not os.path.exists(target_dir):
        os.makedirs(target_dir)

    # 调用函数拆分数据并保存
    split_data_and_save(hdf5_path, target_dir, b_size)

    # 创建文件名列表并写入txt文件
    filenames = os.listdir(target_dir)
    filenames.sort(key=lambda x: int(x.split('.')[0])) # 按文件编号排序
    with open(os.path.join(target_dir, 'list.txt'), 'w') as txtfile:
        for filename in filenames:
            txtfile.write(filename + '\n')

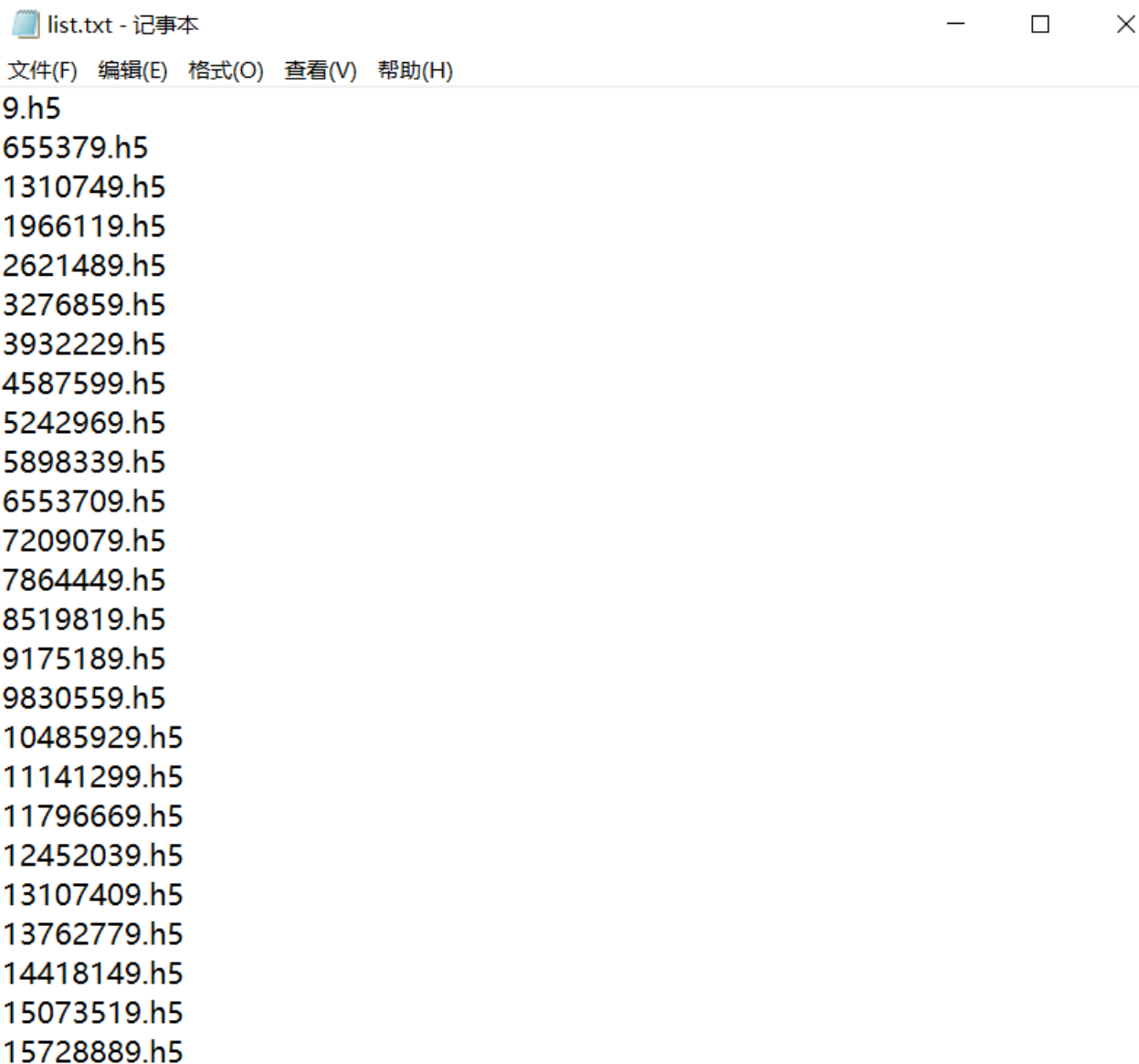
if __name__ == "__main__":
    args = sys.argv
    # HDF5文件的路径
    hdf5_path = args[1]
    # 目标保存目录
    target_dir = args[2]
    chunk_size = int(args[3])
    main(hdf5_path, target_dir, chunk_size)

```

最后执行完结果如下：

此电脑 > 桌面 > college > junior2 > 工业大数据 > 实验 > lab1 > data > split_10				
	名称 ^	修改日期	类型	大小
✦	9.h5	2024/4/10 10:30	H5 文件	3,909 KB
✦	655379.h5	2024/4/10 10:30	H5 文件	3,909 KB
✦	1310749.h5	2024/4/10 10:30	H5 文件	3,909 KB
✦	1966119.h5	2024/4/10 10:30	H5 文件	3,909 KB
	2621489.h5	2024/4/10 10:30	H5 文件	3,909 KB
	3276859.h5	2024/4/10 10:30	H5 文件	3,909 KB
	3932229.h5	2024/4/10 10:30	H5 文件	3,909 KB
	4587599.h5	2024/4/10 10:30	H5 文件	3,909 KB
