海量数据处理中的云计算

C9. HBase（三）

北京邮电大学信息与通信工程学院

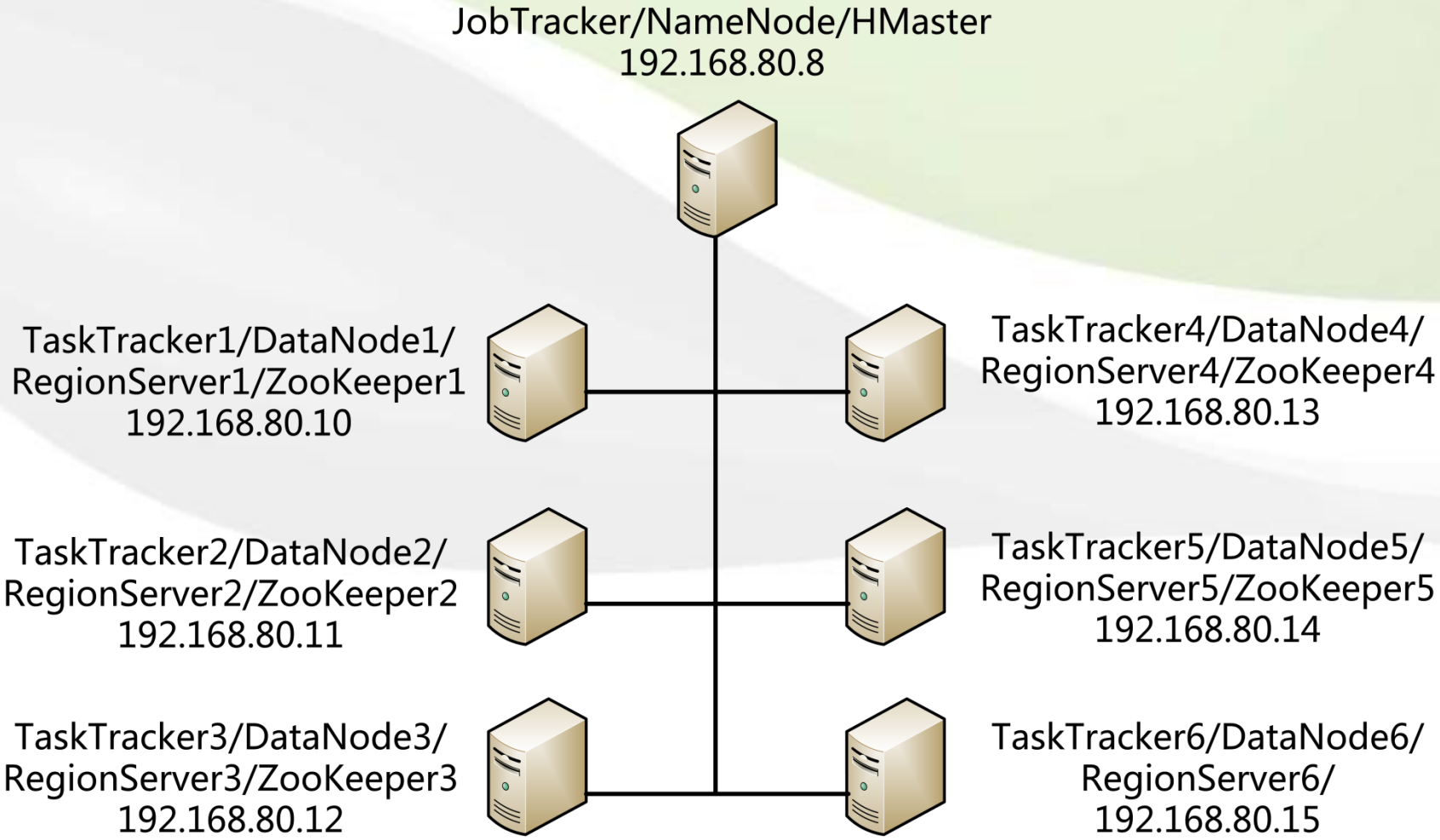
2013年春季学期

本节目录

- 测试环境
- 数据导入方法
- 实例分析

测试环境

系统部署


JobTracker/NameNode/HMaster
192.168.80.8
TaskTracker1/DataNode1/
RegionServer1/ZooKeeper1
192.168.80.10
TaskTracker2/DataNode2/
RegionServer2/ZooKeeper2
192.168.80.11
TaskTracker3/DataNode3/
RegionServer3/ZooKeeper3
192.168.80.12
TaskTracker4/DataNode4/
RegionServer4/ZooKeeper4
192.168.80.13
TaskTracker5/DataNode5/
RegionServer5/ZooKeeper5
192.168.80.14
TaskTracker6/DataNode6/
RegionServer6/
192.168.80.15

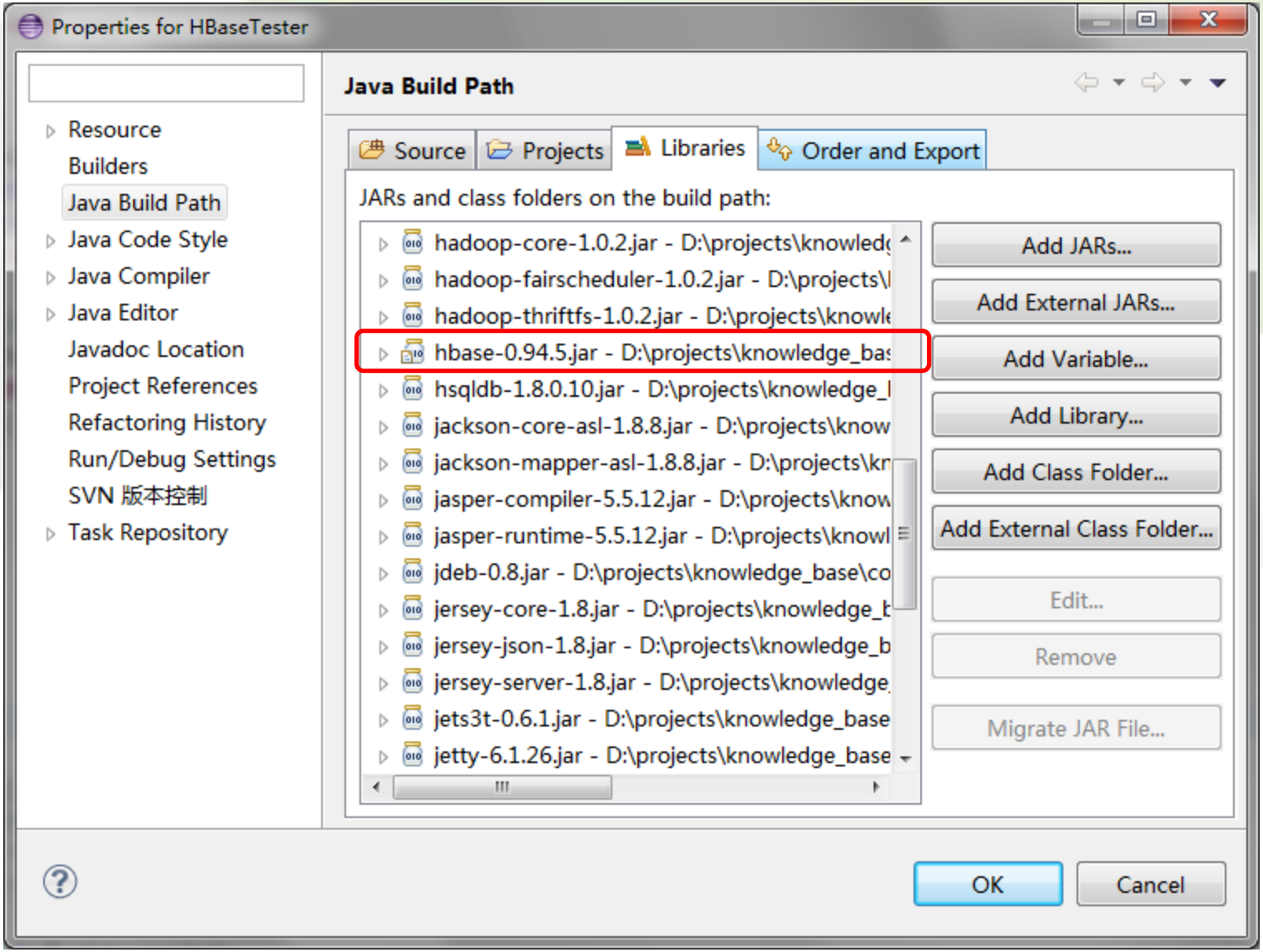
数据源

- 数据文件
  - 某地1天的网络流量
  - 986个文件，共864GB，存放在NameNode服务器的/data目录下

- 数据格式

	A	B	C	D	E	F	G
1	time	userID	deviceID	typeIndex	zoneID	traffic	...
2	记录产生时间	用户标识	终端标识	终端型号	小区标识	流量	...

HBase开发环境


Properties for HBaseTester
Resource
Builders
Java Build Path
Java Code Style
Java Compiler
Java Editor
Javadoc Location
Project References
Refactoring History
Run/Debug Settings
SVN 版本控制
Task Repository
Java Build Path
Source
Projects
Libraries
Order and Export
JARs and class folders on the build path:
hadoop-core-1.0.2.jar - D:\projects\knowled
hadoop-fairscheduler-1.0.2.jar - D:\projects\
hadoop-thriftfs-1.0.2.jar - D:\projects\knowle
hbase-0.94.5.jar - D:\projects\knowledge_bas
hsqldb-1.8.0.10.jar - D:\projects\knowledge_
jackson-core-asl-1.8.8.jar - D:\projects\know
jackson-mapper-asl-1.8.8.jar - D:\projects\kn
jasper-compiler-5.5.12.jar - D:\projects\know
jasper-runtime-5.5.12.jar - D:\projects\knowl
jdeb-0.8.jar - D:\projects\knowledge_base\co
jersey-core-1.8.jar - D:\projects\knowledge_b
jersey-json-1.8.jar - D:\projects\knowledge_b
jersey-server-1.8.jar - D:\projects\knowledge
jets3t-0.6.1.jar - D:\projects\knowledge_base
jetty-6.1.26.jar - D:\projects\knowledge_base
Add JARs...
Add External JARs...
Add Variable...
Add Library...
Add Class Folder...
Add External Class Folder...
Edit...
Remove
Migrate JAR File...
OK
Cancel

导入方法

数据导入方法

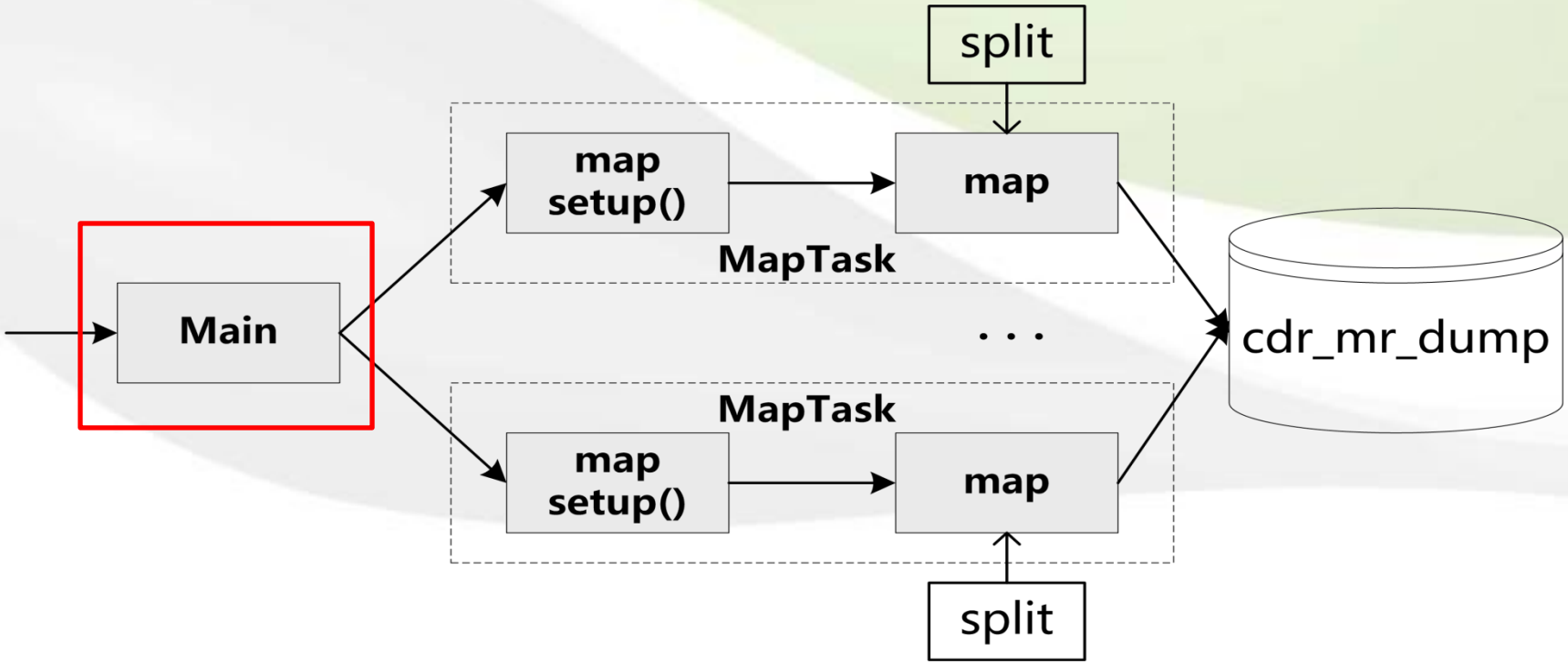
- 三种方式
  - MapReduce导入
  - HFile导入
  - Java多线程导入

导入数据（1） - MapReduce导入

- 导入表结构

	A	B	C	D	E
1	**rowkey**	**列族: cf**			
2		**限定词:deviceID**	**限定词:typeIndex**	**限定词:zoneID**	**限定词:content**
3	userID+time	deviceID	typeIndex	zoneID	entireLine

导入数据（1） - MapReduce导入流程



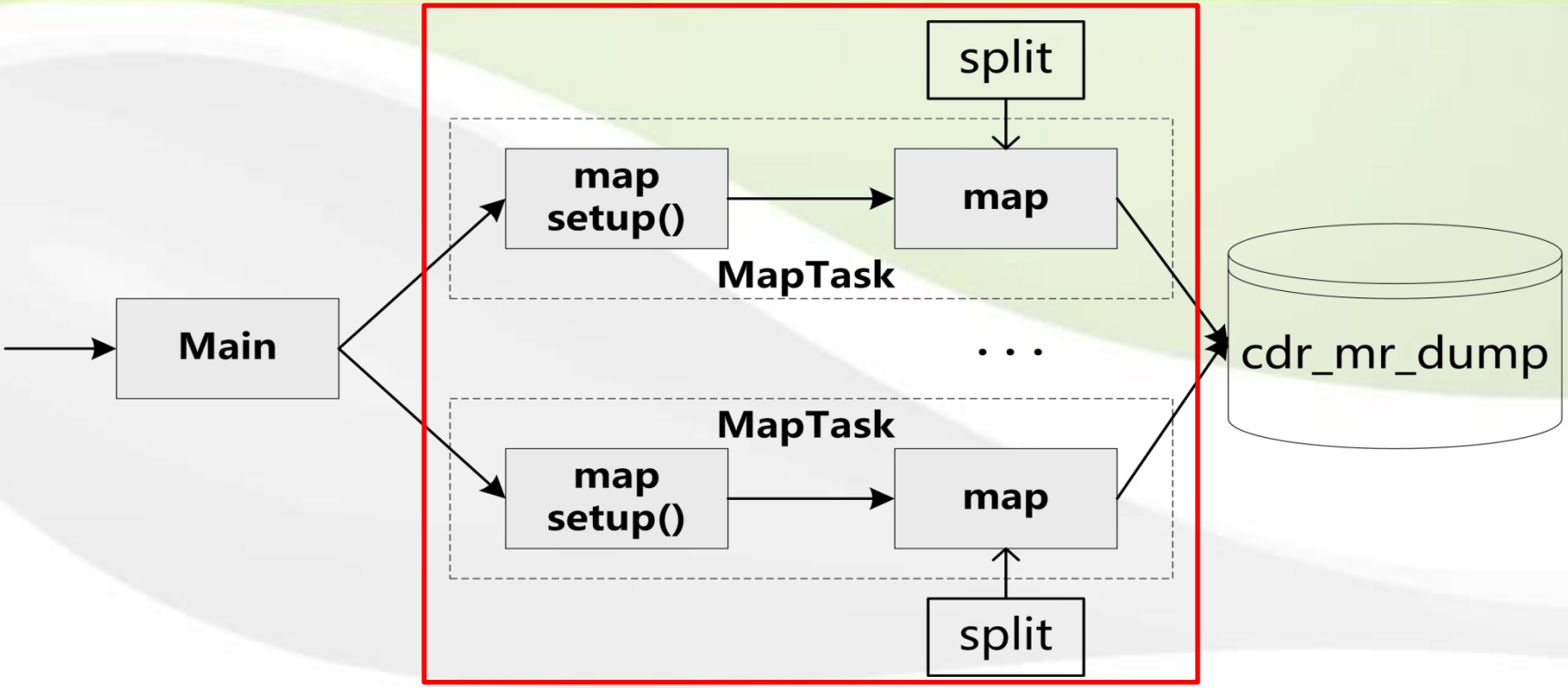
导入数据（1） - Main

```
1:  public static void main(String[] args) throws Exception {
2:    Configuration conf = new Configuration(); // 初始化MapReduce配置对象
3:    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
4:    if (otherArgs.length != 3) { // 判断参数输入格式是否合法
5:      System.err.println("Usage: inputPath fileNumberPerJob startIndex");
6:      System.exit(2);
7:    }
8:    File path = new File(otherArgs[0]); // 文件路径
9:    File[] fileList = path.listFiles(); // 提取文件列表
10:   int fileNumberPerJob = Integer.parseInt(otherArgs[1]); // 每个Job处理的文件数量
      // 需要读取的第一个文件索引，用于支持多次导入一个目录下的文件
11:   int startIndex = Integer.parseInt(otherArgs[2]);
12:   int filesToBeRead = fileList.length - startIndex;
13:   while (filesToBeRead > 0) { // 为每fileNumberPerJob个文件启动一个job
14:     Job job = new Job(conf, "cdrMRDump" + index);
        // 设置job参数
15:     job.setJarByClass(ImportMapReduceIndexColumn.class);
16:     job.setMapperClass(cdrMapper.class);
17:     job.setOutputFormatClass(NullOutputFormat.class);
```

