大数据编程模型和使用技巧

流式计算

陈一帅

yschen@bjtu.edu.cn

北京交通大学电子信息工程学院

内容

- 定义
- 原理
- 系统

流式计算

- Streaming
- Batch 分析 vs 实时或近实时分析
  - 实时或近实时分析变得越来越关键
  - 自动驾驶汽车，电网传感器
  - 社会网络上的热门话题和主题标签
- 分析来自无限制流的数据的活动，即数据流分析
  - 可追溯到 1990 年代在斯坦福，加州理工学院和剑桥等地进行的复杂事件处理的基础研究

内容

- 定义
- 原理
- 系统

节点预分析

- 数据流的分析有时需要靠近源
- 正在出现执行预分析的工具，其目的是确定应发送到云以进行更深入分析的数据子集
- 如 Apache Edgent 边缘分析工具，能够在 Raspberry Pi 等小型系统中运行

基本挑战

- 正确性和一致性
- 无限流中的数据在时间上不受限制
- 但如果要显示分析结果，则不能等到时间结束
- 因此，需要在合理的时间窗口结束时显示结果
- 如，可以在当天结束时，基于当天事件得到每日摘要
- 但是，如果您想更频繁地（例如每秒）获得结果怎么办?
- 挑战：因为处理分布式，如果时间间隔太短，可能无法收集整个系统的全局状态，某些事件可能会丢失或计数两次。在这种情况下，报告可能不一致。

四种时间窗口

- 固定时间窗口：将输入流分为逻辑段，每段对应于一个指定的处理时间间隔。间隔不重叠
- 滑动窗口：允许窗口重叠。例如，窗口大小为 10 秒，每 5 秒启动一次
- 以会话为单位的窗口：将流划分为与数据的某些键相关的活动的会话（Session）。例如，某位用户的一连串鼠标点击可以捆绑到一起，作为一个时间上临近的系列点击会话
- 全局窗口：封装整个有界流

触发

- 与窗口相关联
- 触发对窗口内容的分析，并发布结果

内容

- 定义
- 原理
- 系统

系统

- Spark Streaming
- Apache Storm, 来自 Twitter 的 Heron
- Apache Flink，来自德国 Stratosphere 项目
- Apache Beam，来自 Google 的 Cloud Dataflow，可以在 Flink, Spark, Google 云上运行
- Amazon Kinesis
- Azure Event Hubs
- IBM Stream Analytics

AWS Kinesis

- 来自亚马逊的事件流软件堆栈 Kinesis，包括三个服务
  - Kinesis Streams：提供有序的、可重播的实时流数据
  - Kinesis Firehose：支持极高规模的事件处理，可以将数据直接加载到 S3 或其他 Amazon 服务中
  - Kinesis Analytics：提供了基于 SQL 的分析工具，实时分析 Kinesis Streams 或 Firehose 中的流数据

分片

- 每个 Kinesis 流由一个或多个分片（Shard）组成
- 可以将一个流视为由许多股线组成的绳索
- 每条线都是一个分片（Shard），在流中移动的数据分布在组成该流的各个分片中
- 数据生产者向分片写入数据
- 消费者从分片读取数据

分片读写限制

- 写
  - 每个分片可以支持最高每秒 1000 条记录的写操作
  - 每秒最多可以写入 1 MB 数据
  - 但是，任何单个记录都不能大于 1 MB
- 读
  - 数据使用者每秒最多可以读五个事务（Transaction），总吞吐量 2 MB /秒
- 注意这是一个分片的限制
  - 你可以在一个流里包含数千个分片

和消息队列的区别

- Amazon Simple Queue Service（SQS）消息队列
- 工作机制
  - 消息（事件）生产者将项目添加到 SQS 队列
  - SQS 客户端可从队列中检索消息以进行处理
- 特点
  - 客户端不会删除队列中的消息。它们将保留一段时间（通常为 24 小时），或直到明确刷新整个队列为止
  - 队列中每个消息都有序列号。基于此号，客户端可用 API 一次获取该消息和所有后续消息（有最大数量限制）
  - 因此，客户端可随时重复对队列的分析，不同客户端也可以以相同或不同的方式处理相同的队列

发送温度传感器数据

- 创建一个 JSON 的温度记录（包括时间，温度）
- 将其转换为二进制数组

```
client = boto3.client('kinesis')
tz = pytz.timezone('America/Los_Angeles')
ts = datetime.datetime.now(tz)
item = {'id': 'sensor 1', 'val': 73, 'label': 'temperature',
        'localtime': str(ts)}
data = json.dumps(item)
client.put_record(
            StreamName='cbookstream2',
            Data= bytearray(data),
            PartitionKey = 'a'
        )
```