	5242969.h5	2024/4/10 10:30	H5 文件	3,909 KB
	5898339.h5	2024/4/10 10:30	H5 文件	3,909 KB
	6553709.h5	2024/4/10 10:30	H5 文件	3,909 KB
ona	7209079.h5	2024/4/10 10:30	H5 文件	3,909 KB
	7864449.h5	2024/4/10 10:30	H5 文件	3,909 KB
	8519819.h5	2024/4/10 10:30	H5 文件	3,909 KB
	9175189.h5	2024/4/10 10:30	H5 文件	3,909 KB
	9830559.h5	2024/4/10 10:30	H5 文件	3,909 KB
	10485929.h5	2024/4/10 10:30	H5 文件	3,909 KB
	11141299.h5	2024/4/10 10:30	H5 文件	3,909 KB
	11796669.h5	2024/4/10 10:30	H5 文件	3,909 KB
	12452039.h5	2024/4/10 10:30	H5 文件	3,909 KB
	13107409.h5	2024/4/10 10:30	H5 文件	3,909 KB
	13762779.h5	2024/4/10 10:30	H5 文件	3,909 KB
	14418149.h5	2024/4/10 10:30	H5 文件	3,909 KB
	15073519.h5	2024/4/10 10:30	H5 文件	3,909 KB
	15728889.h5	2024/4/10 10:30	H5 文件	3,909 KB
	16384259.h5	2024/4/10 10:30	H5 文件	3,909 KB
	17039629.h5	2024/4/10 10:30	H5 文件	3,909 KB
	17694999.h5	2024/4/10 10:30	H5 文件	3,909 KB
	18350369.h5	2024/4/10 10:30	H5 文件	3,909 KB

list.txt中的内容为：



这样我们就完成了数据分片。

蚂蚁搬大象：向Hbase转储数据分片

接下来我们就可以向Hbase中转储分切好的数据片了。记住我们的任务：

1. 转储一份原始数据
2. 转储一份滑动窗口

那么每次转储的数据属于哪个子数据集，就依赖于list.txt来决定。我们每次从list.txt中取出一行数据（也即一个文件名），然后删除这一行（这样下次取出的就是下一个文件名），并加载此次的子数据集。