导入数据（1） - Main（cont.）

```
18:     job.setNumReduceTasks(0); // 不需要reduce
19:     job.setInputFormatClass(LogFileFormat.class);
        // 将每个job负责的文件路径添加到job中
22:     if (filesToBeRead < fileNumberPerJob) {
23:       for (int i = 0; i < filesToBeRead; i++) {
24:         FileInputFormat.addInputPath(job, new Path(fileList[index + i].getPath()));
25:       }
26:     } else {
27:       for (int i = 0; i < fileNumberPerJob; i++) {
28:         FileInputFormat.addInputPath(job, new Path(fileList[index + i].getPath()));
29:       }
30:     }
31:     job.waitForCompletion(true);
32:     index += fileNumberPerJob;
33:     filesToBeRead -= fileNumberPerJob;
34:   }
35: }
```

导入数据（1） - MapReduce导入 Map




```
1:   public void run(Context context) throws IOException, InterruptedException {
2:     setup(context);
3:     while (context.nextKeyValue()) {
4:       map(context.getCurrentKey(), context.getCurrentValue(), context);
5:     }
6:     cleanup(context);
7:   }
```

导入数据（1） - Map setup()

```
1:  public static class cdrMapper extends Mapper<Long, LogLine, NullWritable, NullWritable> {
2:    public static Configuration conf = HBaseConfiguration.create();
3:    String tableName = "cdr_mr_dump"; // 写入的表名
4:    Put put = null; // put对象
5:    HTableInterface table = null; // 表操作对象
6:    HTablePool pool = null; // 表连接线程池
      // setup函数，在每个task启动时，会执行一次
7:    public void setup(Context context) {
8:      System.out.println("Loading cdr_mr_dump table.....");
9:      try {
10:       tablePool = new HTablePool(conf, 1); // 初始化HTablePool对象
11:       table = tablePool.getTable(tableName); // 加载或创建表
12:       table.setAutoFlush(false); // 禁用自动提交机制
13:       System.out.println("Load cdr_mr_dump successfully!");
14:     } catch (Exception e) {
15:       e.printStackTrace();
16:     }
```

导入数据（1） - Map

```
1:   public void map(Long key, LogLine value, Context context) throws IOException {
2:     try {
3:       if (value.column_num == 84) { // 判断每行数据是否合法
4:         long[] offset = value.log_offset; // 字段偏移数组
5:         byte[] logvalue = value.log_value; // 字段值数组
6:         int bytes_all = value.bytes_all; // 该行数据总字节数
7:         String time = getValue(offset, logvalue, 0); // 记录产生时间字段
8:         String deviceID = getValue(offset, logvalue, 5); // 终端标识字段
9:         String userID = getValue(offset, logvalue, 4); // 用户标识字段
10:        String typeIndex = ""; // 终端型号字段
11:        if (deviceID.length() >= 8) {
12:          typeIndex = deviceID.substring(0, 8); // 提取终端型号字段
13:        }
14:        String LAC = getValue(offset, logvalue, 24); // 小区标识字段1
15:        String CI = getValue(offset, logvalue, 25); // 小区标识字段2
           // 以userID+time为rowkey
16:        byte[] cdrRowKey = new byte[Bytes.toBytes(userID).length + Bytes.toBytes(time).length];
```

导入数据（1） - Map（cont.）

```
        // 构造rowkey
17:     Bytes.putBytes(cdrRowKey, 0, Bytes.toBytes(userID), 0, Bytes.toBytes(userID).length);
18:     Bytes.putBytes(cdrRowKey, Bytes.toBytes(userID).length, Bytes.toBytes(time), 0,
19:       Bytes.toBytes(time).length);
20:     Put put = new Put(cdrRowKey); // 设置put的rowkey
        // 存入同一列族cf的四个列
22:     put.add("cf".getBytes(), "deviceID".getBytes(), deviceID.getBytes());
23:     put.add("cf".getBytes(), "typeIndex".getBytes(), typeIndex.getBytes());
24:     put.add("cf".getBytes(), "zoneID".getBytes(), (LAC + CI).getBytes());
25:     put.add("cf".getBytes(), "content".getBytes(), getLine(logvalue, bytes_all));
26:     table.put(put); // 执行put操作，但并不触发flush，而要等到缓存满或显式flush
27:     }
28:   } catch (ArrayIndexOutOfBoundsException e) {
29:     e.printStackTrace();
30:   }
31: }
```

导入数据（1） - Map cleanup()

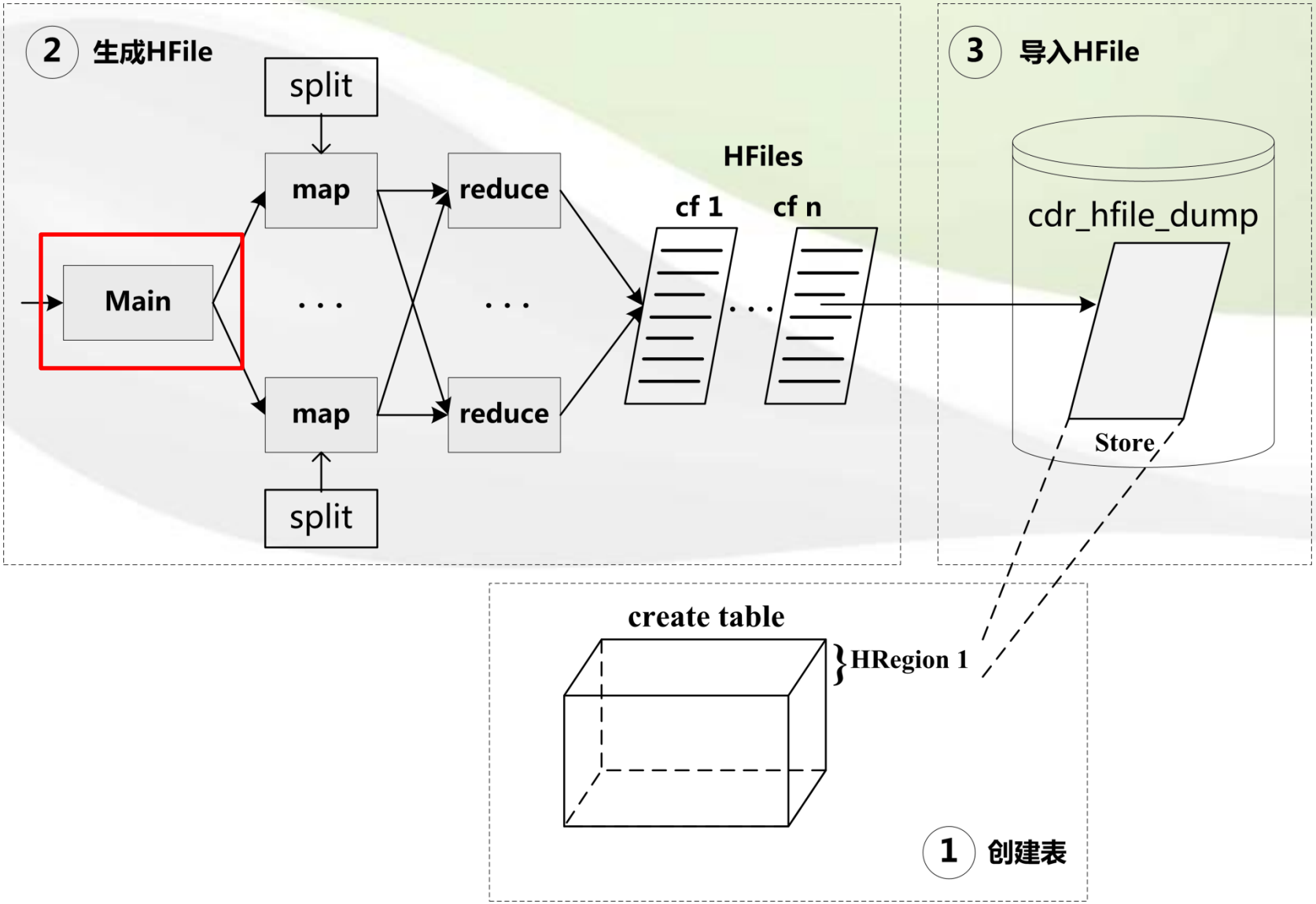
```
    // cleanup函数，在每个task结束时，会执行一次
1:  public void cleanup(Context context) throws IOException {
2:    table.close(); // 关闭表
3:    tablePool.close(); // 关闭连接
4:  }
```

导入数据（2） - HFile导入

- 导入表结构

	A	B
1	**rowkey**	**列族: cf**
2		**限定词:content**
3	userID+time	entireLine

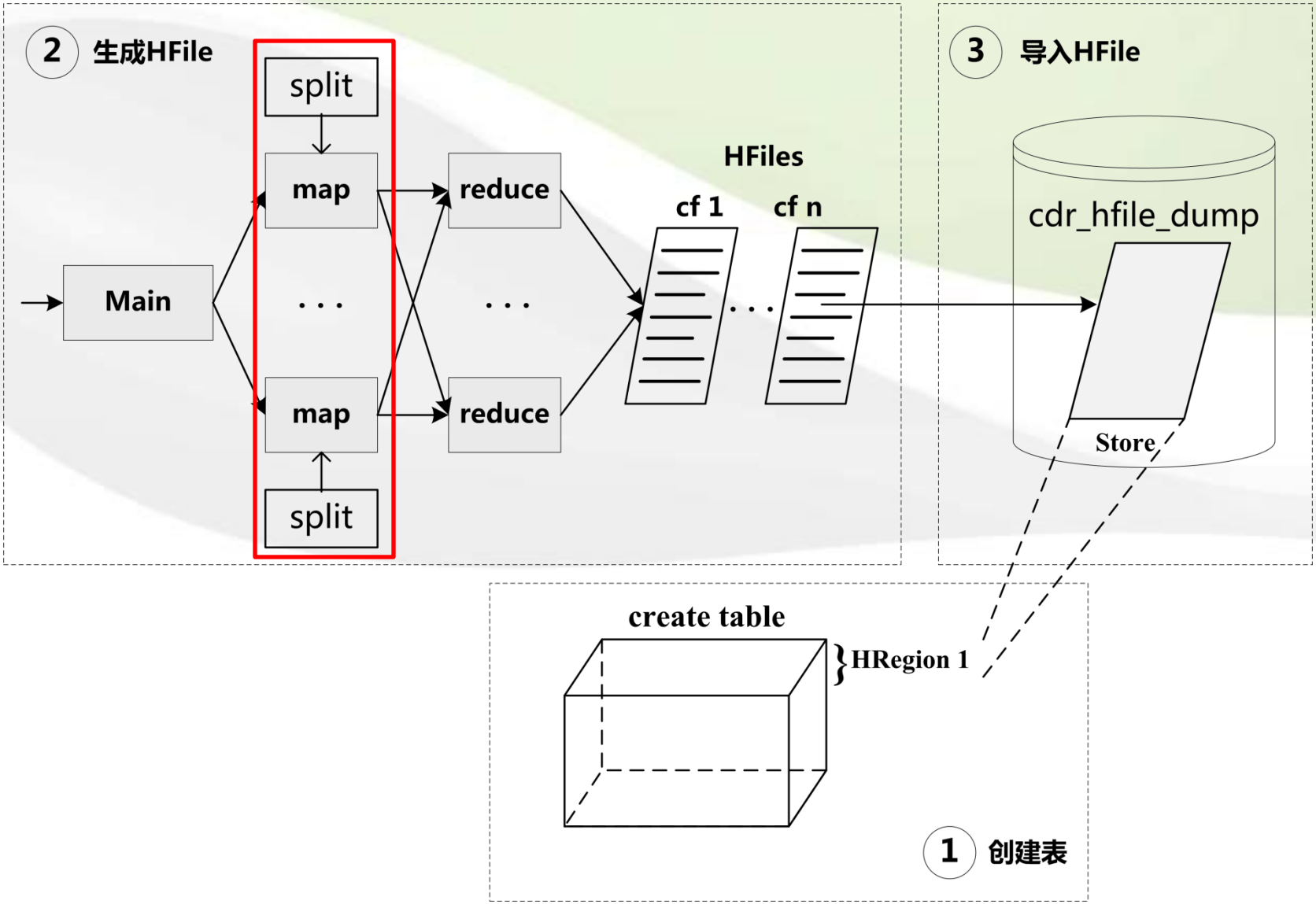
导入数据（2） - HFile导入流程



导入数据（2） - HFile导入 - 生成HFile的main

```
1:  public static void main(String[] args) throws IOException, InterruptedException {
2:    Configuration conf = HBaseConfiguration.create(); // 初始化mapreduce配置对象
      // 初始化作业信息
3:    Job job = new Job(conf, "cdr_hfile_dump");
4:    job.setJarByClass(HFileGenerator.class);
5:    FileInputFormat.addInputPath(job, new Path(args[0])); // 输入数据文件路径
6:    FileOutputFormat.setOutputPath(job, new Path(args[1])); // 输出HFile文件路径
      // MapReduce相关设置
7:    job.setMapperClass(HFileGeneratorMapper.class); // 自定义map类
8:    job.setInputFormatClass(LogFileFormat.class); // 输入数据格式
9:    job.setOutputKeyClass(ImmutableBytesWritable.class); // map输出的key类型
10:   job.setOutputValueClass(KeyValue.class); // map输出的value类型
11:   job.setReducerClass(KeyValueSortReducer.class); // HBase提供的reduce类
12:   job.setOutputFormatClass(HFileOutputFormat.class); // 输出数据格式
13:   HTable table = new HTable(conf, "cdr_hfile_dump"); // 写入表名称
14:   HFileOutputFormat.configureIncrementalLoad(job, table); // HBase提供的排序输出类
15:   System.exit(job.waitForCompletion(true) ? 0 : 1); // 启动作业
16: }
```

