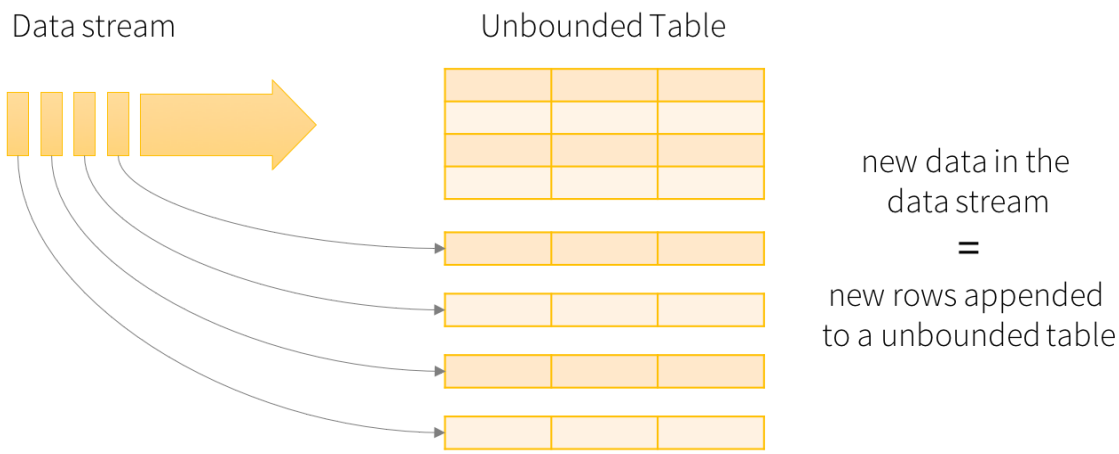
# 1，structured streaming介绍

## 1.1 基本概念

Structured Streaming 是一个可扩展，可容错，基于Spark SQL执行引擎的流处理引擎。使得流处理的逻辑实现和静态数据的批处理的逻辑实现是一致的。随着实时流数据的到来，Spark SQL引擎会连续处理数据并且更新结果到最终的Table中。可以在Spark SQL上引擎上使用DataSet/DataFrame API处理流数据的聚集，事件时间窗口，和流与批次的连接操作等。Structured Streaming 可以实现端到端的exactly once语义，以及通过检查点机制和WAL机制实现对容错的支持。