```
def getSplit(path):
    path = path + "/list.txt"
    with open(path, 'r', encoding='utf-8') as file:
        lines = file.readlines()

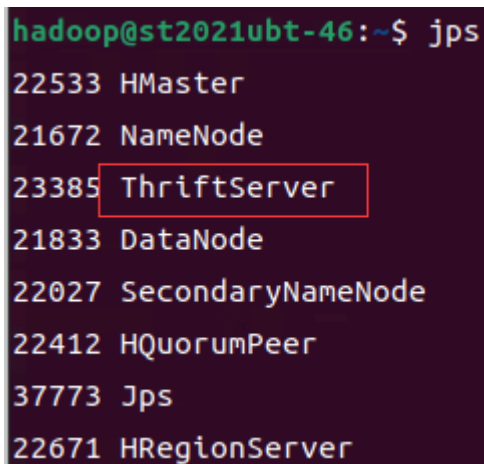
    length = len(lines)
    res = ''
    if length >= 1:
        res = lines[0].rstrip('\n')
        lines = lines[1:]

    with open(path, 'w', encoding='utf-8') as file:
        file.writelines(lines)

    return res
```

随后需要提一下的是，我使用了happybase这个简单的第三方库与本地的Hbase建立连接。而Hbase通过Thrift的Thrid Party组件来进行网络通信，因此我们必须启动一下thrift server：

```
hbase-daemon.sh start thrift
```



```
hadoop@st2021ubt-46:~$ jps
22533 HMaster
21672 NameNode
23385 ThriftServer
21833 DataNode
22027 SecondaryNameNode
22412 HQuorumPeer
37773 Jps
22671 HRegionServer
```

如果jps看到ThriftServer说明通信服务器起来了。

如果遇到hbase对命令无响应，可能是因为进入了NameNode的SafeMode，关掉就没事了：

```
hdfs dfsadmin -safemode leave
```

随后，进入第一步，转储原始数据，这个比较简单：

```
end = index & 0xFFFF
start = (index >> 16) & 0xFFFF
x, y, z = data.shape

for i in range(x):
    for j in range(y):
        for k in range(z):
            value = data[i, j, k]

            row_key = generate_row_key(i + start, j, k)
            duplicate_table.put(
                row_key,
                {'y_label:value': value})
```

这里注意行键的映射，第 q 个子任务($1 \leq q \leq 8$)的第一维的第 i 个数据，应该映射到整个数据集的 $(q-1)*1000+i$ 个第一维的数据上。而 j 和 k 没有经过分割，所以直接用最trivial的映射就行了。

之后，进入第二步，转储滑窗数据。

这里要注意一点，假设第一个任务中滑窗长度为6，存储了0-999这些数据点，但994-999这些数据点的监督值为1000-1005，但1000-1005不在当前这个数据集里。

所以下一个数据集要负责补上前一个数据集的滑窗的监督值的空缺，换句话说 $data[i,j,k]$ 要负责补上上一次的 $data[i+start-window,j,k]$ 到 $data[i+start-1,j,k]$ 的滑窗的监督值。

```
print('-----storing sliding windows to Hbase')
meta_table = pick_table(meta_name, connection)

meta_table.put('window_size', {"window_size": window})
if start > 0:
```

```

for i in range(window):
    for j in range(y):
        for k in range(z):
            value = data[i, j, k]

            row_key = generate_row_key(i + start - window, j, k)
            meta_table.put(row_key, {'y_label:value': value})

```

最后，加上本次数据集的滑窗即可：

```

for i in range(x - window):
    for j in range(y):
        for k in range(z):
            value = data[i + window, j, k]

            row_key = generate_row_key(i + start, j, k)
            meta_table.put(row_key, {'y_label:value': value})

```

整体代码见附件。

2.4. 掉电与宕机容错

这里的问题是，我是在自己的电脑上跑这个应用程序的。众所周知软件园校区工作日一到12点就要断电，掉电之后我的电脑不久就会因没电而关机，再次重启后就不知道应用程序执行到哪里了，或者说不知道处理到哪个数据分块了。

因此我们有必要建立一个简单的日志系统。这里我打算把中间产生的日志信息写到一个log文件中，不过我们的数据集实在太庞大了，因此这个log文件也将非常大，所以我并没有拼接日志，而是简单的覆写之前已经有的内容：

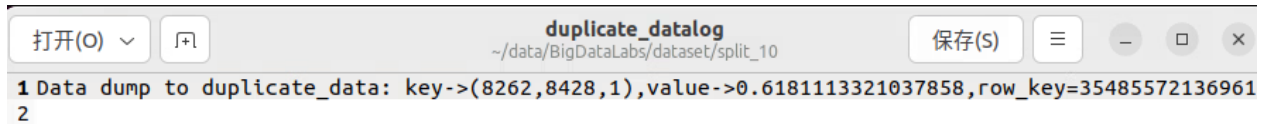
```

def dataDumpLog(log_path, log_info):
    if os.path.exists(log_path):
        with open(log_path, "w") as f:
            f.write(log_info)
    else:

```