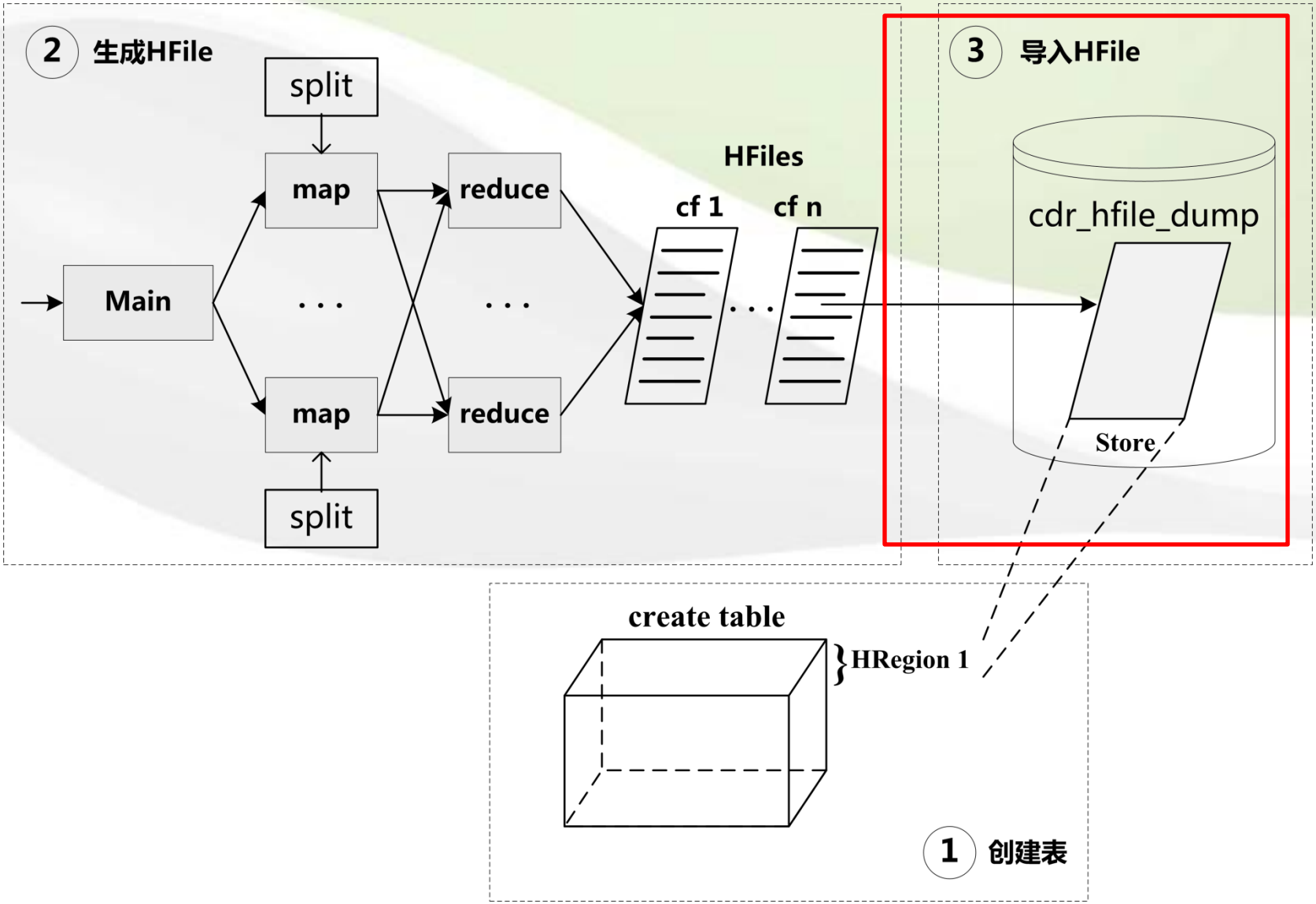
导入数据（2） - HFile导入流程 map



导入数据（2） - HFile导入 - 生成HFile的map

```
1:   protected void map(Object key, LogLine value, Context context) throws IOException {
2:     if (value.column_num == 84) { // 判断每行数据是否合法
3:       long[] offset = value.log_offset; // 字段偏移数组
4:       byte[] logvalue = value.log_value; // 字段值数组
5:       String time = getValue(offset, logvalue, 0); // 记录产生时间字段
6:       String userID = getValue(offset, logvalue, 4); // 用户标识字段
         // 以userID+time为rowkey，并构造rowkey
7:       byte[] cdrRowKey = new byte[Bytes.toBytes(userID).length + Bytes.toBytes(time).length];
8:         Bytes.putBytes(cdrRowKey, 0, Bytes.toBytes(userID), 0, Bytes.toBytes(userID).length);
9:       Bytes.putBytes(cdrRowKey, Bytes.toBytes(userID).length, Bytes.toBytes(time), 0,
10:        Bytes.toBytes(time).length);
         // 构造用于HFileOutputFormat的key/value对
11:      ImmutableBytesWritable k = new ImmutableBytesWritable(cdrRowKey);
12:      KeyValue kvContent = new KeyValue(cdrRowKey, "cf".getBytes(), "content".getBytes(),
13:        getLine(value.log_value, value.bytes_all)); // HBase提供的构造kv值函数
14:      context.write(k, kvContent); // 输出
15:    }
16:  }
```

导入数据（2） - HFile导入流程 导入



导入数据（2） - HFile导入 - 导入HFile

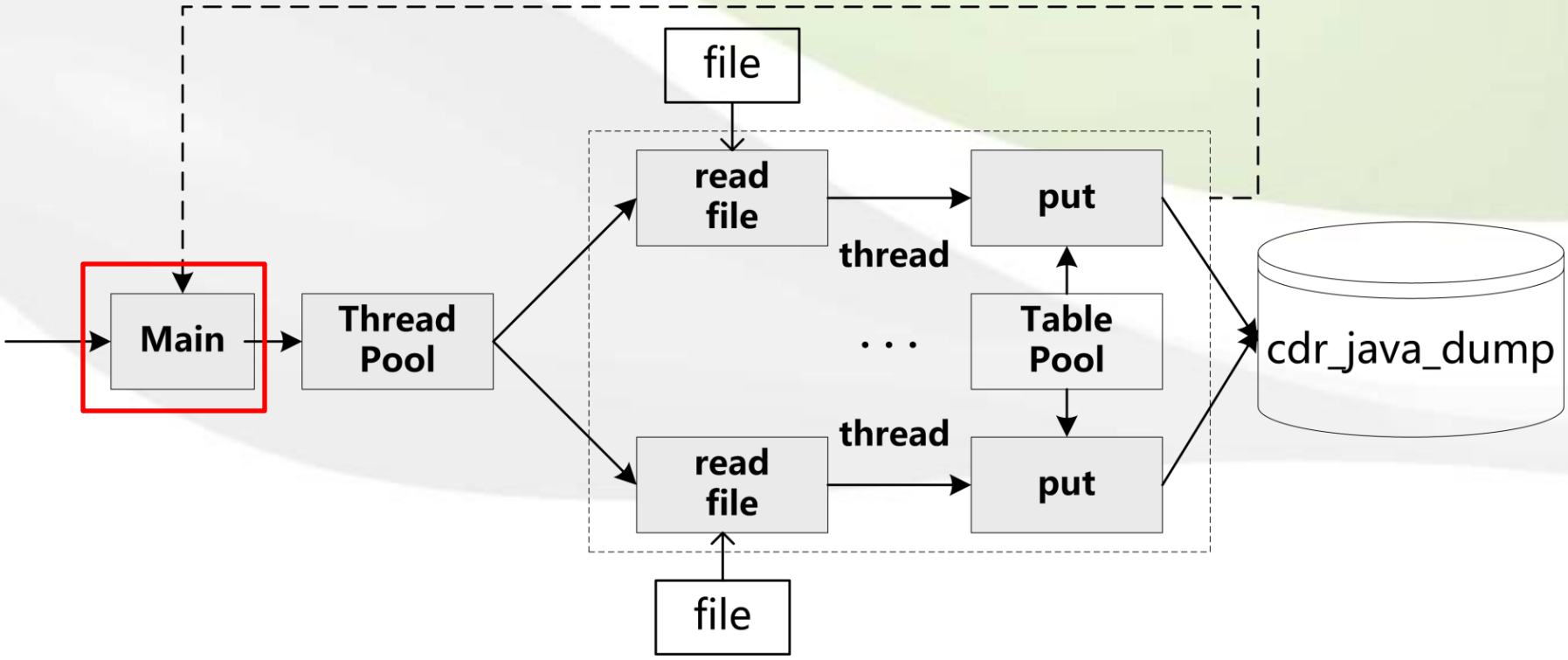
```
1:  public static void main(String[] args) throws Exception {
2:    Configuration conf = HBaseConfiguration.create();
3:    byte[] tableName= Bytes.toBytes(args[0]); // 输入参数为表名
4:    HTable table = new HTable(conf, tableName);
5:    LoadIncrementalHFiles loader = new LoadIncrementalHFiles(conf);
6:    loader.doBulkLoad(new Path(args[1]), table);
7:  }
```

导入数据（3） - Java多线程导入

- 导入表结构

	A	B
1	**rowkey**	**列族: cf**
2		**限定词:content**
3	userID+time	entireLine

导入数据（3） - Java多线程导入流程



导入数据（3） - Java导入 - main

```
1:  public static void main(String[] args) throws Exception {
2:    if (args.length < 4) { // 程序使用方法提示
3:      System.out.println("Usage: inputPath threadNumber poolSize startIndex endIndex");
4:      System.exit(2);
5:    }
6:    File path = new File(args[0]); // 输入数据文件所在路径
7:    File[] fileList = path.listFiles(); // 输入数据文件列表
8:    int threadNumber = Integer.parseInt(args[1]); // 导入线程数
9:    int poolSize = Integer.parseInt(args[2]); // 数据表连接池大小
10:   int startIndex = Integer.parseInt(args[3]); // 起始文件索引
11:   int endIndex = Integer.parseInt(args[4]); // 结束文件索引
12:   int fileNumber = endIndex- startIndex; // 文件数量
13:   tablePool =  new HTablePool(conf, poolSize); // 创建表连接池
      // 如果数据表不存在则创建
14:   HBaseAdmin admin = new HBaseAdmin(conf);
15:   if (!admin.tableExists(tableName)) {
16:     HTableDescriptor tableDescriptor = new HTableDescriptor(tableName);
17:     tableDescriptor.addFamily(new HColumnDescriptor("cf"));
18:     admin.createTable(tableDescriptor);
19:   }
20:   admin.close();
```

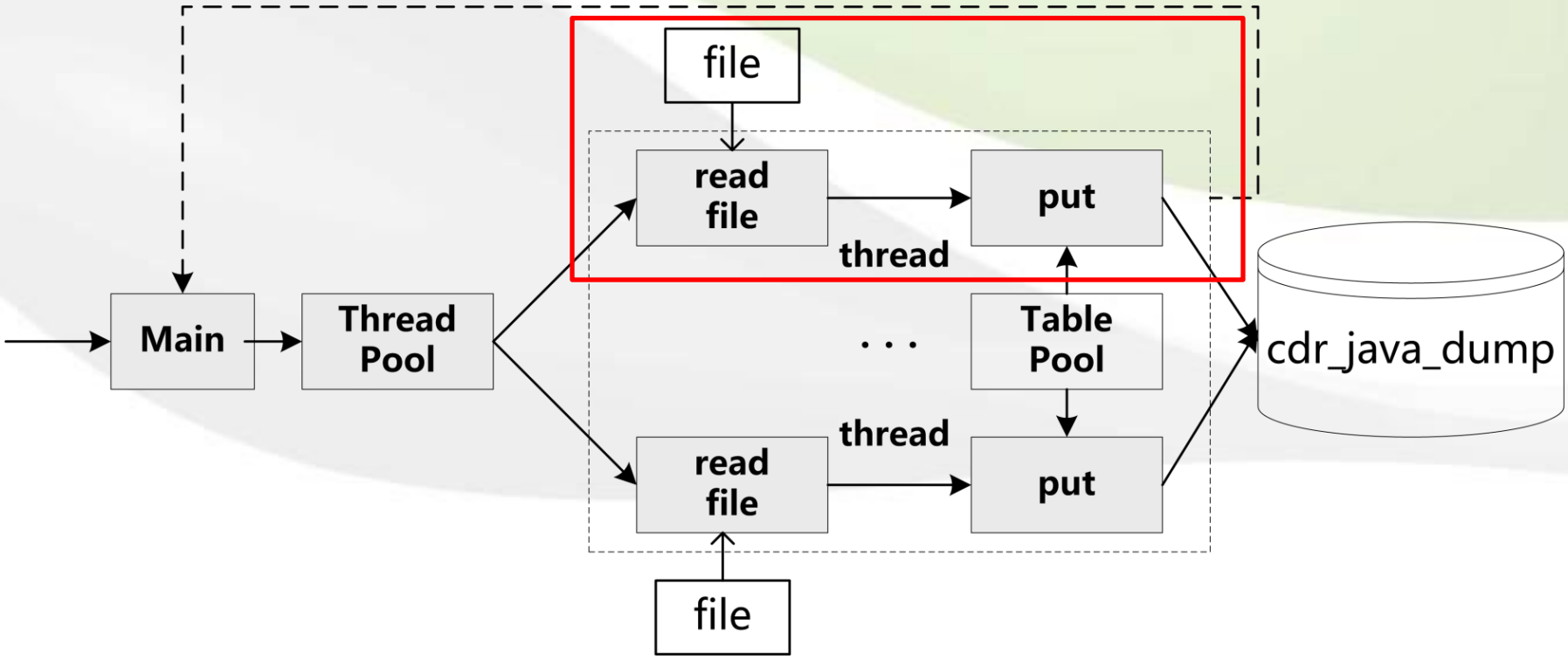
导入数据（3） - Java导入 - main（cont.）

```
21:   Thread[] threadPool = new Thread[threadNumber]; // 初始化线程池
22:   int filesToBeRead = fileList.length - startIndex; // 剩余需要读取的文件数量
23:   while (fileNumber > 0 && filesToBeRead > 0) {
24:     if (filesToBeRead < threadNumber) { // 剩余文件数量小于线程数
25:       for (int i = 0; i < filesToBeRead; i++) { // 为每个文件创建一个导入线程
26:         threadPool[i] = new HBaseImportThread(i, fileList[startIndex + i].getCanonicalPath(),
27:           tablePool);
28:       }
29:       for (int i = 0; i < filesToBeRead; i++) {
30:         threadPool[i].join(); // 等待子线程结束后后续代码方可继续执行
31:       }
32:     } else { // 剩余文件数量大于等于线程数
33:       for (int i = 0; i < threadNumber; i++) { // 为每个文件创建一个导入线程
34:         threadPool[i] = new HBaseImportThread(i, fileList[startIndex + i].getCanonicalPath(),
35:           tablePool);
36:       }
37:       for (int i = 0; i < threadNumber; i++) {
38:         threadPool[i].join(); // 等待子线程结束后后续代码方可继续执行
39:       }
```

导入数据（3） - Java导入 - main（cont.）

```
40:      startIndex += threadNumber; // 更新起始文件索引
41:      fileNumber -= threadNumber; // 更新总文件数量
42:      filesToBeRead -= threadNumber; // 更新剩余文件数量
43:    }
44:    tablePool.close(); // 关闭连接池
45:  }
```

导入数据（3） - Java多线程导入 HBaseImportThread



导入数据（3） - Java导入 - HBaseImportThread

```
1:  class HBaseImportThread extends Thread {
2:    private LogReader reader; // 读取数据文件类
3:    private HTableInterface table; // 表对象
4:    private int threadIndex; // 线程索引
5:    private HTablePool tablePool; // 数据库连接池
6:    public HBaseImportThread(int threadIndex, String fileName, HTablePool tablePool) {
7:      try {
8:        System.out.println("Thread index is " + threadIndex + ". File name is " + fileName);
9:        this.reader = new LogReader(fileName);
10:     } catch (IOException e) {
11:       e.printStackTrace();
12:     }
13:     this.threadIndex = threadIndex; // 保存线程索引
14:     this.tablePool = tablePool; // 保存表连接池
15:     start(); // 启动线程
16:   }
```

导入数据（3） - Java导入 - HBaseImportThread

```
17:   public void run() {
18:     table = tablePool.getTable("cdr_java_dump"); // 开启表
19:     table.setAutoFlush(false); // 禁用自动提交
20:     while (reader.readLine()) { // 逐行读取数据文件
21:       String[] columnList;
22:       if (reader.getColumnNum() == 84) { // 判断数据格式合法性
23:         columnList = reader.getColumnValueAll(); // 读取数据全部列
24:         byte[] userID = Bytes.toBytes(columnList[4]);
25:         byte[] time = Bytes.toBytes(columnList[0]);
26:         byte[] rowKey = new byte[userID.length + time.length]; // 以userID+time构造rowkey
27:         Bytes.putBytes(rowKey, 0, userID, 0, userID.length);
28:         Bytes.putBytes(rowKey, userID.length, time, 0, time.length);
29:         Put put = new Put(rowKey); // put数据
30:         put.add("cf".getBytes(), "content".getBytes(), reader.getLogBytes());
31:         table.put(put);
32:       }
33:     }
34:     table.flushCommits(); // 读取完后显式提交一次
35:     table.close(); // 关闭表
36:   }
37: }
```

数据导入方法总结

- MapReduce导入：
  - 初期速度较快
  - TaskTracker与RegionServer竞争节点资源，后续会变慢
- HFile导入：
  - 生成HFile的过程比较慢
  - 生成HFile后写入HBase非常快
  - 只适用于空表导入
- Java导入
  - Client和RegionServer分开，硬盘读写分开，效率较高
  - 应采用多客户端、多线程同时入库

实例分析

需求分析

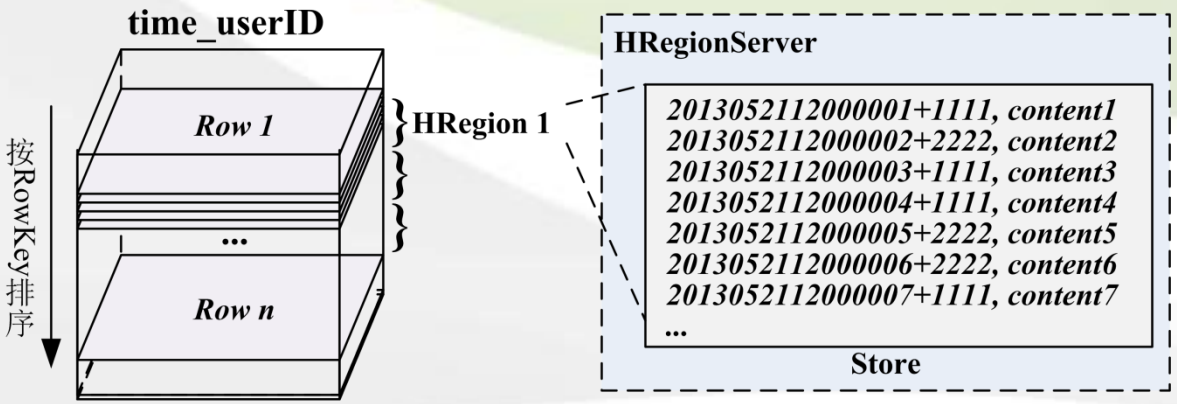
- 源数据：

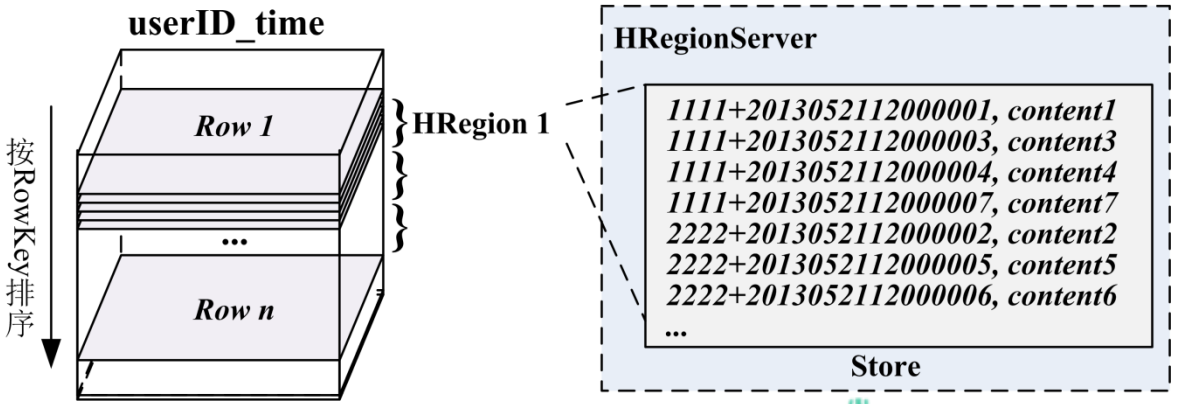
	A	B	C	D	E	F	G
1	time	userID	deviceID	typeIndex	zoneID	traffic	...
2	记录产生时间	用户标识	终端标识	终端型号	小区标识	流量	...