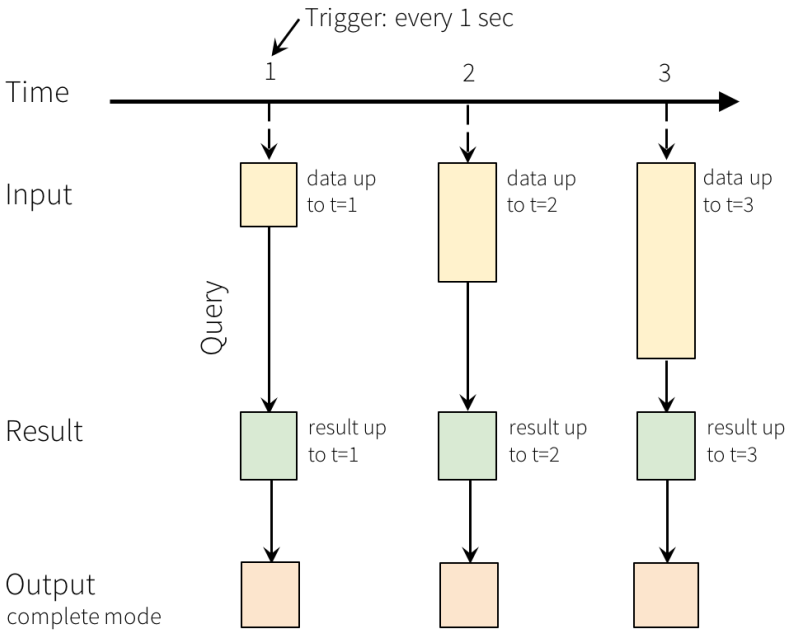
**编程模型**

结构化流的关键思想是将实时数据流视为一个连续附加的表，即把输入的数据流看做一个表，持续到来的数据作为新的行不断追加到该表中。




Data stream as an unbounded table

对输入流的查询操作，会生成一个结果表，每一次触发的时间段内，新的数据行追加到表中，最终在结果中更新



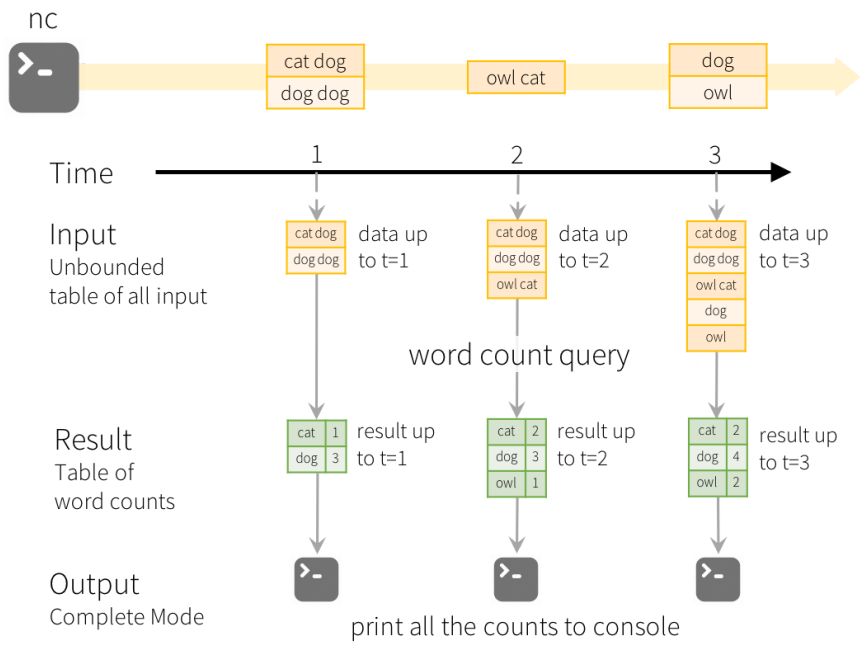

Programming Model for Structured Streaming

## 1.2 Output模式

Output是写入到外部存储的写方式，写入方式有不同的模式：

- Complete模式： 将整个更新表写入到外部存储，写入整个表的方式由存储连接器决定。
- Append模式：只有自上次触发后在结果表中附加的新行将被写入外部存储器。这仅适用于结果表中的现有行不会更改的查询。
- Update模式：只有自上次触发后在结果表中更新的行将被写入外部存储器（在Spark 2.0中尚不可用）。注意，这与完全模式不同，因为此模式不输出未更改的行。

  Complete模式下，下沉到控制台的示意图如下：




Model of the Quick Example

# 2，内置输入源

## 2.1 Structured内置的输入源 Source

| Source | Options | Fault-tolerant | Notes |
| --- | --- | --- | --- |
| File Source | maxFilesPerTrigger：每个触发器中要考虑的最大新文件数（默认值：无最大值）latestFirst：是否先处理最新的新文件，当存在大量积压的文件时有用（默认值：false） fileNameOnly：是否基于以下方法检查新文件只有文件名而不是完整路径（默认值：false）。 | 支持容错 | 支持glob路径，但不支持以口号分割的多个路径 |
| Socket Source | host：要连接的主机，必须指定 port：要连接的端口，必须指定 | 不支持容错 | |
| Rate Source | rowsPerSecond（例如100，默认值：1）：每秒应生成多少行。rampUpTime（例如5s，默认值：0s）：在生成速度变为之前加速多长时间rowsPerSecond。使用比秒更精细的粒度将被截断为整数秒。numPartitions（例如10，默认值：Spark的默认并行性）：生成的行的分区号 | 支持容错 | |
| Kafka Source | http://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html | 支持容错 | |

