SL01 :슬라이드 : 8.6 迟到数据处理

今天我们要学习的是 Spark Structured Streaming (结构化流处理) 中一个非常重要的概念——迟到数据处理 (Late Data Processing)。

大家请看这张幻灯片，我们今天的内容主要分为五个部分：

- 首先我们会讲事件时间 (Event Time) 的概念
- 然后理解什么是迟到数据 (Late Data)
- 接着学习水印 (Watermark) 机制
- 再了解多水印规则 (Multiple Watermark Rules)
- 最后通过一个完整的实例来看迟到数据是如何处理的

这些概念在实时流处理 (Real-time Stream Processing) 中非常关键，

因为在真实的分布式系统 (Distributed System) 中，

数据不可能总是按时到达。

SL02 :슬라이드 : 8.6.1 事件时间

好，我们先来看第一个概念——事件时间（Event Time）。

假设我们有一个数据流（Data Stream），

每条数据都包含了数据生成的时间戳（Timestamp）。

我们的任务是计算每10分钟内接收到的单词个数，

并且每5分钟更新一次结果。

大家看这个图，我来给大家详细解释一下：

上面是输入数据流（Input Data Stream），你们可以看到：

- 在12:02时刻，我们收到了"cat dog"
- 在12:03时刻，收到了"dog dog"
- 在12:07时刻，收到了"owl cat"
- 在12:11时刻，收到了"dog"
- 在12:13时刻，收到了"owl"

下面的时间轴(**zhóu**)显示的是处理时间（Processing Time），

从12:00开始，每5分钟触发一次。

关键在于我们使用的是滑动窗口（**huádòng chuāngkǒu** Sliding Window）：

- 窗口大小（Window Size）是10分钟
- 滑动间隔（Sliding Interval）是5分钟

所以你们可以看到，每次触发时，

我们都会计算多个重叠(**chóngdié**)的时间窗口内的词频(**cípín**)统计。

这就是事件时间处理的基本概念。

SL03 :슬라이드：8.6.2 迟到数据

现在我们来看一个更复杂的情况——如果数据迟到了怎么办？

在现实的分布式系统中，

由于网络延迟（Network Latency)、

系统故障（System Failure）等各种原因，

数据可能不会按照事件发生的时间顺序到达。

大家看这个图，我来解释一下发生了什么：

正常情况下：

- 前两条数据在12:00-12:05时间窗口内到达
- 一条数据在12:05-12:10时间窗口内到达
- 两条数据在12:10-12:15时间窗口内到达

但是注意这里有一个问题！

有一条数据，它的事件时间（Event Time）是12:04，

但是直到12:11才到达系统。

这就是我们说的迟到数据（Late Data)。

你们可以看到，由于这条迟到数据的到达，

12:00-12:10时间窗口内的dog的计数从3变成了4。

这说明迟到数据确实影响了我们的计算结果。

这就引出了一个重要问题：我们应该如何处理这些迟到的数据？

SL04 :슬라이드 : 8.6.3 水印（代码部分）

为了解决迟到数据的问题，

Spark Structured Streaming 引入了水印 (Watermark) 机制。(왜 워터마크인가? 데이터가 물흐름이라고 하니까, 물위에 마크가 있다는 표시로서 해석, （为什么叫"水印"？因为数据被比喻为水流，所以可以将其理解为数据流上的标记。）

大家看这段代码，这是定义水印的标准方法：

```
words = ... # DataFrame流的结构为 { timestamp: Timestamp, word: String }

# 对窗口和字母进行聚合(jùhé)并计算计数
windowedCounts = words \
    .withWatermark("timestamp", "10 minutes") \
    .groupBy(
        window(words.timestamp, "10 minutes", "5 minutes"),
        words.word) \
    .count()
```

关键在于这一行：.withWatermark("timestamp", "10 minutes")

这里我们告诉 Spark：

- 在 timestamp 字段上设置水印
- 允许数据最多迟到10分钟

这意味着，如果一个事件的时间比当前水印时间早超过10分钟，

那么这个事件就会被丢弃(diūqì)。

SL05 :슬라이드 : 8.6.3 水印（图解）

现在我们通过这个图来理解水印是如何工作的。

这个图很重要，我来逐个(**zhú gè)**解释这些符号：

- 空心圆：正常到达的数据
- 黑色实心圆点 ：成为最大事件时间的数据
- 三角形：迟到到达但在水印范围内的数据
- **五边形** (wǔbiānxíng)：迟到并且超出水印范围的数据

实线表示水印（Watermark)，

虚线(**xùxiàn)**表示到目前为止能看到的最大事件时间。

水印的计算公式是：当前水印 = 最大事件时间 - 允许迟到时间

比如说，如果我们看到的最大事件时间是12:21，允许迟到10分钟，那么当前水印就是12:11。

你们可以看到，处理时间每5分钟触发一次，从12:05到12:25。

每次触发时，Spark 会：

1. 更新水印
2. 丢弃超出水印范围的迟到数据
3. 处理在水印范围内的所有数据
4. 输出更新的结果

这样既(**jì)**能处理一定程度的迟到数据，又能保证系统性能不会因为无限期等待而下降。

SL06 은 앞의 장을 칼라로 표현한 것임.