- 检索条件：
  - 记录产生时间（time）是必选条件，最小粒度为5分钟
  - 可能的单索引：userID（最常用）、zoneID、traffic
  - 可能的组合索引：单索引的任意两两组合
- 性能要求
  - 一天数据量864GB，当天完成导入
  - 以userID查询，首结果时延不超过5秒
  - 以zoneID、traffic查询，首结果时延不超过30秒
- 数据表设计：？
- 程序结构：单线程 vs. 多线程？

数据表设计 - 主表

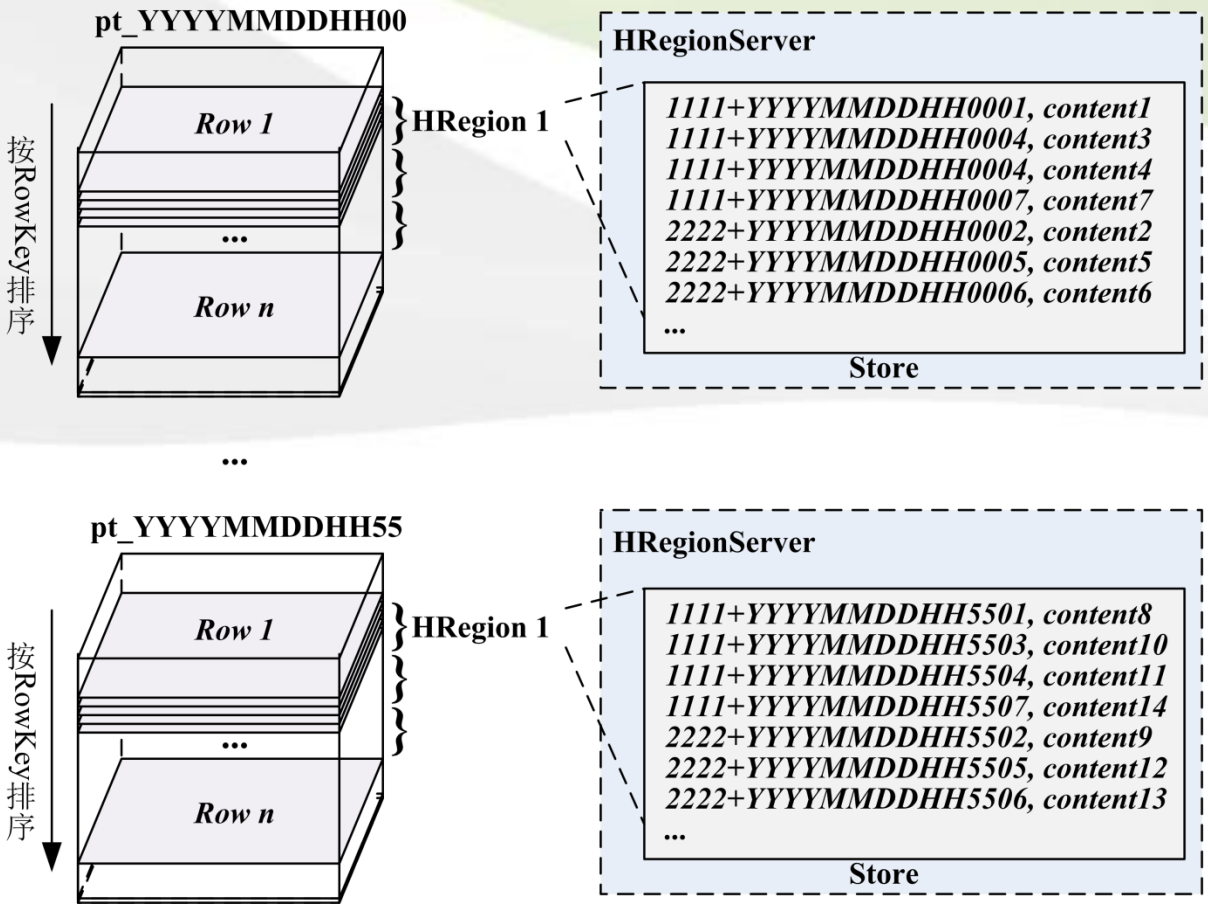
- 主表rowkey设计
  - 检索条件：
    - 记录产生时间（time）是必选条件，最小粒度为5分钟
    - 可能的单索引：userID（最常用）、zoneID、traffic
  - rowkey：
    - time+userID？
    - userID+time？
  - 所有数据存在1张表中？





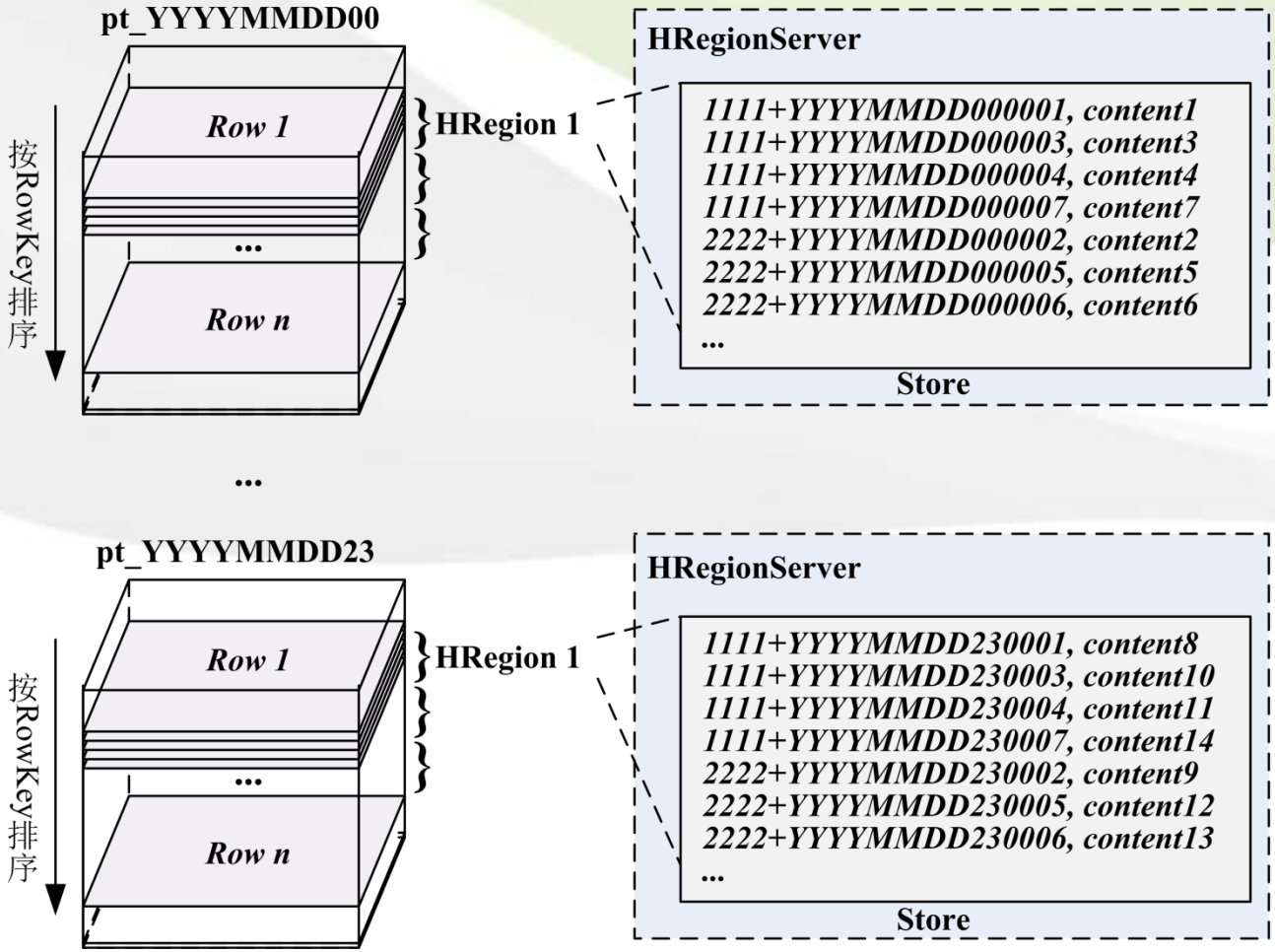
数据表设计 - 主表

- 主表rowkey设计
  - 检索条件：
    - 记录产生时间（time）是必选条件，最小粒度为5分钟
    - 可能的单索引：userID（最常用）、zoneID、traffic
  - rowkey：
    - userID+time
  - 每5分钟一张表



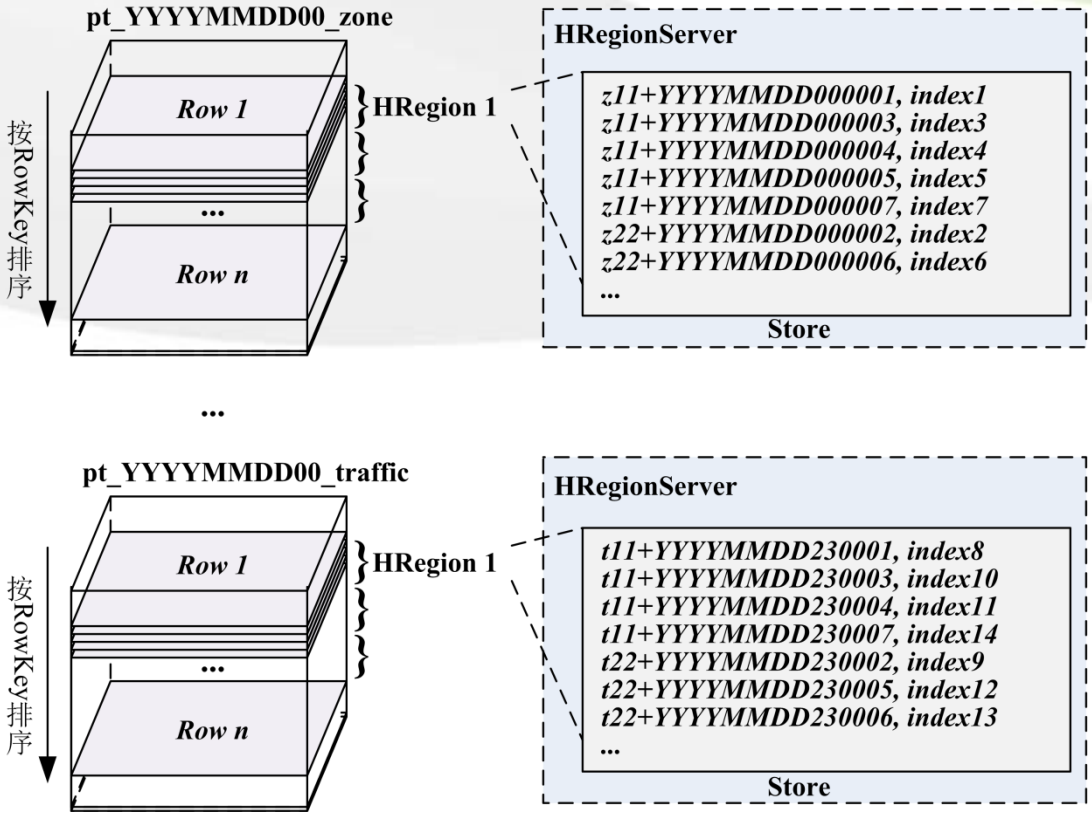
数据表设计 - 主表

- 优化主表
  - 每1小时一张表

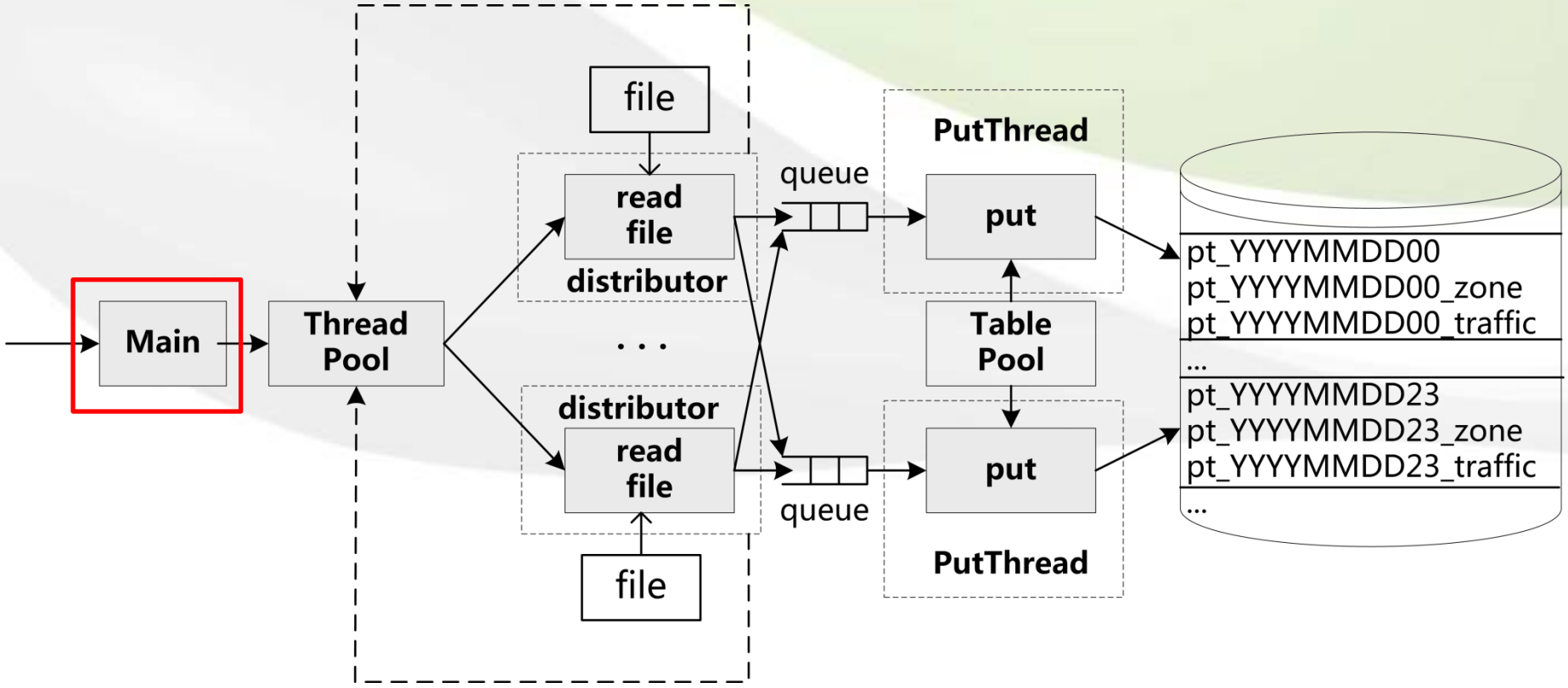


数据表设计 - 索引表

- 检索条件：
  - 记录产生时间（time）是必选条件，最小粒度为5分钟
  - 可能的单索引：userID（最常用）、**zoneID**、**traffic**
  - 可能的组合索引：单索引的任意两两组合
- index：主表的rowkey



数据导入 - 流程


Main
Thread Pool
file
read file
distributor
. . .
distributor
read file
file
queue
queue
PutThread
put
Table Pool
put
PutThread
pt_YYYYMMDD00
pt_YYYYMMDD00_zone
pt_YYYYMMDD00_traffic
...
pt_YYYYMMDD23
pt_YYYYMMDD23_zone
pt_YYYYMMDD23_traffic
...