### 2.1.2 File Source

将目录中写入的文件作为数据流读取。支持的文件格式为：text、csv、json、orc、parquet

```
def main(args: Array[String]): Unit = {

    // 创建Spark程序入口
    val sparkSession = SparkSession
      .builder()
      .appName("StructuredNetworkWordCount")
      .master("local[*]")
      .getOrCreate()


   // val frame: DataFrame = lines.toDF("id","name","age")
    val userSchema = new StructType().add("name", "string").add("age", "integer")
    val source = sparkSession
      .readStream
      // Schema must be specified when creating a streaming source DataFrame.
      .schema(userSchema)
      // 每个trigger最大文件数量
      .option("maxFilesPerTrigger",100)
      // 是否首先计算最新的文件，默认为false
      .option("latestFirst",value = true)

      // 是否值检查名字，如果名字相同，则不视为更新，默认为false
```

```
      .option("fileNameOnly",value = true)
        .json("jsonres")

    val query: StreamingQuery =
source.writeStream.outputMode("append").format("console").start()
    // 执行
    query.awaitTermination()

  }
```

### 2.1.3 Socket Source

从Socket中读取UTF8文本数据。一般用于测试，使用nc -lc 端口号 向Socket监听的端口发送数据。

```
val lines = spark.readStream
  .format("socket")
  .option("host", "localhost")
  .option("port", 9090)
  .load()
```

### 2.1.4 Rate Source

以每秒指定的行数生成数据，每个输出行包含一个`timestamp`和`value`。其中`timestamp`是一个`Timestamp`含有信息分配的时间类型，并且`value`是`Long`包含消息的计数从0开始作为第一行类型。此源用于测试和基准测试。

```
    val rate = spark.readStream
      .format("rate")
      // 每秒生成的行数，默认值为1
      .option("rowsPerSecond", 10)
      .option("numPartitions", 10)
      .option("rampUpTime",0)
      .option("rampUpTime",5)
      .load()
```

### 2.1.5 Kafka Source

```
val df = spark
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
  .option("subscribePattern", "topic.*")
  .load()
```

## 2.2 socket数据源Wordcount

```
object StructuredStreamingDemo {

    def main(args: Array[String]): Unit = {
```

```
    // 创建Spark程序入口
    val sparkSession = SparkSession
      .builder()
      .appName("StructuredNetworkWordCount")
      .master("local[*]")
      .getOrCreate()

    import sparkSession.implicits._

    // 创建监听 localhost:9999 的DataFrame流
    val lines = sparkSession.readStream
      .format("socket")
      .option("host", "node1")
      .option("port", "9999")
      .load()

    // 将行数据分割成单词
    /**
      * lines DataFrame代表一个包含流文本数据的无界表,这个表只有一列数据，列名为“value”。
      * 流文本数据中的每一行都会成为表的一行。
      * 为了使用 flatMap函数，我们使用.as[String]方法将DataFrame转换为DataSet[String]
      */

    val words = lines.as[String]
      .flatMap(_.split(" "))

    // 计算 word count
    val wordCounts = words.groupBy("value").count()

    // 开始查询，把查询结果打印在控制台（完整模式）
    /**
      * 输出模式有三种，complete,append,update：
      * Complete Mode:输出所有结果
      * Append Mode: 只输出当次批次中处理的结果（未和之前处理的结果合并）
      * Update Mode: 只输出结果有变化的行
      */
    val query = wordCounts.writeStream
      .outputMode("complete")
      .format("console")
      .start()
    // 执行
    query.awaitTermination()

}
```

# 3，内置输出源 Sink

## 3.1 Structured内置的输入源 Source

| Sink | Supported Output Modes | Options | Fault-tolerant | Notes |
| --- | --- | --- | --- | --- |
| File Sink | Append | path：输出路径（必须指定） | 支持容错（exactly-once） | 支持分区写入 |
| Kafka Sink | Append,Update,Complete | See the Kafka Integration Guide | 支持容错（at-least-once） | Kafka Integration Guide |
| Foreach Sink | Append,Update,Complete | None | | Foreach Guide |
| ForeachBatch Sink | Append,Update,Complete | None | | Foreach Guide |
| Console Sink | Append,Update,Complete | numRows：每次触发器打印的行数（默认值：20） truncate：是否过长时截断输出（默认值：true | | |
| Memory Sink | Append,Complete | None | | 表名是查询的名字 |

### 3.1.1 File Sink

将结果输出到文件，支持格式parquet、csv、orc、json等

```
val fileSink = source.writeStream
  .format("parquet")
  //.format("csv")
  //.format("orc")
 // .format("json")
  .option("path", "data/sink")
  .option("checkpointLocation", "/tmp/temporary-" + UUID.randomUUID.toString)
  .start()
```