发送温度传感器数据

- 通过分区键标识分片
  - 因为只有一个分片，所以用字符串'a'
- 该键值将会被散列化为一个整数，然后被用来选择分片
  - 如果只有一个分片，则所有记录都将映射到分片 0

```
client = boto3.client('kinesis')
tz = pytz.timezone('America/Los_Angeles')
ts = datetime.datetime.now(tz)
item = {'id': 'sensor 1', 'val': 73, 'label': 'temperature',
        'localtime': str(ts)}
data = json.dumps(item)
client.put_record(
            StreamName='cbookstream2',
            Data= bytearray(data),
            PartitionKey = 'a'
        )
```

读取流

- 先创建 Iterator
  - 每个被写入分片的记录都有一个序列号
  - 要读取这些记录，需要提供一个分片的迭代器
- 三种方法
  - 指定一个时间戳，读取在指定时间之后到达的记录
  - 将迭代器定位到流的最新点，获取该点之后的新记录
  - 根据分片号和序列号创建一个迭代器

读取流

- 第三种方法
  - 根据分片号（ShardId）和序列号（SequenceNumber）创建迭代器
  - 可通过调用 describe_stream(StreamName)函数获得分片及其起始序列号

```
client = boto3.client('kinesis')
iter = client.get_shard_iterator(
       StreamName='cbookstream2',
       ShardId='shardID',
       ShardIteratorType = 'AT_SEQUENCE_NUMBER',
       StartingSequenceNumber='seqno'
       )
```

获取记录

- 循环 get_records（）函数返回记录列表和元数据
  - 返回"下一个分片的迭代器"，用于获取下一批记录
- 限制
  - 一次最多返回 10 MB，每秒最多返回 2 MB
  - 可以限制返回的记录数，避免 10 MB 的限制
  - 如接近极限，最好添加新分片，或使用拆分分片功能
- 读后分析数据（analyzeData）

```
iterator = iter['ShardIterator']
while True:
    time.sleep(5.0)
    resp = client.get_records(ShardIterator=iterator)
    iterator = resp['NextShardIterator']
    analyzeData(resp['Records'])
```

分析数据

- 对读出来的记录，逐个解析
  - 每个记录都包括近似到达时间
- 得到 JSON 格式的记录数据，里面包括
  - 记录的本地时间

```
def analyzeData(resp):
    #resp is the response['Records'] field
    for rec in resp:
        data = rec['Data']
        arrivetime = rec['ApproximateArrivalTimestamp']
        print('Arrival time = '+str(arrivetime))
        item = json.loads(data)
        prints('Local time = ' + str(parse(item['localtime'])))
        delay = arrivetime - parse(item['localtime'])
        secs = delay.total_seconds()
        print('Message delay to stream = '+str(secs) + ' seconds')
```

Firehose Batch Stream 转存

- 和 SQS 不同，Kinesis Firehose 处理自动传输到 S3 或 Amazon Redshift 的大量数据流
- 它是面向批处理的：它将传入的数据缓存到最大 128 MB 的缓冲区中，并按照你指定的特定间隔（从每分钟到每 15 分钟）将缓冲区转储到 S3。你还可以指定数据被压缩和/或加密
- 因此，Firehose 并非为实时分析而设计，而是为近实时的大规模分析而设计

Spark Streaming

- 基于 Spark
- 高级库，旨在处理大多数云上的流数据
- 原理
  - 利用 Spark Core 的快速调度功能，按窗口获取流数据，将其转换为一种特殊的 RDD：Dstream
  - Dstream 进入流处理引擎，该引擎可以遵循 MapReduce 或任何 DAG 模型，完成计算
- 优点
  - 基于 RDD 的设计使为批处理分析编写的同一套应用程序代码可以在流分析中使用

Streaming 支持的 Transformation 操作

- Map
- Filter
- Repartition
- Union
- Reduce
- Join
- Transform
  - 通过对源 DStream 的每个 RDD 应用 RDD-to-RDD 函数来返回新的 DStream。这可用于在 DStream 上执行任意的 RDD 操作

部署

- 数据源
  - 可以部署为从 HDFS，Flume，Kafka，Twitter 和 ZeroMQ 源读取数据
- 可以应用于大型集群或单个引擎
  - 在大型集群的生产模式下，可使用 ZooKeeper 和 HDFS 来实现高可用性

部署

- 首先部署集群管理器，对其进行识别，分配资源
- 打包应用程序，将程序编译为 JAR
  - 如果程序使用高级资源（例如，Kafka，Flume，Twitter），程序须链接到它们
  - 比如使用 TwitterUtils 的程序必须包含 spark-streaming-twitter_2.10 及其依赖项
- 为程序配置足够内存以容纳接收到的数据。例如，如果要执行 10 分钟窗口操作，则必须至少将最后 10 分钟数据保留在内存中

部署：容错

- 部署检查点
  - 将 Hadoop API 兼容的容错存储中的目录（例如 HDFS，S3 等）配置为检查点目录。检查点信息可用于故障恢复。
- 配置重启
  - 配置程序驱动程序的自动重启，程序须监视驱动程序进程，在驱动程序失败时重新启动驱动程序
- 预写日志
  - 将接收的所有数据写入检查点目录中的预写日志，可以防止驱动程序恢复时丢失数据，从而确保零数据丢失。但会以单个接收器的接收吞吐量为代价。可通过并行运行更多接收器以提高总吞吐量来纠正此问题
  - 启用预写日志后，可以禁用 Spark 收到的数据的复制，因为该日志已经存储在复制的存储系统中