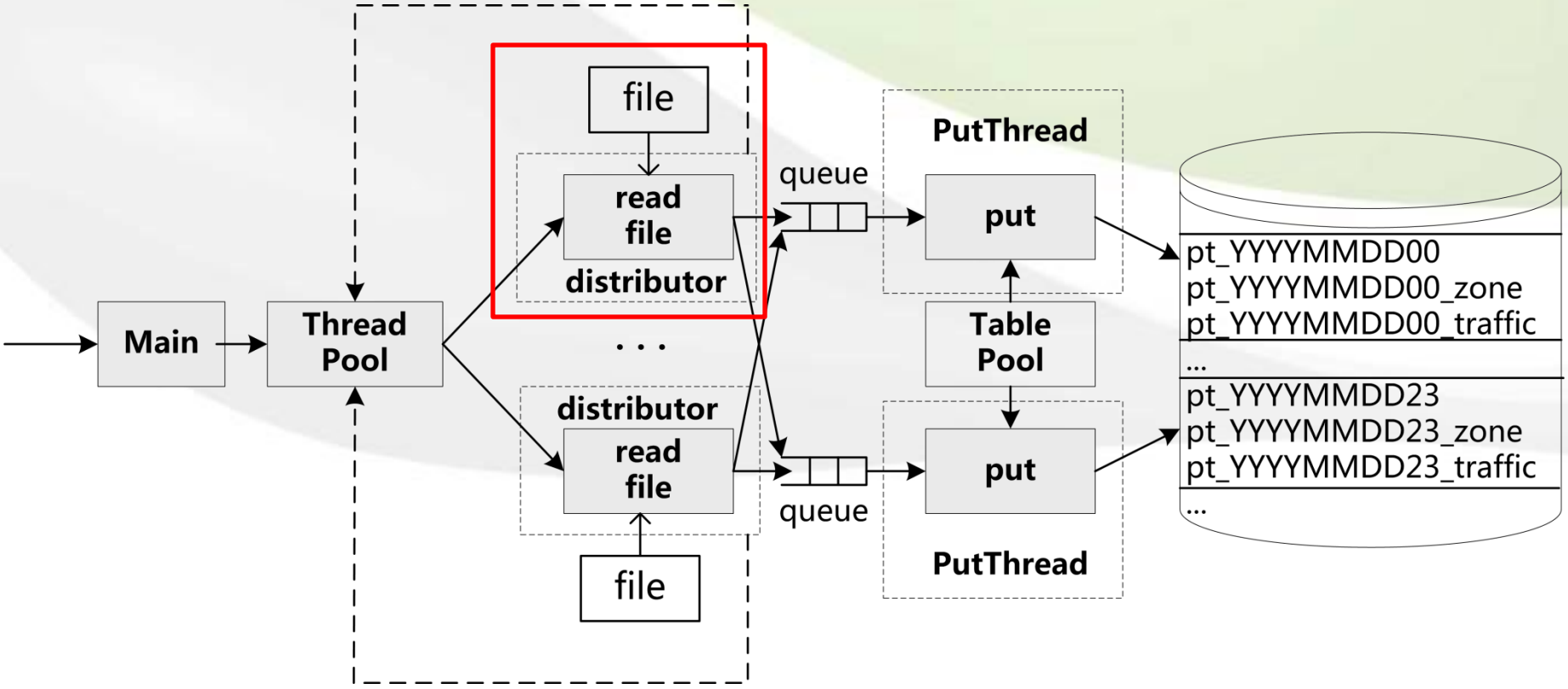
数据导入 - ImportClient - main

```
1:   public static void main(String[] args) {
2:     if (args.length < 4) { // 程序使用方法提示
3:       System.out.println("Usage: inputPath threadNumber startIndex endIndex");
4:       System.exit(2);
5:     }
6:     File path = new File(args[0]); // 数据文件所在路径
7:     File[] fileList = path.listFiles(); // 文件列表
8:     Arrays.sort(fileList); // 文件排序
9:     int threadNumber = Integer.parseInt(args[1]); // 分发线程数
10:    int startIndex = Integer.parseInt(args[2]); // 起始文件索引
11:    int endIndex = Integer.parseInt(args[3]); // 结束文件索引
12:    if (endIndex > fileList.length) { // 如果超过总文件数，则重置
13:      endIndex = fileList.length;
14:    }
15:    int filesToBeRead = endIndex - startIndex; // 需要读取的文件总数
16:    ExecutorService executor = Executors.newFixedThreadPool(threadNumber); // 并发线程环境
17:    for (int i = 0; i < threadNumber; i++) { // 为每个线程设置一个对应的文件路径，并启动线程
18:      DistributeThread distributeThread = new DistributeThread(i, fileList[startIndex + i].getPath());
19:      executor.execute(distributeThread); // 启动线程
20:      filesToBeRead--; // 待读取的文件数更新
21:    }
```

数据导入 - ImportClient - main（cont.）

```
22:   lock.lock(); // 为并发线程同步的lock
23:   try {
24:     while (filesToBeRead > 0) { // 为剩下的文件启动相应分发线程
25:       condition.await(); // 等待有空闲线程
26:       DistributeThread distributeThread = new DistributeThread(endIndex-filesToBeRead,
27:         fileList[endIndex - filesToBeRead].getPath()); // // 为每个线程设置一个对应的文件路径
28:       executor.execute(distributeThread); // 启动线程
29:       filesToBeRead--; // 待读取的文件数更新
30:     }
31:     executor.shutdown(); // 关闭执行环境，不接受新线程请求，但当前运行线程可以正常完成
32:   } catch (InterruptedException e) {
33:     e.printStackTrace();
34:   } finally {
35:     lock.unlock(); // 取消lock
36:   }
37:   while (!executor.awaitTermination(30, TimeUnit.SECONDS)) {// 等待全部线程结束
38:     System.out.println("wait for executor shutdown");
39:   }
38:   DistributeThread.pool.close(); // 关闭数据表连接池
39:   DistributeThread.admin.close(); // 关闭admin对象
40: }
```

数据导入 - DistributeThread



数据导入 - DistributeThread

```
1:   public void run() {
2:     while (reader.readLine()) { // 逐行读取输入数据
3:       if (reader.getColumnNum() != 84) continue;
4:       byte[] userID = reader.getColumnBytes(4); // 用户ID
5:       byte[] time = reader.getColumnBytes(0); // 记录生成时间
6:       byte[] lac = reader.getColumnBytes(24); // lac
7:       byte[] ci = reader.getColumnBytes(25); // ci
8:       byte[] traffic = reader.getColumnBytes(41); // 下行流量
9:       byte[] tableRowkey = getUserIDTime(userID, time); // 主表rowkey
10:      byte[] indexTableZoneRowkey = getZoneIDTime(lac, ci, time); // 小区索引表rowkey
11:      byte[] indexTableTrafficRowkey = getTrafficTime(traffic, time); // 流量索引表rowkey
12:      byte[] line = reader.getLogBytes();
13:      distribute(new String(time),tableRowkey,indexTableZoneRowkey,indexTableTrafficRowkey, line);
14:    }
15:    reader.close(); // 关闭输入文件
16:    done.set(true); // 标记完成
17:    exit(); // 退出
18:  }
```

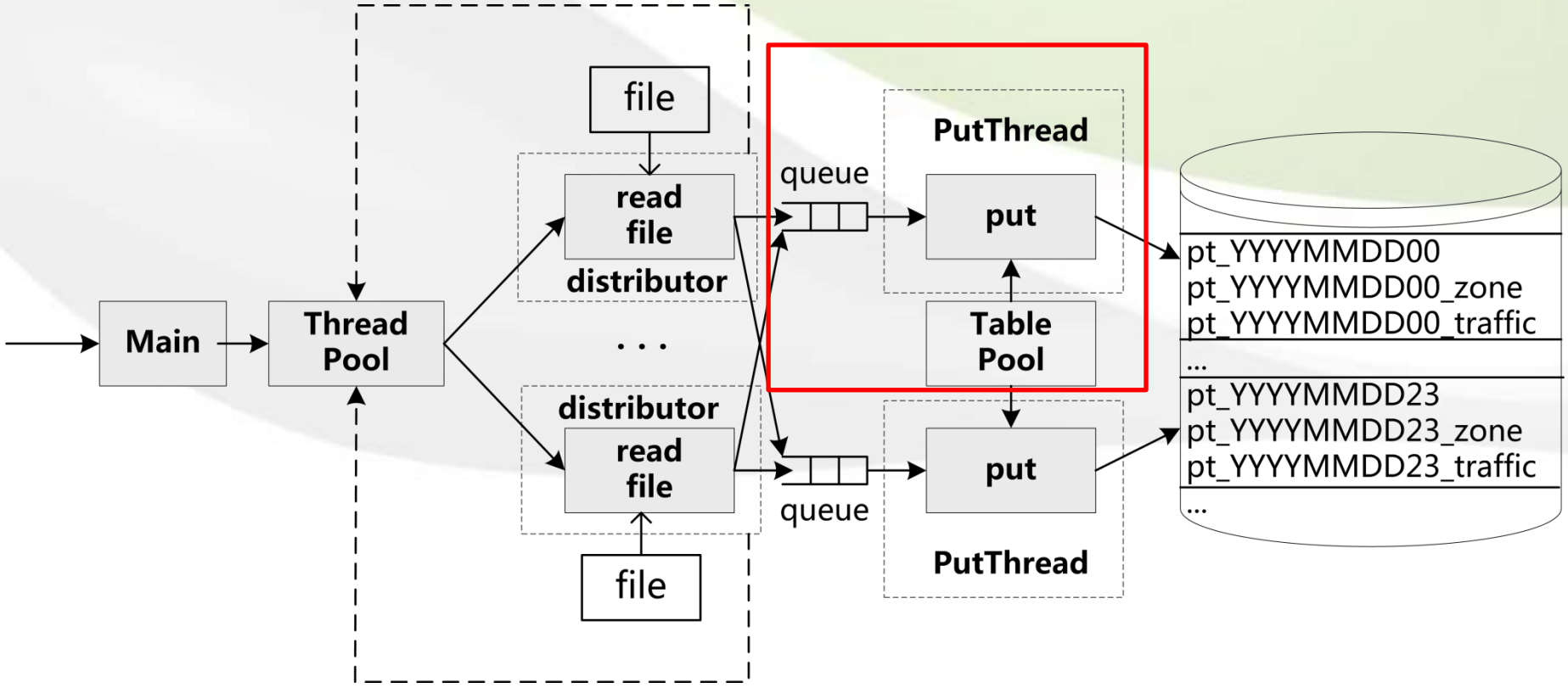
数据导入 - DistributeThread（cont.）

```
1:  public void distribute(String time, byte[] tableRowkey, byte[] indexTableZoneRowkey,
2:    byte[] indexTableTrafficRowkey, byte[] line) {
3:    long t = Long.parseLong(time.substring(0, time.indexOf("."))) / 3600 * 3600;
4:    String tableName = "pt_" + t; // 每小时1张表
5:    if (threadMap.containsKey(tableName)) { // 如果已存在对应的put线程，则直接使用
6:      PutThread thread = threadMap.get(tableName);
7:      thread.addRequest(tableRowkey, indexTableZoneRowkey, indexTableTrafficRowkey, line);
8:    } else { // 如果不存在对应的put线程，则创建表，并生成对应put线程
9:      createTable(tableName);
10:     PutThread thread = new PutThread(threadIndex, tableName, done);
11:     thread.start();
12:     thread.addRequest(tableRowkey, indexTableZoneRowkey, indexTableTrafficRowkey, line);
13:     threadMap.put(tableName, thread);
14:   }
15: }
```

数据导入 - DistributeThread（cont.）

```
1:  public synchronized void createTable(String tableName) {
2:    try {
3:      String indexTableZoneName = tableName + "_zone"; // 小区索引表名
4:      String indexTableTrafficName = tableName + "_traffic"; // 流量索引表名
5:      if (!admin.tableExists(tableName)) { // 主表不存在则需要创建
6:        HTableDescriptor tableDescriptor = new HTableDescriptor(tableName);
7:        tableDescriptor.addFamily(new HColumnDescriptor("cf"));
8:        admin.createTable(tableDescriptor);
9:      }
10:     if (!admin.tableExists(indexTableZoneName)) { // 小区索引表不存在则需要创建
11:       HTableDescriptor tableDescriptor = new HTableDescriptor(indexTableZoneName);
12:       tableDescriptor.addFamily(new HColumnDescriptor("cf"));
13:       admin.createTable(tableDescriptor);
14:     }
15:     if (!admin.tableExists(indexTableTrafficName)) { // 流量索引表不存在则需要创建
16:       HTableDescriptor tableDescriptor = new HTableDescriptor(indexTableTrafficName);
17:       tableDescriptor.addFamily(new HColumnDescriptor("cf"));
18:       admin.createTable(tableDescriptor);
19:     }
20:   } catch (org.apache.hadoop.hbase.TableExistsException e) {
21:     wait(5000);
22:   }
23: }
```

数据导入 - PutThread



数据导入 - PutThread

```
1:  public PutThread(int threadIndex, String tableName, AtomicBoolean done) {
2:    this.requestList = new LinkedBlockingQueue<Put[]>(10000);
3:    this.done = done;
4:    try {
5:      getTable(tableName, DistributeThread.tablePool);
6:    } catch (org.apache.hadoop.hbase.TableNotFoundException e) {
7:      this.wait(5000);
8:      getTable(tableName, DistributeThread.tablePool);
9:    }
10: }
11: public void getTable(String tablename, HTablePool pool)
12:   throws org.apache.hadoop.hbase.TableNotFoundException {
13:   table[0] = pool.getTable(tablename);
14:   table[1] = pool.getTable(tablename + "_zone");
15:   table[2] = pool.getTable(tablename + "_traffic");
16:   table[0].setAutoFlush(false);
17:   table[1].setAutoFlush(false);
18:   table[2].setAutoFlush(false);
19: }
```

数据导入 - PutThread（cont.）

```
20:  public void run() {
21:    while (!done.get()) {
22:      submitRequest();
23:    }
24:    exit();
25:  }
26:  public synchronized void submitRequest() throws InterruptedException, IOException {
27:    Put[] put = null;
28:    while ((put = requestList.poll(pollInterval, TimeUnit.SECONDS)) != null) {
29:      table[0].put(put[0]);
30:      table[1].put(put[1]);
31:      table[2].put(put[2]);
32:    }
33:  }
34:  public void exit() throws IOException {
35:    table[0].flushCommits();
36:    table[1].flushCommits();
37:    table[2].flushCommits();
38:    table[0].close();
39:    table[1].close();
40:    table[2].close();
41:  }
```

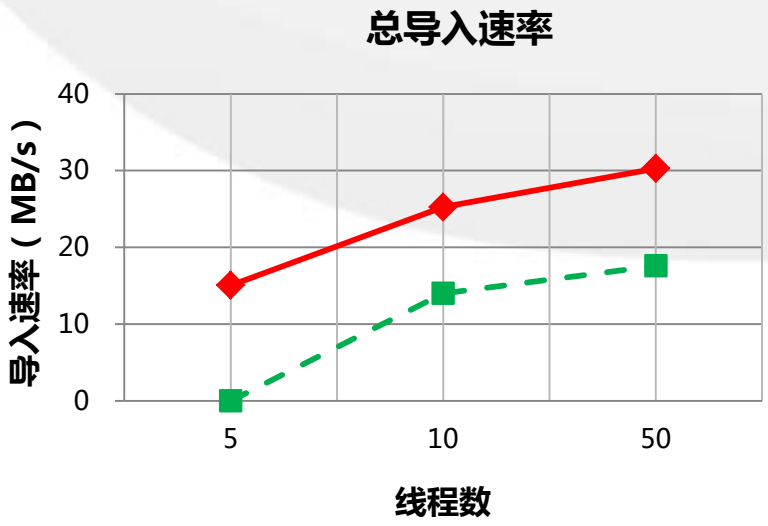
数据导入 - PutThread（cont.）

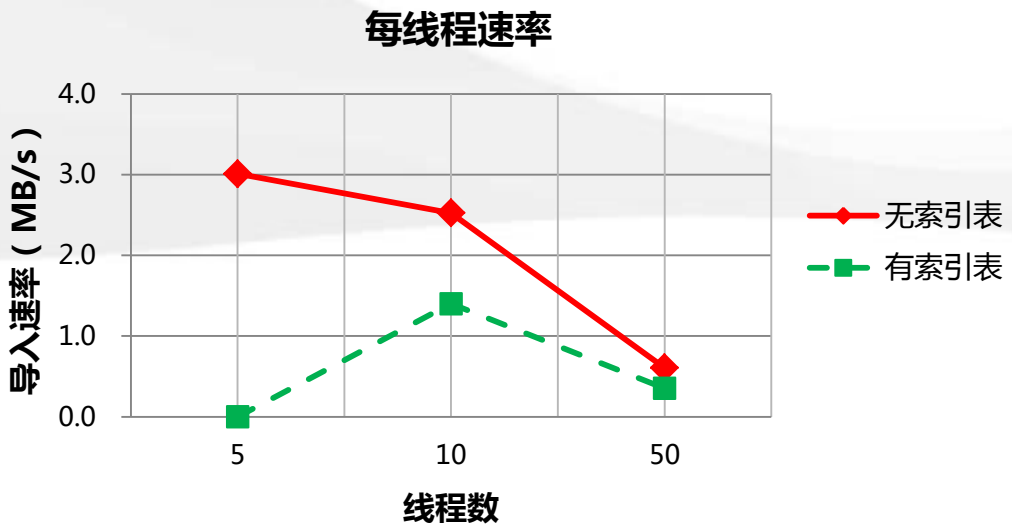
```
42:  public void addRequest(byte[] tableRowkey, byte[] indexTableZoneRowkey,
43:    byte[] indexTableTrafficRowkey, byte[] line) {
44:    Put[] put = new Put[3];
45:    put[0] = new Put(tableRowkey);
46:    put[0].add("cf".getBytes(), "content".getBytes(), line);
47:    put[1] = new Put(indexTableZoneRowkey);
48:    put[1].add("cf".getBytes(), "index".getBytes(), tableRowkey);
49:    put[2] = new Put(indexTableTrafficRowkey);
50:    put[2].add("cf".getBytes(), "index".getBytes(), tableRowkey);
51:    try {
52:      requestList.put(put);
53:    } catch (InterruptedException e) {
54:      e.printStackTrace();
55:    }
56:  }
```