```
with open(log_path, "w") as f:
    f.write(log_info)
```

这样记录的日志长成：



如果我们想还原现场，以这条日志为例，只需要计算 $((8262//10)<<32)|(8262//10+9)$ 即可得到data split的文件名，随后我们把这条文件名重新添加回list.txt的队首，重新跑转储任务。

这样我们最多冗余地存储 $10*10000*5=50w$ 条数据，但是换来了数据的完整性和一致性。

3. 数据检索引擎与缓存机制

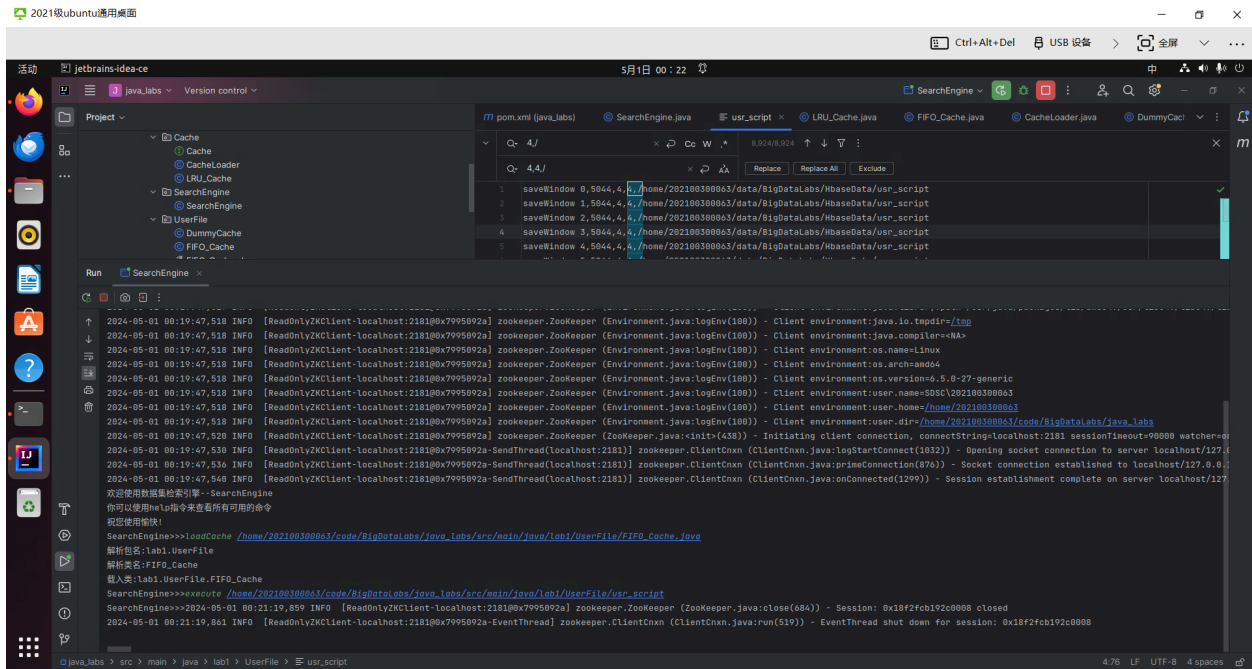
为了能够高效检索数据集，我引入了一个数据检索引擎，功能点大致如下：