적당히 알아서 설명할 것

SL08 :슬라이드 : 8.6.4 多水印规则

在实际应用中，我们经常需要处理来自多个数据源的流。

这就涉及到多水印规则 (Multiple Watermark Rules)。

让我来解释一下这个复杂的场景：

当一个查询涉及多个输入源的联合 (Union) 或连接 (Join) 时：

1. 独**立水印**：每个输入源都可以定义自己独立的水印来跟踪中间状态。

   单个输入源定义的水印不会影响其他输入源。

2. 独**立跟踪**：Structured Streaming 会独立跟踪每个数据流中的事件时间，并分别计算延迟。

3. **全局水印选择**：如果联合或连接查询存在有状态的操作 (Stateful Operations)，Structured Streaming 会选择一个全局水印来跟踪所有流计算的中间状态。

4. **保守策略**：默认情况下，为了保证所有流的迟到数据都会被处理，Structured Streaming 会选择最长的迟到水印时间。

举个例子：如果流A允许数据迟到10分钟，流B允许数据迟到20分钟，那么全局水印就会允许保留20分钟的中间状态。

这样的设计确保了即使在多流场景下，

我们也不会丢失重要的迟到数据。

SL09 :슬라이드：8.6.5 处理迟到数据的实例（导入模块）

好，现在我们通过一个完整的实例来看如何处理迟到数据。

这是一个 Python 程序，导入需要用到的模块

```
# 导入需要用到的模块
import os
import shutil
from functools import partial

from pyspark.sql import SparkSession
from pyspark.sql.functions import window
from pyspark.sql.types import StructType, StructField
from pyspark.sql.types import TimestampType, StringType
```

这里我们导入了：

- os 和 shutil：用于文件系统操作
- SparkSession：创建 Spark 会话
- window 函数：用于定义时间窗口
- 数据类型定义：StructType, StructField, TimestampType, StringType

这些都是构建流处理应用程序的基础组件。

SL10 :슬라이드：8.6.5 处理迟到数据的实例（环境设置）

接下来我们看环境设置的代码：

```
# 定义CSV文件的路径常量
TEST_DATA_DIR = '/tmp/testdata/'
TEST_DATA_DIR_SPARK = 'file:///tmp/testdata/'

# 搭建测试环境
def test_setUp():
    if os.path.exists(TEST_DATA_DIR):
        shutil.rmtree(TEST_DATA_DIR, ignore_errors=True)
    os.mkdir(TEST_DATA_DIR)

# 恢复测试环境
def test_tearDown():
    if os.path.exists(TEST_DATA_DIR):
        shutil.rmtree(TEST_DATA_DIR, ignore_errors=True)
```

这些函数的作用是：

- test_setUp()：清理并创建测试目录
- test_tearDown()：清理测试环境

注意(**zhùyì**)这里有两个路径：

一个是本地文件系统路径，

一个是 Spark 使用的 file:// 协议路径。

这在分布式环境中是很常见的做法。

SL11 :슬라이드：8.6.5 处理迟到数据的实例（文件写入函数）

这里我们定义了一个模拟数据输入的函数：

```
# 编写模拟输入的函数，传入CSV文件名和数据。注意写入应当是原子性的
# 如果写入时间较长，应当先写入临时文件再移动到CSV目录内
# 这里采取直接写入的方式
def write_to_csv(filename, data):
    with open(TEST_DATA_DIR + filename, "wt", encoding="utf-8") as f:
        f.write(data)
```

这个函数很简单但很重要。

在实际的流处理系统中，我们需要模拟数据的实时到达。

注释中提到了一个重要概念：**原子性写入**（Atomic Write)。

在生产环境中，如果写入时间较长，我们应该：

1. 先写入临时文件
2. 写入完成后再移动到目标目录

这样可以避免 Spark 读取到不完整的文件。

SL12 :슬라이드：8.6.5 处理迟到数据的实例（主函数开始）

现在我们进入主函数：

```
if __name__ == "__main__":
    test_setUp()

    # 定义模式，该模式为字符串类型的word和时间戳类型的eventTime两个列组成
    schema = StructType([
        StructField("word", StringType(), True),
        StructField("eventTime", TimestampType(), True)])

    spark = SparkSession ₩
        .builder ₩
        .appName("StructuredNetworkWordCountWindowedDelay") ₩
        .getOrCreate()

    spark.sparkContext.setLogLevel('WARN')
```