数据导入性能分析

- 导入速率测试：

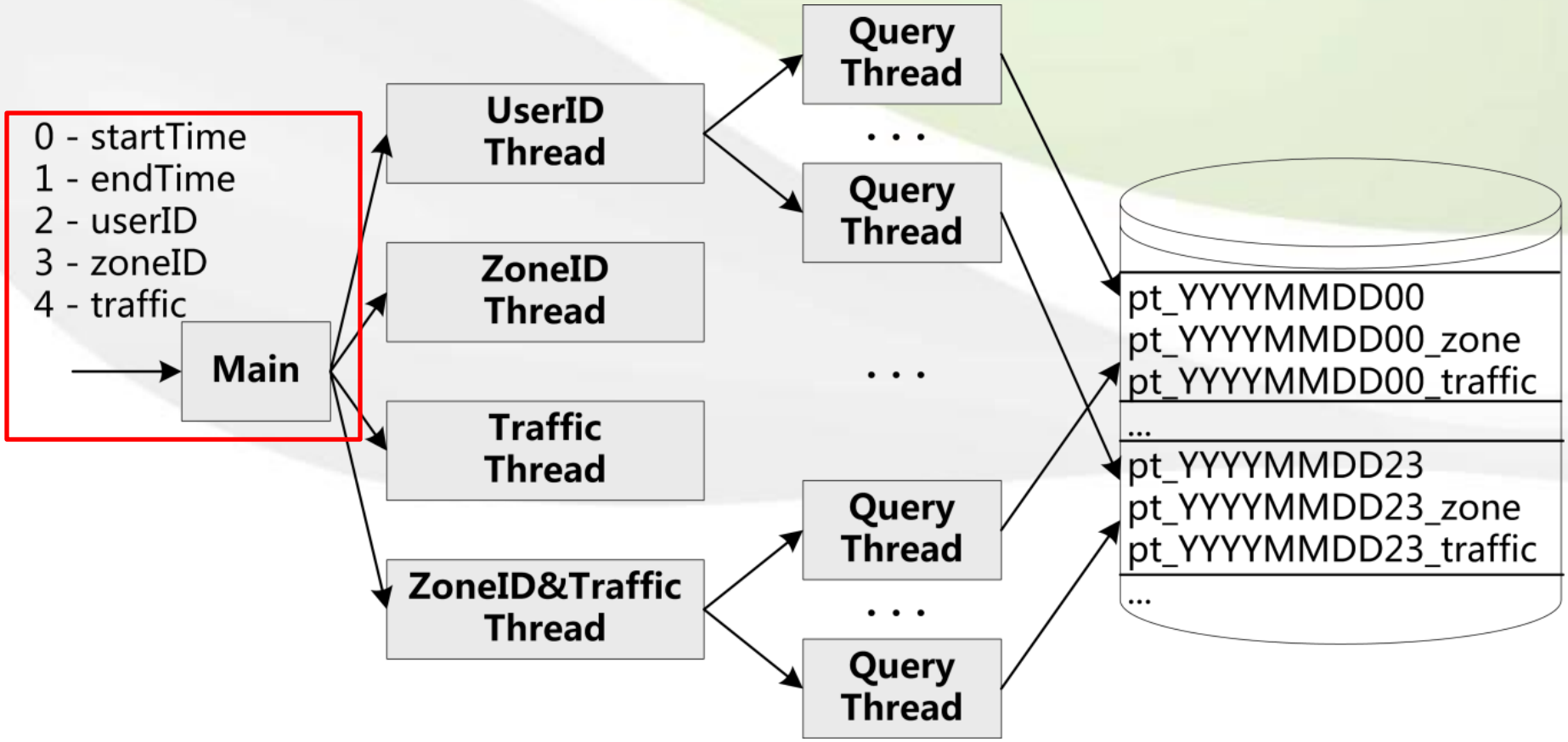
数据量（MB）	线程数	用时（s）	无索引表导入速率（MB/s）		有索引表速率（MB/s）	
			总速率	每线程速率	总速率	每线程速率
4771	5	317	15.1	3.0	-	-
9543	10	378	25.2	2.5	14	1.4
41482	50	1370	30.3	0.6	17.6	0.352





- 可满足当天数据当天完成导入

数据检索 - 流程


0 - startTime
1 - endTime
2 - userID
3 - zoneID
4 - traffic
Main
UserID Thread
ZoneID Thread
Traffic Thread
ZoneID&Traffic Thread
Query Thread
Query Thread
Query Thread
Query Thread
pt_YYYYMMDD00
pt_YYYYMMDD00_zone
pt_YYYYMMDD00_traffic
...
pt_YYYYMMDD23
pt_YYYYMMDD23_zone
pt_YYYYMMDD23_traffic
...

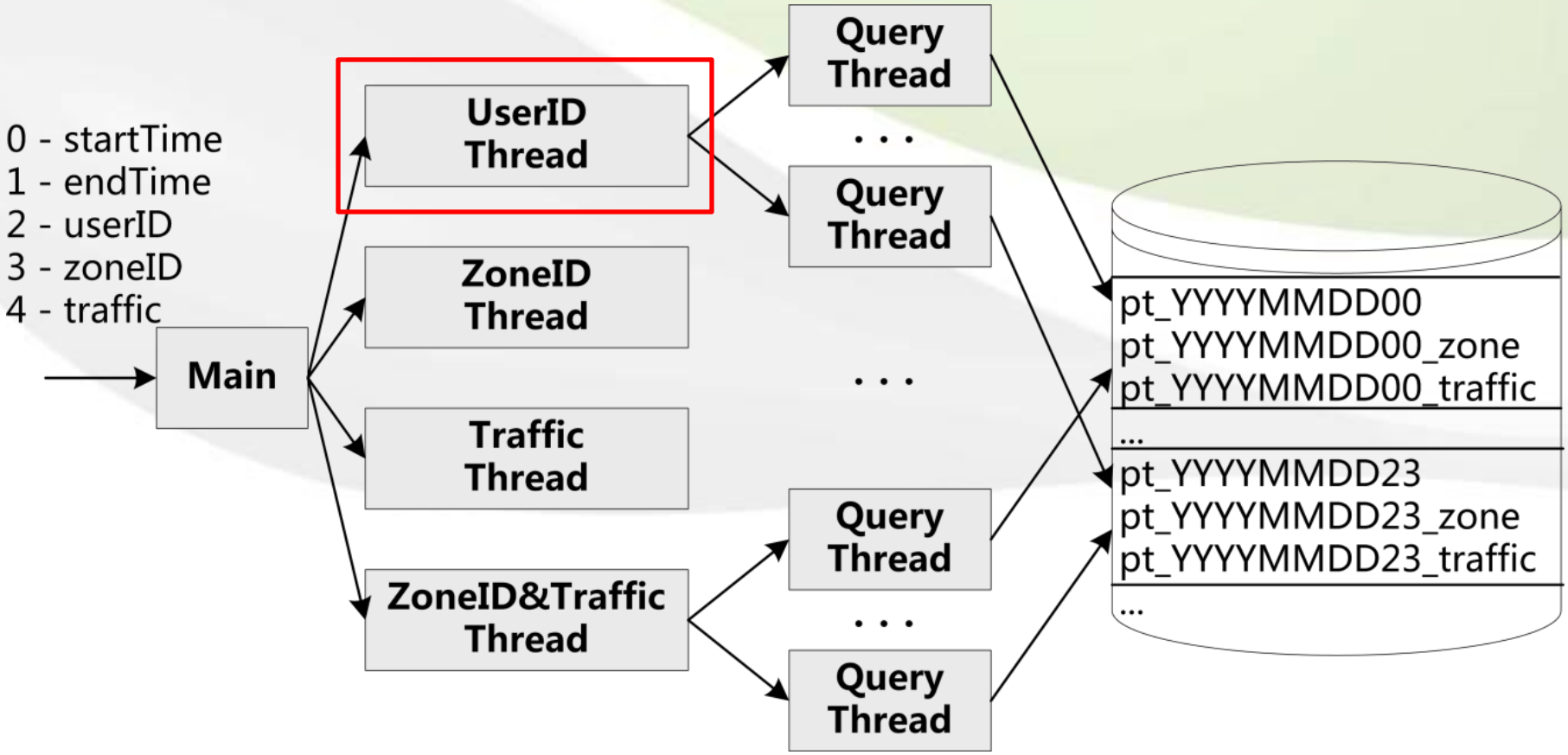
数据检索 - QueryClient

```
1:   public static void main(String[] args) {
2:     QueryClient client = new QueryClient();
3:     client.fileName = args[0]; // 结果文件路径
4:     for (int i = 1; i < args.length; i++) {
5:       String colConditionIndex = args[i]; // 查询条件索引
6:       String colConditionStr = args[++i]; // 查询条件字符串
7:       if (colConditionIndex.equals("0")) { // 起始时间
8:         client.startTime = Long.parseLong(colConditionStr);
9:       } else if (colConditionIndex.equals("1")) { // 结束时间
10:        client.endTime = Long.parseLong(colConditionStr);
11:      } else if (colConditionIndex.equals("2")) { // 用户ID
12:        client.userID = colConditionStr;
13:      } else if (colConditionIndex.equals("3")) { // 小区ID
14:        int index = colConditionStr.indexOf("_");
15:        client.zoneID = getZoneID(colConditionStr.substring(0, index).getBytes(),
16:        colConditionStr.substring(index + 1).getBytes());
17:      } else if (colConditionIndex.equals("4")) { // 下行流量
18:        client.traffic = getTraffic(colConditionStr.getBytes());
19:      }
20:    }
21:    client.run(); // 执行查询
22: }
```

数据检索 - QueryClient（cont.）

```
23:  public void run() throws IOException, InterruptedException {
24:    if (startTime == 0 || endTime == 0) { // 起始时间和结束时间是必选条件
25:      return;
26:    }
27:    if (zoneID == null && traffic == null) { // 仅使用用户ID查询
28:      QueryByUserIDThread query=new QueryByUserIDThread(startTime,endTime,userID,fileName);
29:      query.start();
30:    } else if (zoneID != null && traffic != null) { // 使用zoneID和traffic查询
31:      QueryByZoneIDAndTrafficThread query = new QueryByZoneIDAndTrafficThread(startTime,
32:        endTime, traffic, zoneID, fileName);
33:      query.start();
34:    } else if (zoneID != null) { // 仅使用zoneID查询
35:      QueryByZoneIDThread query=new QueryByZoneIDThread(startTime,endTime,zoneID,fileName);
36:      query.start();
37:    } else if (traffic != null) { // 仅使用traffic查询
38:      QueryByTrafficThread query = new QueryBytednThread(startTime, endTime, traffic, fileName);
39:      query.start();
40:    }
41:  }
```

数据检索 - QueryByUserIDThread

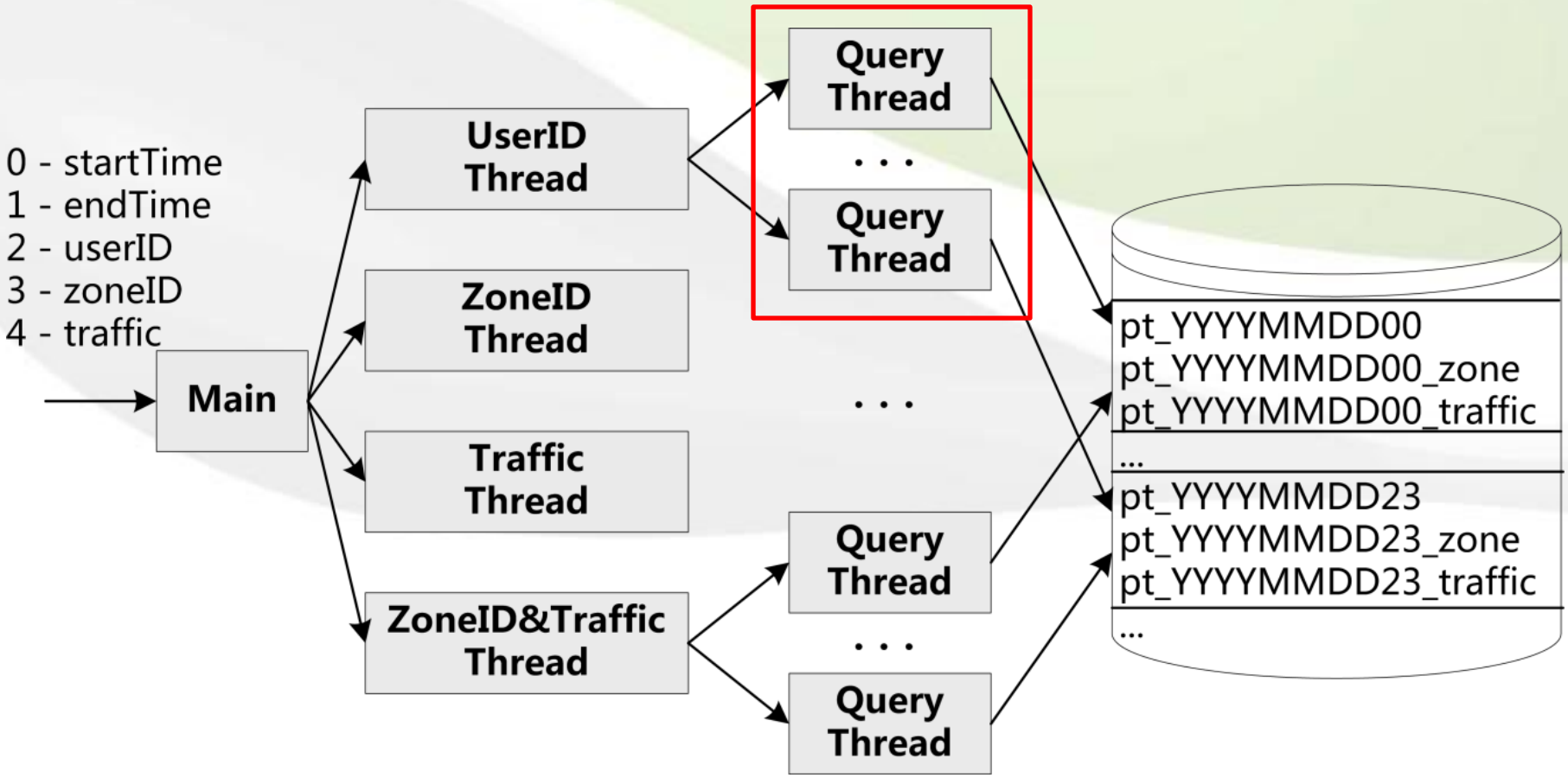


userID条件检索 - QueryByUserIDThread

```
1:  public QueryByUserIDThread(long startTime, long endTime, String userID, String fileName) {
2:    this.startTime = startTime; // 起始时间查询条件
3:    this.endTime = endTime; // 结束时间查询条件
4:    this.userID = userID; // 用户ID查询条件
5:    this.output = new FileOutputStream(fileName); // 结果文件
6:  }

7:  public void run() {
8:    int threadNum = (int) (endTime - startTime) / 3600 + 1; // 每小时的一张表对应一个线程
9:    signal = new CountDown(threadNum); // 初始化计数器
10:   for (int i = 0; i < threadNum; i++) { // 逐个启动查询线程
11:     QueryThread query = new QueryThread(startTime + i * 3600, userID, tablePool, signal);
12:     new Thread(query).start();
13:   }
14:   while (true) {
15:     writeResult(); // 保存结果
16:     if (signal.getCount() == 0) break; // 全部查询线程执行完毕后退出
17:   }
18:   exit();
19: }
```