```
String sb = "以下是你可以使用的其他命令\n" +
    "exit:\t退出程序\n" +
    "get \"i,j,k\":\t指定数据在data中的索引i,j,k,返回y值\n" +
    "getWindow \"i,j,k>window_size\":\t指定数据在data中的索引i,j,k,返回window_size\n" +
    "getRange \"si:ei,sj:ej,sk:ek\":\t指定数据在data中的范围,返回范围内的数据\n" +
    "save \"i,j,k,file_path\":\t将data[i,j,k]以拼接的方式存储到指定文件\n" +
    "saveWindow \"i,j,k>window_size,file_path\":\t将data[i:i>window_size]以拼接的方式存储到指定文件\n" +
    "saveRange \"si:ei,sj:ej,sk:ek,file_path\":\t将指定范围内的数据以拼接的方式存储到指定文件\n" +
    "enableCache:\t允许使用缓存(默认LRU)\n" +
    "loadCache file_path:\t从指定的文件中加载缓存插件\n";
```

大致有以下三类：

1. **系统相关**：获取指令集，退出系统，允许使用缓存，注入缓存等
2. **获取数据**：get系列，获取指定数据，输出到控制台
3. **保存数据**：save系列，获取指定数据，保存到指定文件中

4. 批量执行：execute系列，批量执行用户脚本中的命令



3.1. Hbase连接、存取数据以及关闭断开连接

这一部分比较简单，按照官方给的API做就可以

连接数据库

```
public static void init(String hbase_dir,String address){
    configuration = HBaseConfiguration.create();
    configuration.set(hbase_dir,address);
    try{
        connection = ConnectionFactory.createConnection(configuration);
        admin = connection.getAdmin();
    }catch (IOException e){
        System.out.println("数据库连接失败");
        System.exit(-1);
    }
}
```

获取数据

注意我们的数据集已经保存到数据库了，因此逻辑上这部分数据是只读的，因此只能getData，而不能putData。

```
public static double getData(String tableName,String rowKey,
    Table table = connection.getTable(TableName.valueOf(tableName));
    Get get = new Get(rowKey.getBytes());
    get.addColumn(colFamily.getBytes(),col.getBytes());
    Result result = table.get(get);
    byte[] value = result.getValue(colFamily.getBytes(), col.getBytes());
    table.close();

    return byteArrayToDouble(value);
}

public static double byteArrayToDouble(byte[] bytes) {
    return ByteBuffer.wrap(bytes).order(ByteOrder.BIG_ENDIAN).getDouble();
}
```

Hbase中存储的都是字节串，为了转换为想要的数据类型（double），还需要把byte[]数组转为double类型。

关闭连接

```
public static void close(){
    try{
        if(admin != null){
            admin.close();
        }
        if(null != connection){
            connection.close();
        }
    }catch (IOException e){
        System.out.println("释放与数据库的连接失败");
        System.exit(-1);
    }
}
```

```
    }  
}
```

3.2. 缓存机制

因为Hbase数据库访问数据时会读磁盘，效率远比读内存要慢。所以我选择基于内存设置一个缓存模块，提升查询效率。

对于一个缓存模块，他应该有以下功能：

```
package lab1.Cache;  
  
public interface Cache {  
    Double get(int i,int j,int k);  
    void put(int i,int j,int k,double value);  
    void setCapacity(int capacity);  
}
```

第一就是获取数据，鉴于我们的行键是由i,j,k三个索引通过位运算生成的，这里就以i,j,k作为获取数据的唯一标识。

第二个是在未命中的情况下替换数据/增加数据的。

第三个是设置缓存大小的。

Default:LRU Cache

先来看一个我自己实现的默认缓存。

```
public class LRU_Cache implements Cache{  
  
    private final List<Long> keys;  
    private int capacity;  
    private final HashMap<Long,Double> dict;  
  
    public LRU_Cache(int capacity){  
        dict = new HashMap<>();  
        keys = new ArrayList<>(capacity);  
    }  
}
```



```

        this.capacity = capacity;
    }
}

```

数据结构上，采用一个列表保存所有的key，当get时，更新该列表（更新某项数据的访问时间戳，以便更新LRU的数据）；同时还有一个Hash表，存储了key与value的映射。命中时，我们可以通过Hash表访问value。

对于get方法：

```

@Override
public Double get(int i, int j, int k) {
    long rk = gen_rk(i, j, k);
    if(hit(i,j,k)){
        keys.remove(rk);
        keys.add(0,rk);
    }

    return dict.get(rk);
}

```

若命中，移除key，并添加到列表首部。随后返回这个key对应的value。（没命中返回null）

对于put方法：