这里我们：

1. 调用 test_setUp() 初始化环境
2. 定义了数据的 Schema：包含 word（字符串）和 eventTime（时间戳）两个字段
3. 创建 SparkSession
4. 设置日志级别为 WARN，这样可以减少控制台输出

Schema 定义很重要，因为 Structured Streaming 需要知道数据的结构才能正确处理。

SL13 :슬라이드：8.6.5 处理迟到数据的实例（流定义）

现在我们定义数据流和窗口操作：

```
lines = spark ₩
        .readStream ₩
        .format('csv') ₩
        .schema(schema) ₩
        .option("sep", ";") ₩
        .option("header", "false") ₩
        .load(TEST_DATA_DIR_SPARK)

    # 定义窗口
    windowDuration = '1 hour'

    windowedCounts = lines ₩
        .withWatermark("eventTime", "1 hour") ₩
        .groupBy('word', window('eventTime', windowDuration)) ₩
        .count()
```

这里发生了几个关键步骤：

1. **创建流数据源**：从 CSV 文件读取数据，使用分号作为分隔符
2. **设置水印**：允许数据最多迟到1小时
3. **定义窗口操作**：按照1小时的窗口对单词进行计数

注意这里的水印设置：withWatermark("eventTime", "1 hour")，

这意味着超过1小时的迟到数据将被丢弃。

SL14 :슬라이드：8.6.5 处理迟到数据的实例（查询启动）

现在我们启动流查询并写入第一个测试文件：

```
query = windowedCounts ₩
        .writeStream ₩
        .outputMode("update") ₩
        .format("console") ₩
        .option('truncate', 'false') ₩
        .trigger(processingTime="8 seconds") ₩
        .start()

    # 写入测试文件file1.csv
    write_to_csv('file1.csv', """
正常;2023-10-01 08:00:00
正常;2023-10-01 08:10:00
正常;2023-10-01 08:15:00
正常;2023-10-01 08:20:00
""")
```

这里我们：

1. 启动流查询，使用 update 输出模式
2. 每8秒触发一次处理
3. 写入第一批测试数据

第一批数据都是"正常"到达的，时间分别是08:00, 08:10, 08:20。

SL15 :슬라이드：8.6.5 处理迟到数据的实例（第一次处理）

接下来我们处理第一批数据并添加第二批：

```
# 处理当前数据
    query.processAllAvailable()

    # 此时事件时间更新到上次看到的最大的2023-10-01 08:20:00

    write_to_csv('file2.csv', """
正常;2023-10-01 20:10:00
正常;2023-10-01 19:00:00
一小时以内延迟到达;2023-10-01 10:00:00
一小时以内延迟到达;2023-10-01 10:50:00
""")
```

这里很有意思：

1. 第一条数据时间是20:00:00，这会更新我们的最大事件时间
2. 后面两条数据的事件时间是10:00:00和10:50:00，虽然迟到了，但是在1小时的水印范围内
3. 当前水印 = 20:00:00 - 1小时 = 19:00:00，所以这些迟到数据仍然会被处理

SL16 :슬라이드：8.6.5 处理迟到数据的实例（第二次处理）

继续处理并添加第三批数据：

```
    # 处理当前数据
    query.processAllAvailable()

    # 此时事件时间更新到上次看到的最大的2023-10-01 20:00:00

    write_to_csv('file3.csv', """
正常;2023-10-01 20:20:00
正常;2023-10-01 20:30:00
正常;2023-10-01 19:20:00
一小时以外延迟到达;2023-10-01 10:00:00
一小时以外延迟到达;2023-10-01 10:50:00
一小时以内延迟到达;2023-10-01 19:00:00
一小时以内延迟到达;2023-10-01 19:30:00
""")
```

第三批数据更加有趣：

1. 时间为20:00:00的数据是正常的
2. 时间为10:00:00和10:50:00的数据超出了水印范围（早于19:00:00），会被丢弃
3. 时间为19:00:00的数据恰好在水印边界上，会被处理

这完美展示了水印机制的工作原理！

SL17 :슬라이드：8.6.5 处理迟到数据的实例（完成）

最后我们完成处理并清理环境：

```
# 处理当前数据
    query.processAllAvailable()

    query.stop()

    test_tearDown(
```

这个完整的例子展示了：

1. 如何设置水印
2. 在水印范围内的迟到数据如何被正确处理
3. 超出水印范围的数据如何被丢弃
4. 流查询的完整生命周期