数据检索 - userID条件检索 QueryThread

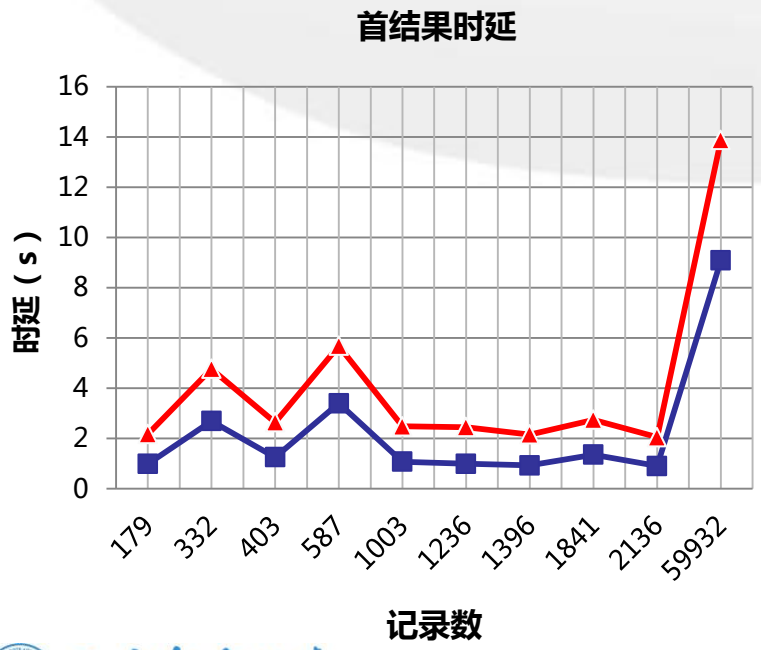


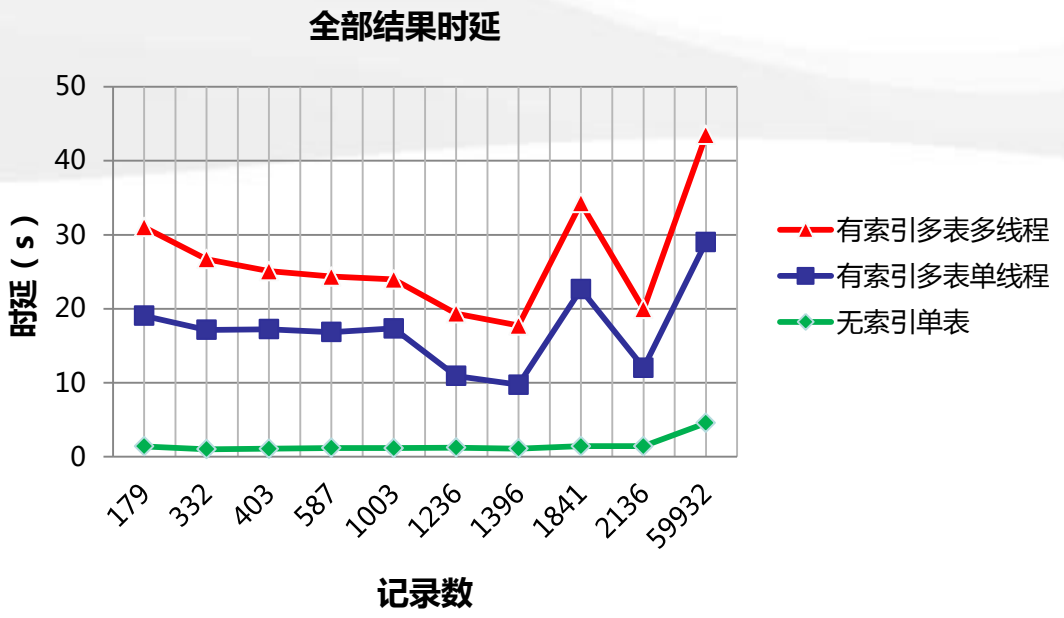
userID条件检索 - QueryThread

```
1:  public QueryThread(long time, String userID, HTablePool pool, CountDown signal) {
2:    tableName = "pt_" + time; // 表名称
3:    table = pool.getTable(tableName);
5:    this.signal = signal;
6:    this.userID = userID;
7:  }
8:  public void run() {
9:    Scan scan = new Scan(); // scan对象
10:   scan.addColumn("cf".getBytes(), "content".getBytes()); // 列
11:   scan.setCaching(500); // scan缓存
12:   scan.setCacheBlocks(false); // 不缓存块
13:   scan.setStartRow(Bytes.toBytes(userID)); // 起始rowkey
14:   scan.setStopRow(Bytes.toBytes(userID + "9")); // 结束rowkey
15:   ResultScanner rs = table.getScanner(scan);
16:   for (Result r : rs) {
17:     for (KeyValue kv : r.raw()) {
18:       resultQ.put(kv.getValue());
19:     }
20:   }
21:   signal.countDown();
22:   table.close();
23: }
```

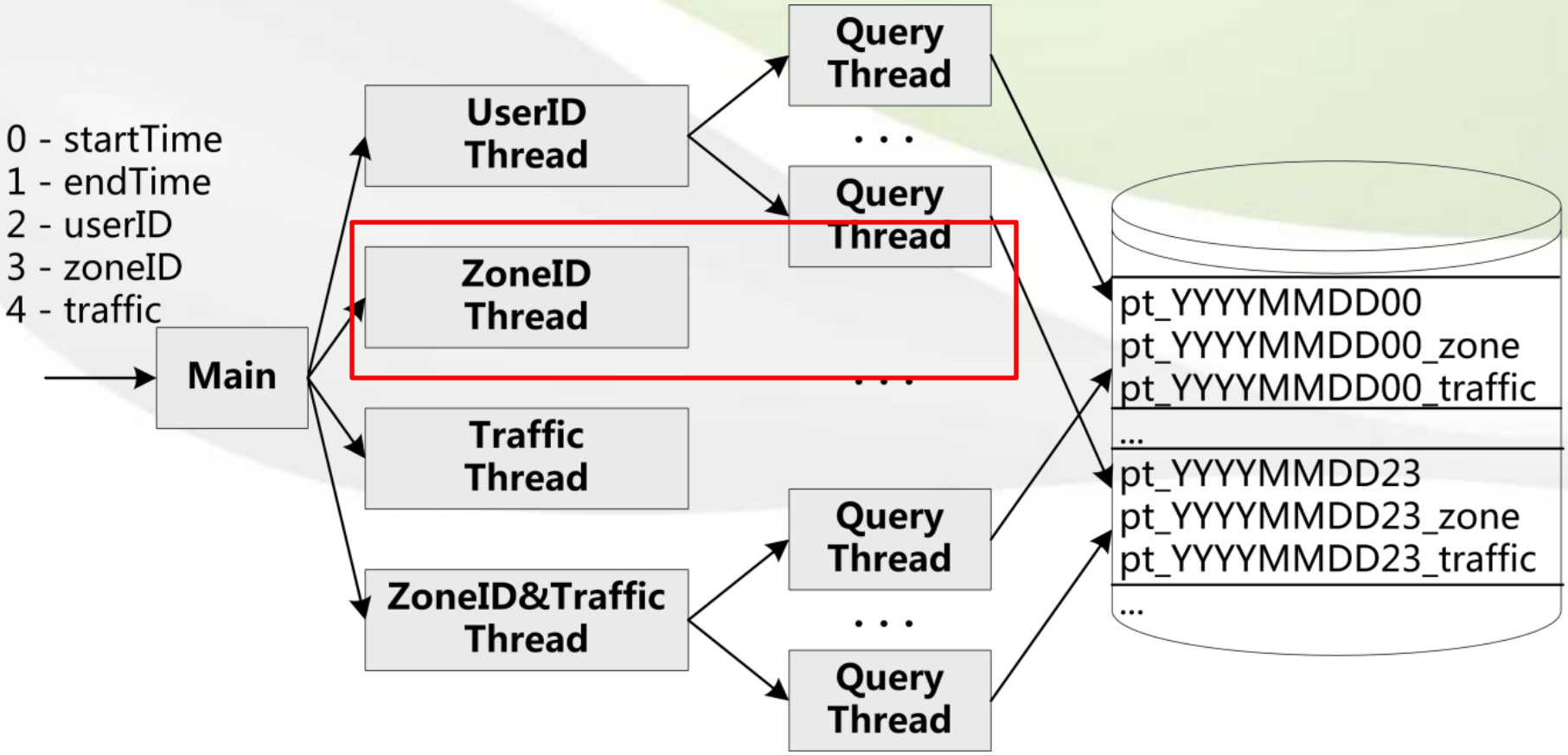
userID条件检索性能分析

结果记录数	首结果			全部结果		
	无索引单表	有索引多表（5分钟）		无索引单表	有索引多表（5分钟）	
		单线程	多线程		单线程	多线程
179	-	0.99	1.18	1.406	17.653	11.972
332	-	2.695	2.075	1.024	16.137	9.55
403	-	1.254	1.39	1.079	16.155	7.855
587	-	3.405	2.285	1.192	15.666	7.495
1003	-	1.075	1.409	1.162	16.178	6.619
1236	-	0.993	1.46	1.236	9.689	8.438
1396	-	0.929	1.231	1.111	8.629	8
1841	-	1.36	1.388	1.434	21.204	11.629
2136	-	0.905	1.146	1.43	10.602	7.94
59932	-	9.088	4.789	4.568	24.454	14.474





数据检索 - zoneID条件检索 QueryThread



zoneID条件检索 - QueryThread

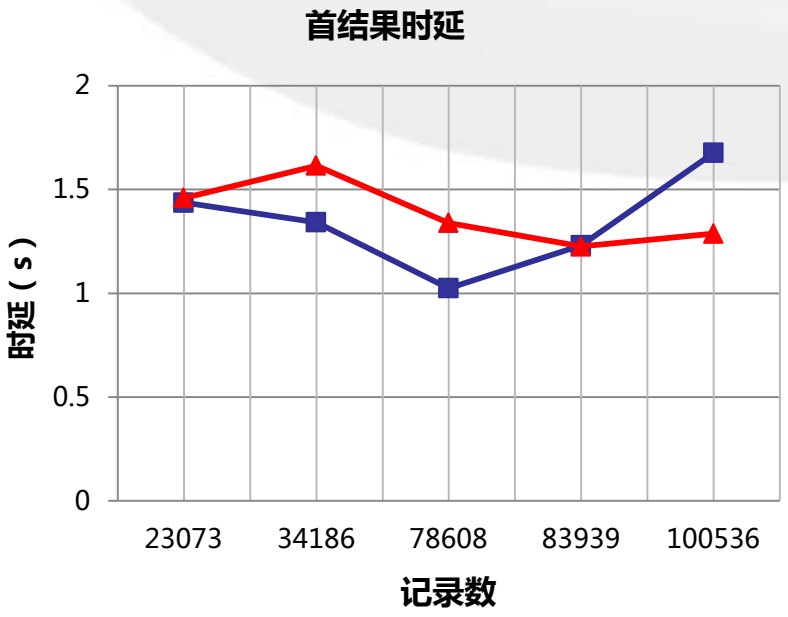
```
1:  public QueryThread(long time, String zoneID, HTablePool tablePool, CountDown signal) {
2:    tableName = "pt_" + time; // 主表名称
3:    indexTableName = tableName + "_zone"; // 索引表名称
4:    table = tablePool.getTable(tableName);
5:    indexTable = tablePool.getTable(indexTableName);
6:    this.zoneID = zoneID;
7:    this.signal = signal;
8:  }
9:  public void run() {
10:   Scan scan = new Scan(); // scan对象
11:   scan.addColumn("cf".getBytes(), "index".getBytes()); // index列
12:   scan.setCaching(500); // scan缓存
13:   scan.setCacheBlocks(false);
14:   scan.setStartRow(Bytes.toBytes(zoneID)); // 起始rowkey
15:   scan.setStopRow(Bytes.toBytes(zoneID + "9")); // 结束rowkey
16:   List<Get> getList = new ArrayList<Get>();
17:   ResultScanner rsIndexTable = indexTable.getScanner(scan); // scan结果
18:   for (Result r : rsIndexTable) {
19:     for (KeyValue kv : r.raw()) {
20:       byte[] index = kv.getValue();
21:       Get get = new Get(index);
```

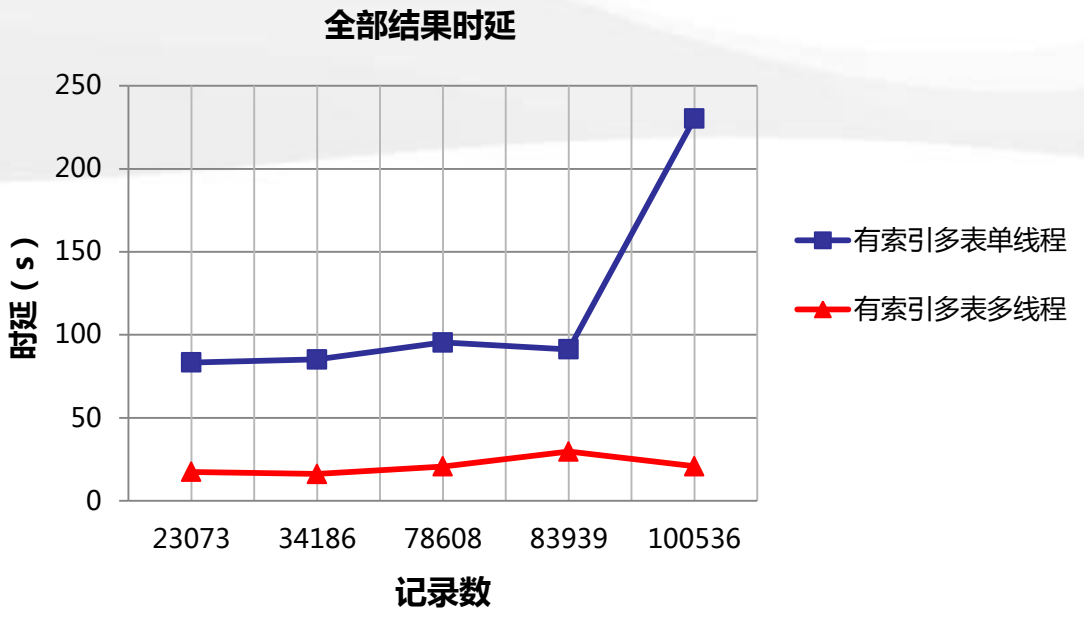
zoneID条件检索 - QueryThread（cont.）

```
22:       get.addColumn("cf".getBytes(), "content".getBytes());
23:       getList.add(get); // 放入get列表
24:       if (getList.size() == 1000) { // 每1000个提交一次
25:         Result[] results = table.get(getList); // 从主表检索索引对应的结果
26:         for (Result rs : results) {
27:           for (KeyValue kvs : rs.raw()) {
28:             resultQ.put(kvs.getValue()); // 放入结果队列
29:           }
30:         }
31:         getList.clear(); // 重置get列表
32:       }
33:     }
35:     if (!getList.isEmpty()) { // 执行剩余的get
36:       Result[] results = table.get(getList);
37:       for (Result r : results) {
38:       for (KeyValue kv : r.raw()) {
39:         resultQ.put(kv.getValue());  // 放入结果队列
40:       }
41:     }
42:   }
43:   signal.countDown(); // 更新完成线程计数器
44:   indexTable.close(); // 关闭索引表
45:   table.close(); // 关闭主表 }
```