```

@Override
public void put(int i, int j, int k, double value) {
    assert !hit(i,j,k);

    long rk = gen_rk(i, j, k);
    if(keys.size()<capacity){
        keys.add(rk);
        dict.put(rk,value);
    }else{
        Long key = keys.remove(keys.size() - 1);
        dict.remove(key);
    }
}

```

```

        keys.add(0, key);
        dict.put(rk, value);
    }
}

```

我认为这个方法调用时，一定是没有命中的，如果命中了就没必要put，所以我加了个断言判断这条逻辑。

随后，如果Cache未满，则置入新的一条kv对，同时在记录时间戳的列表中加入key。否则，选择列表尾部的元素，然后把新的元素放到列表首部。

对于setCapacity方法：

```

@Override
public void setCapacity(int capacity) {
    this.capacity = capacity;
}

```

更新最大容量。

这样我们在get数据时，就可以利用这个Cache：

```

private static Double get(int i,int j,int k) throws IOException {
    long rk = gen_rk(i, j, k);
    if(enableCache){
        assert cache!=null;

        Double val = cache.get(i, j, k);
        if(val!=null){
            return val;
        }else{
            double data = getData(table_name, rk + "", cf, cq);
            cache.put(i,j,k,data);
            return data;
        }
    }else{
        return getData(table_name, rk + "", cf, cq);
    }
}

```

```
    }
}
```

首先我们先看下Cache机制是否开启了，如果开启了，查看能否在Cache中访问到这个数据，如果访问不到，我们直接从Hbase中获取，并put到Cache中。如果能查看到，那么直接返回即可。

如果没开启Cache，就直接从Hbase中查找。

用户自定义的Cache

为了注入用户自己编写的Cache，就需要在搜索引擎运行时热加载用户编写的组件，因此先用JavaCompiler编译用户的.java文件，随后通过正则表达式匹配用户代码的包名与类名，根据完全限定类名加载这个类，随后返回该类的实例注入到搜索引擎中，以此实现热加载。

因为用户的代码没有经过编译，所以一开始绝对是不在java的ClassPath里的，所以我们要动态地加载这个Cache。这就需要我们自己实现一个动态类加载器：

```
static class CustomClassLoader extends URLClassLoader {
    public CustomClassLoader(URL[] urls) {
        super(urls);
    }

    @Override
    public Class<?> findClass(String name) throws ClassNotFoundException {
        try {
            byte[] data = loadClassData(name);
            return defineClass(name, data, 0, data.length);
        } catch (IOException e) {
            throw new ClassNotFoundException("Class not found");
        }
    }

    private byte[] loadClassData(String name) throws IOException {
        try (InputStream in = getResourceAsStream(name.replace(".", "/"))) {
            if (in == null) {
                throw new FileNotFoundException();
            }
        }
    }
}
```

```

    }
    try (ByteArrayOutputStream out = new ByteArrayOutputStream();
         byte[] buffer = new byte[1024];
         int bytesRead;
         while ((bytesRead = in.read(buffer)) != -1)
             out.write(buffer, 0, bytesRead);
        )
        return out.toByteArray();
    }
}
}
}

```

首先我们假设这个.java文件已经通过某些手段编译成.class文件了。因此我们可以读取这个.class文件，然后从中解析出类的Class对象。以此实现热加载。

不过怎么编译这个.java文件呢？我选择了用javax的JavaCompiler组件：

```

private static void compileJavaFile(String javaFilePath) throws Exception {
    JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
    int compilationResult = compiler.run(null, null, null, javaFilePath);
    if (compilationResult != 0) {
        throw new RuntimeException("Error compiling cache Java file: " + javaFilePath);
    }
}

```

那么怎么找到这个.class文件呢，注意，我们把这个.class文件置于和.java文件同一目录下，因此可以直接到这个目录读取。

不过为了让java能准确地找到这个类，我们需要读取它的完全限定类名，也即包名+类名。而包名和类名都已经写在了.java文件的源代码中。因此我们可以读取这个.java文件，用正则表达式匹配包名和类名，然后拼接得到完全限定类名。