示例

以下代码用于计算滑动窗口上的推文：

```
TwitterUtils.createStream(…)
    .filter(_.getText.contains(“Spark”))
    .countByWindow(Seconds(5))
```

Kafka

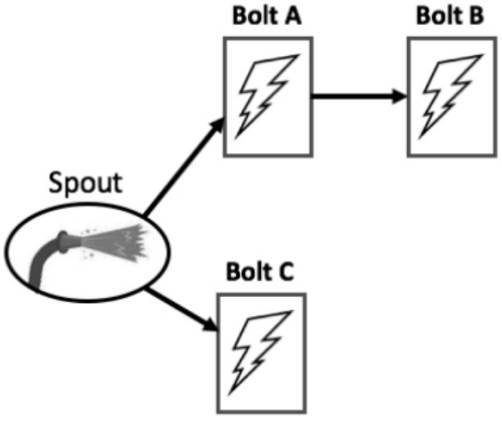
- 包含发布-订阅（Pub-Sub）消息传递和数据流处理的开源消息系统
- 在服务器群集上运行，高度可扩展性
- 其中的流是记录流，记录被分为多个主题
- 每个记录有一个键、一个值和一个时间戳
- 流可以很简单
  - 一个单流客户端使用一个或多个主题的事件
- 也可以很复杂
  - 基于组织成图的生产者和消费者的集合，称为拓扑

Storm 和 Heron Streams

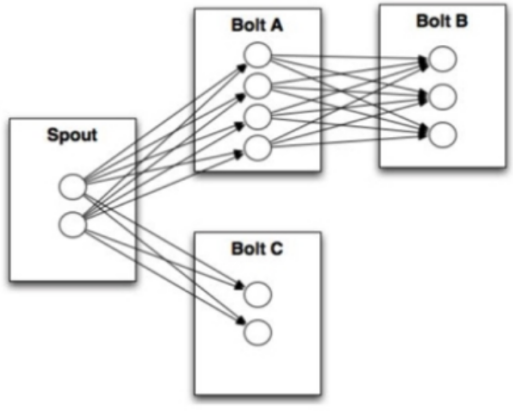
- 执行有向图的任务
- 由 Nathan Marz 创建，2011 年 Twitter 开源
- 用 Lisp 的一个变种 Clojure 编写，可在 Java 虚拟机上运行
- 2015 年，Twitter 重新编写了 Storm，创建了 API 兼容的 Heron Streams

Storm 拓扑

- DAG 有向无环图
  - 程序定义的抽象拓扑（左）
  - 运行时执行的展开并行拓扑（右）
  - 数据源节点：Spout（喷嘴，数据源）
  - 数据转换和处理节点：Bolt（螺栓）



Abstract Topology



Parallelized Topology

Storm 编程模型

- 执行有向图的任务
- Classic 和 Trident
- 扩展基本的 Spout 和 Bolt 类，使用拓扑生成器将所有内容绑定在一起
- 三种方法：prepare()，execute()和 clarifyOutputFields()

Google Dataflow and Apache Beam

- Google Cloud Dataflow 系统的开源版本
- 各种最新数据流分析解决方案的常用入口
- 目标是是将批处理和流处理进行统一
- 各种 Trigger（触发器）

Beam watermark（水印）

- 由 Google Dataflow 引入的概念
- 基于事件时间。当系统估计它已在给定窗口中看到所有数据时就用它来发送结果
- 基于指定水印的不同方法，你可以使用多种方法来定义触发器
- 建议参考 Google Dataflow 文档

Beam vs Spark Streaming

- 原生
  - Spark 是一个批处理系统，流模式是在此基础上附加的
  - Beam 是为了数据流处理从头开始设计的，有批处理能力
- 窗口定义
  - Spark 窗口基于 DStream 中的 RDD，不如 Beam 窗口灵活
- 乱序事件处理
  - 乱序情况下，事件发生时间和处理时间并不相同，这在分布式处理时很常见
  - Beam 能够对此处理。它对事件时间窗口、触发器和水印的介绍是它的主要贡献。因此它可以在乱序情况下仍然及时地生成近似结果

Apache Flink

- 和 Beam 中存在许多相同的核心概念

系统小结

- 系统
  - Kinesis
  - Spark Streaming
  - Azure 流分析
  - Storm / Heron
  - Google Dataflow / Beam
  - Flink
- 共享一些相同的概念，以相似方式创建流水线（Pipeline）
- Storm / Heron 构造图，其他使用函数风格的流水线构造

小结

- 定义
- 原理
- 系统

练习

- Spark Streaming
- AWS Streaming