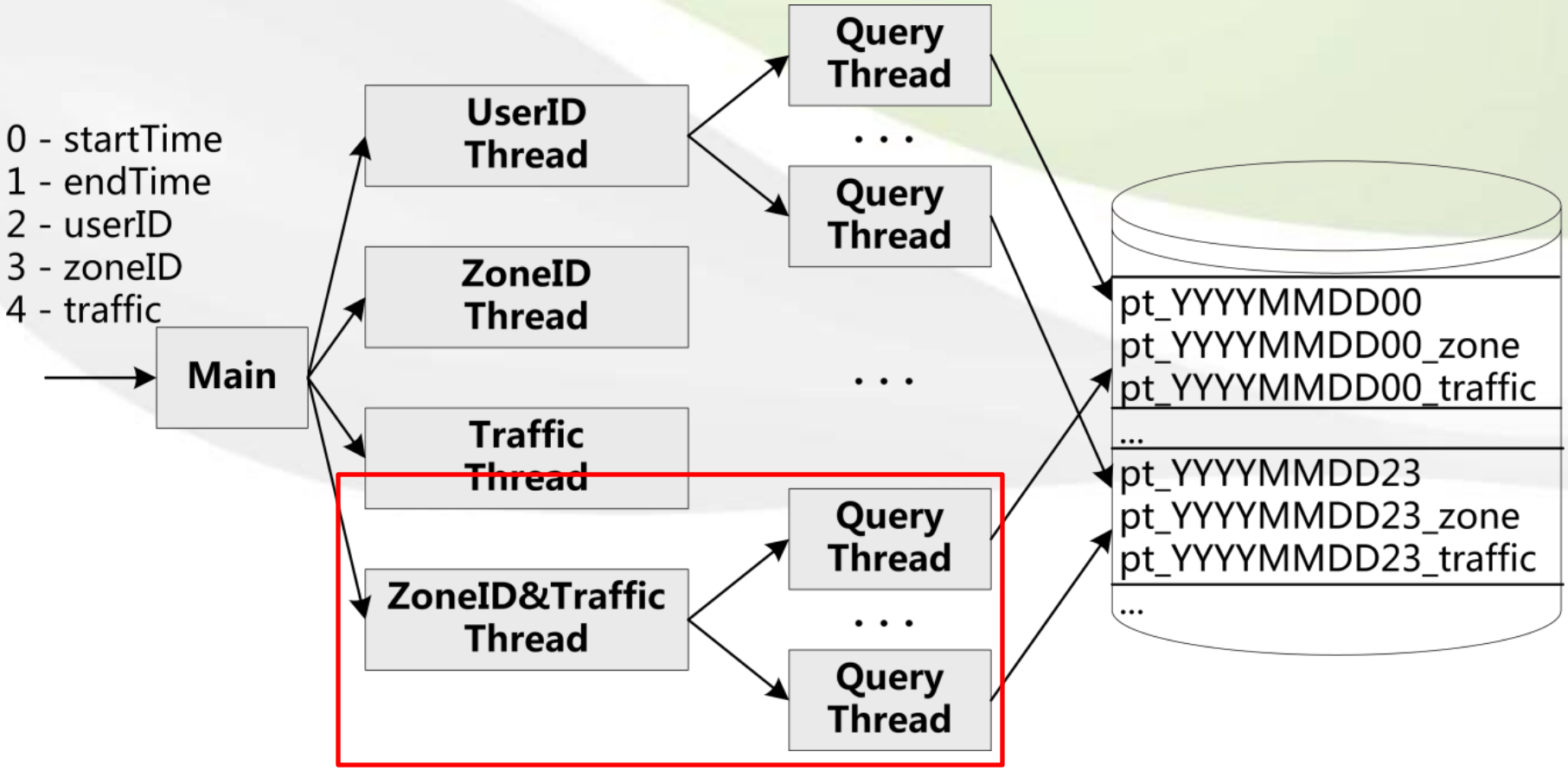
zoneID条件检索性能分析

结果记录数	首结果			全部结果		
	无索引单表	有索引多表		无索引单表	有索引多表	
		单线程	多线程		单线程	多线程
23073	-	1.437	1.459	-	83.338	17.342
34186	-	1.342	1.615	-	85.111	16.139
78608	-	1.024	1.339	-	95.343	20.656
83939	-	1.231	1.226	-	91.247	29.603
100536	-	1.677	1.287	-	230.364	20.712



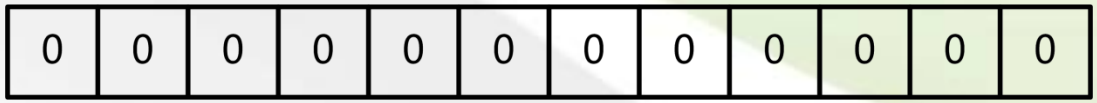


数据检索 - zoneID+traffic组合条件检索

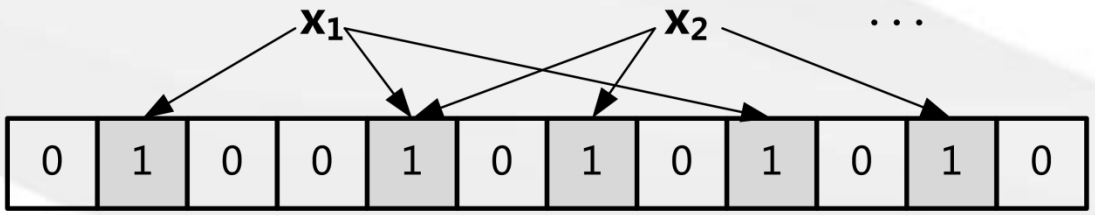


zoneID+traffic组合条件检索

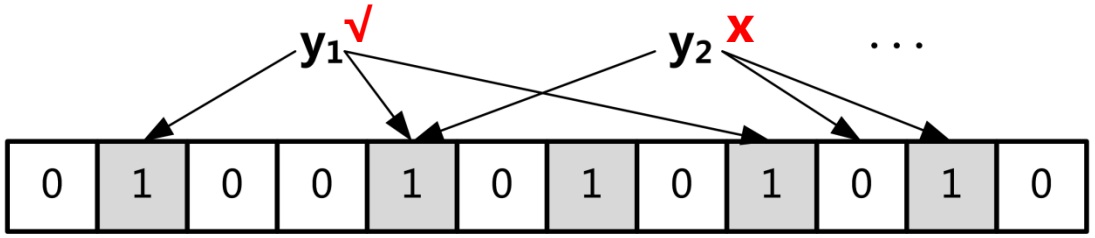
- Bloom Filter：高效数据过滤器
  - 初始状态



  - 放入集合1



  - 过滤集合2



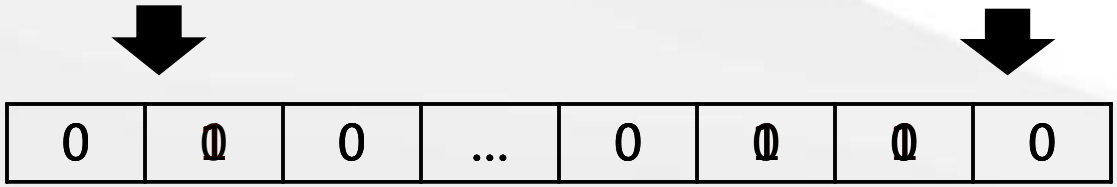
zoneID+traffic组合条件检索

小区索引表

zoneID + time	userID + time
21347_12315A	1111A
21347_12315B	2222B
…	…
21347_12315X	3333X
21347_12316N	4444N

流量索引表

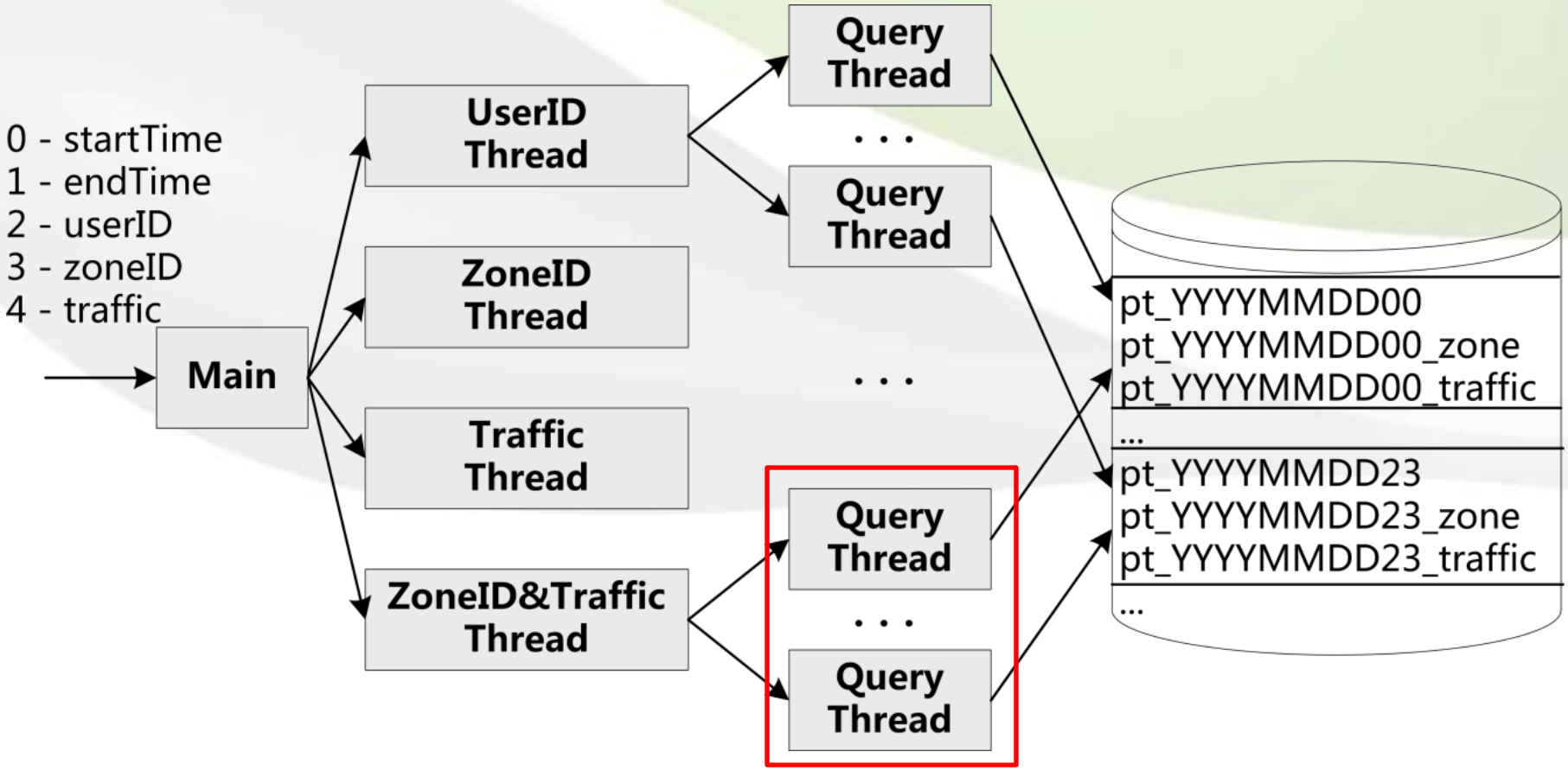
traffic + time	userID + time
000000000A	1111A
000000000X	3333B
…	…
000000001N	4444N
…	…



userID + time	content
1111A	…
…	…
3333X	…
…	…

主表

数据检索 - zoneID+traffic组合条件检索



zoneID+traffic组合条件检索 - QueryThread

```
1:  public QueryThread(long time,String traffic,String zoneID,HTablePool tablePool,CountDown signal){
2:    tableName = "pt_" + time;
3:    indexTableZoneName = tableName + "_zone";
4:    indexTableTrafficName = tableName + "_traffic";
5:    table = tablePool.getTable(tableName);
6:    indexTableZone = tablePool.getTable(indexTableZoneName);
7:    indexTableTraffic = tablePool.getTable(indexTableTrafficName);
8:    this.traffic = traffic;
9:    this.zoneID = zoneID;
10:   this.signal = signal;
11: }
12: public void run() {
13:   Scan scanZone = new Scan(); // 小区索引表scan对象
14:   scanZone.addColumn("cf".getBytes(), "index".getBytes()); // 列
15:   scanZone.setCaching(500); // scan缓存
16:   scanZone.setCacheBlocks(false);
17:   scanZone.setStartRow(Bytes.toBytes(zoneID)); // 起始rowkey
18:   scanZone.setStopRow(Bytes.toBytes(zoneID + "9")); // 结束rowkey
19:   ResultScanner rsZone = indexTableZone.getScanner(scanZone); // 小区索引表检索结果
```

zoneID+traffic组合条件检索 - QueryThread（cont.）

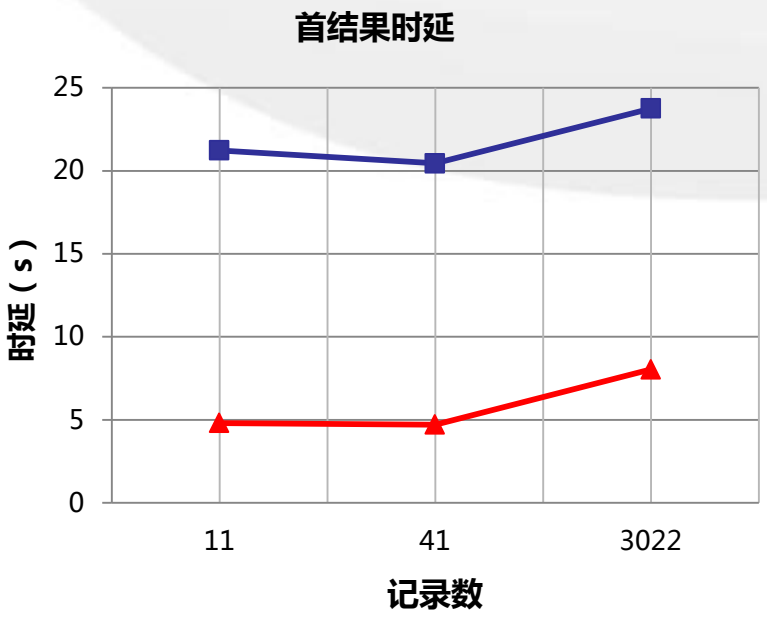
```
20:     Scan scanTraffic = new Scan(); // 流量索引表scan对象
21:     scanTraffic.addColumn("cf".getBytes(), "index".getBytes()); // 列
22:     scanTraffic.setCaching(500); // scan缓存
23:     scanTraffic.setCacheBlocks(false);
24:     scanTraffic.setStartRow(Bytes.toBytes(traffic)); // 起始rowkey
25:     scanTraffic.setStopRow(Bytes.toBytes(traffic + "9")); // 结束rowkey
26:     ResultScanner rsTraffic = indexTableTraffic.getScanner(scanTraffic); // 流量索引表检索结果
27:     List<Get> getList = mergeIndexForGet(rsZone, rsTraffic); // merge两个索引表结果，构建get表
28:     Result[] results = table.get(getList); // 提交get列表
29:     for (Result r : results) {
30:       for (KeyValue kv : r.raw()) {
31:         resultQ.put(kv.getValue()); // 保存结果
32:       }
33:     }
34:     signal.countDown();
35:     table.close(); // 关闭主表
36:     indexTableZone.close(); // 关闭小区索引表
37:     indexTableTraffic.close(); // 关闭流量索引表
38:   }
39: }
```

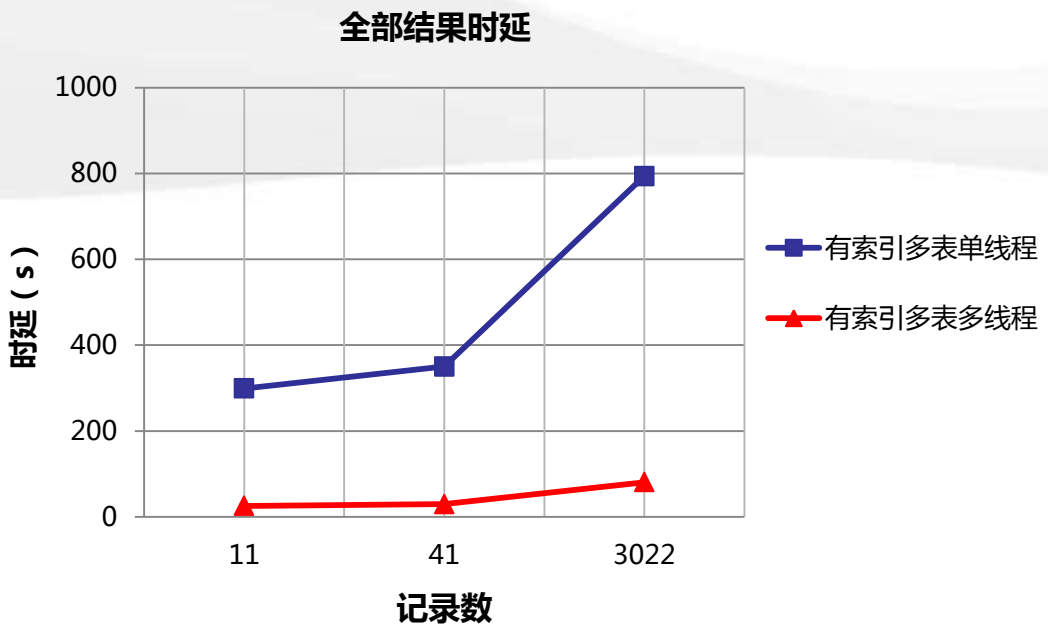
zoneID+traffic组合条件检索 - QueryThread（cont.）

```
40:  public List<Get> mergeIndexForGet(ResultScanner rsZone, ResultScanner rsTraffic) {
41:    List<Get> getList = new ArrayList<Get>();
42:    BloomFilter bf = new BloomFilter();
43:    for (Result r : rsZone) { // 将小区索引表检索结果放入bloom filter中
44:      for (KeyValue kv : r.raw()) {
45:        bf.add(kv.getValue());
46:      }
47:    }
48:    for (Result r : rsTraffic) { // 对流量索引表检索结果进行过滤
49:      for (KeyValue kv : r.raw()) {
50:        byte[] index = kv.getValue();
51:        if (bf.contains(index)) { // 在小区索引表中存在的，构造get请求
52:          Get get = new Get(index);
53:          get.addColumn("cf".getBytes(), "content".getBytes());
54:          getList.add(get);
55:        }
56:      }
57:    }
58:    return getList;
59:  }
```

zoneID+traffic组合条件检索性能分析

结果记录数	首结果			全部结果		
	无索引单表	有索引多表		无索引单表	有索引多表	
		单线程	多线程		单线程	多线程
11	-	21.229	4.802	-	299.128	25.319
41	-	20.453	4.706	-	350.234	29.416
3022	-	23.758	8.035	-	793.921	80.584





作业

- 本节问题
  - 将access.log（QQ群共享）文件采用Java多线程导入到HBase中，并实现rowkey条件检索、非rowkey条件检索和两个非rowkey条件联合检索
- 要求：
  - 将代码、查询结果截图，发送到 liujun@bupt.edu.cn
- 下节课程预告：
  - Hadoop高级数据分析工具