```

private static String getCacheClassName(String javaFilePath) throws Exception {
    try {
        // 读取文件内容
        String content = new String(Files.readAllBytes(Paths.get(javaFilePath)));
    } catch (IOException e) {
        throw new RuntimeException("Error reading cache Java file: " + javaFilePath);
    }
}

```

```

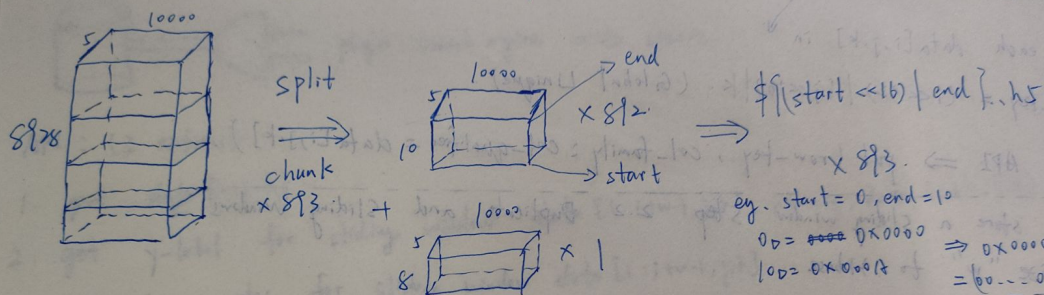
        // 提取包名（如果有的话）
        Pattern packagePattern = Pattern.compile("^package (\\w+\\.)*\\w+");
        Matcher packageMatcher = packagePattern.matcher(content);
        String packageName = "";
        if (packageMatcher.find()) {
            packageName = packageMatcher.group(1);
        }
        System.out.println("解析包名:"+packageName);

        // 提取类名
        Pattern classPattern = Pattern.compile("public class (\\w+\\.)*\\w+");
        Matcher classMatcher = classPattern.matcher(content);
        if (classMatcher.find()) {
            String className = classMatcher.group(1);
            System.out.println("解析类名:"+className);
            return (packageName.isEmpty() ? "" : packageName + "." + className);
        } else {
            return null;
        }
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
}

```

4. Summary

data : 89x8 x 1000 x 5 DOM Step 1 : Data Split



For each Split (.h5 file). Step 2: Data Dump

Filename $\{index\}.h5$

start = (index >> 16) & 0xFFFF

end = index & 0xFFFF

data[start:end, 0:8999, 0:4]

data[start:end, 0:999, 0:4]

Step 2.1 Generate Row Key.

For each data[i,j,k] in

row-key = $(i \ll 32) | (j \ll 16) | k$. (Global Unique)

Hbase API \Rightarrow put (row-key, col-family:col-qualifier = data[i,j,k])

How to store a sliding window with size "w" to Hbase

Step 2.2 Duplicate and Sliding Windows

Suppose the window starts with index i

data[i:i+w-1, j, k]

row-key = $(i \ll 32) | (j \ll 16) | k$.

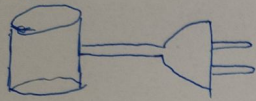
put (row-key, c-f:c-q = data[i+w,j,k])

and a total duplicate of data

Restore a sliding window:

Hbase API \Rightarrow get $((i+offset) \ll 32) | (j \ll 16) | k$. for each offset in $[0:w-1]$.

Hbase



plugin: search engine \Rightarrow users

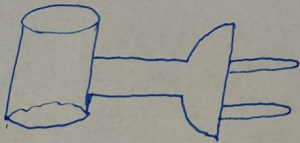
Step 3: Search Engine.

API: (3 options)

1. get data $[i, j, k]$
2. get y-label for sliding window data $[i:i+w-1, j, k]$, namely, data $[i+w, j, k]$
3. get all samples for sliding window data $[i:i+v-1, j, k]$

Step 4: ~~Mem~~ In-Memory Cache

Hbase



Cache



\Rightarrow users

data $[i, j, k] \Rightarrow$

hit \Rightarrow user

missing \Rightarrow Hbase \Rightarrow

Cache full: replace algorithms

Cache idle: store in Cache

LRU

FIFO

MFC

Priority

...

\Rightarrow users