### 3.1.2 Console Sink

将结果输出到控制台

```
val consoleSink = source.writeStream
  .format("console")
  // 是否压缩显示
  .option("truncate", value = false)
  // 显示条数
  .option("numRows", 30)
  .option("checkpointLocation", "/tmp/temporary-" + UUID.randomUUID.toString)
  .start()
```

### 3.1.3 Memory Sink

将结果输出到内存，需要指定内存中的表名。可以使用sql进行查询

```
val memorySink = source.writeStream
  .format("memory")
  .queryName("memorySinkTable")
  .option("checkpointLocation", "/tmp/temporary-" + UUID.randomUUID.toString)
  .start()


new Thread(new Runnable {
  override def run(): Unit = {
    while (true) {
      spark.sql("select * from memorySinkTable").show(false)
      Thread.sleep(1000)
    }
  }
}).start()
memorySink.awaitTermination()
```

### 3.1.4 Kafka Sink

将结果输出到Kafka，需要将DataFrame转成key，value两列，或者topic、key、value三列

```
import org.apache.spark.sql.functions._
import spark.implicits._
val kafkaSink = source.select(array(to_json(struct("*"))).as("value").cast(StringType),
  $"timestamp".as("key").cast(StringType)).writeStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "localhost:9092")
  .option("checkpointLocation", "/tmp/temporary-" + UUID.randomUUID.toString)
  .option("topic", "hiacloud-ts-dev")
  .start()
```

### 3.1.5 Foreach Sink

Foreach 每一条记录，通过继承ForeachWriter[Row]，实现open()，process()，close()方法。在open方法了我们可以获取一个资源连接，如MySQL的连接。在process里我们可以获取一条记录，并处理这条数据发送到刚才获取资源连接的MySQL中，在close里我们可以关闭资源连接。注意，foreach是对Partition来说的，同一个分区只会调用一次open、close方法，但对于每条记录来说，都会调用process方法。

```
package test


import java.sql.{Connection, DriverManager, Statement}
import java.util.UUID

import org.apache.spark.sql.{DataFrame, ForeachWriter, Row, SparkSession}
import org.apache.spark.sql.streaming.{ProcessingTime, StreamingQuery}
import org.apache.spark.sql.types.StructType
class  JDBCSink(url:String, user:String, pwd:String) extends ForeachWriter[Row] {
  val driver = "com.mysql.jdbc.Driver"
  var connection:Connection = _
  var statement:Statement = _

  def open(partitionId: Long,version: Long): Boolean = {
    Class.forName(driver)
    connection = DriverManager.getConnection(url, user, pwd)
    statement = connection.createStatement
    true
  }

  def process(value:Row): Unit = {
    statement.executeUpdate("INSERT INTO zip_test " +
      "VALUES (" + value(0)+ "," + value(1) + ")")
  }

  def close(errorOrNull: Throwable): Unit = {
    connection.close
  }
}
object sss {
  def main(args: Array[String]): Unit = {

    // 创建Spark程序入口
    val sparkSession = SparkSession
      .builder()
      .appName("StructuredNetworkWordCount")
      .master("local[*]")
      .getOrCreate()

    // val frame: DataFrame = lines.toDF("id","name","age")
    val userSchema = new StructType().add("name", "string").add("age", "integer")
    val source = sparkSession
      .readStream
      // Schema must be specified when creating a streaming source DataFrame.
      .schema(userSchema)
      // 每个trigger最大文件数量
      .option("maxFilesPerTrigger",100)
      // 是否首先计算最新的文件，默认为false
      .option("latestFirst",value = true)
      // 是否值检查名字，如果名字相同，则不视为更新，默认为false
      .option("fileNameOnly",value = true)
      .json("jsonres")
```

```
    val url="jdbc:mysql://mysqlserverurl:3306/test"
    val user ="user"
    val pwd = "pwd"

    val writer = new JDBCSink(url,user, pwd)
    val query =
      source.writeStream.foreach(writer)

        .outputMode("update")
        .trigger(ProcessingTime("25 seconds"))
        .start()

  }

}
```