



SPARK STRUCTURED STREAMING 流式大数据处理

Zhang, Lubo (lubo.zhang@intel.com)

Yucai, Yu (yucai.yu@intel.com)

BDT/STO/SSG Aug, 2017

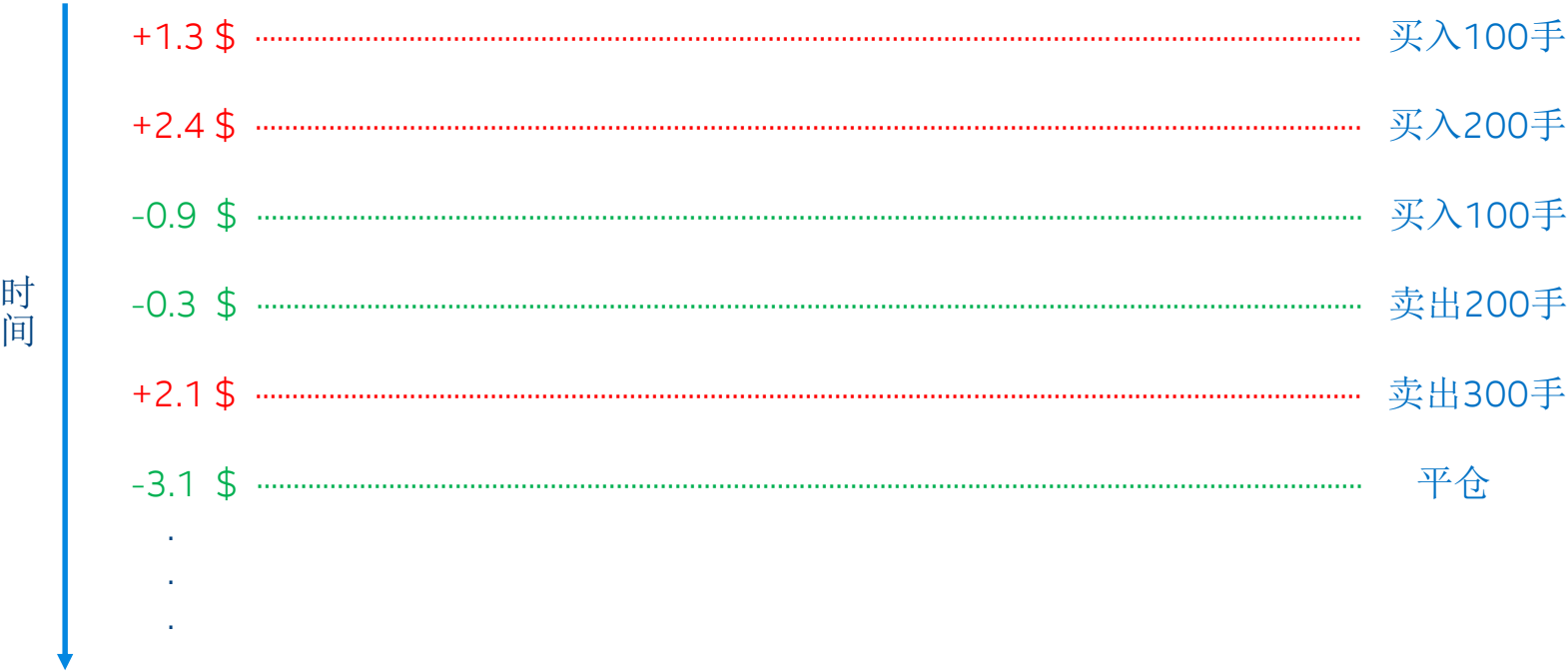
目录

- 流式数据
- Structured Streaming *的核心概念
- Structured Streaming *的高级话题
- Structured Streaming *的执行原理与高可用

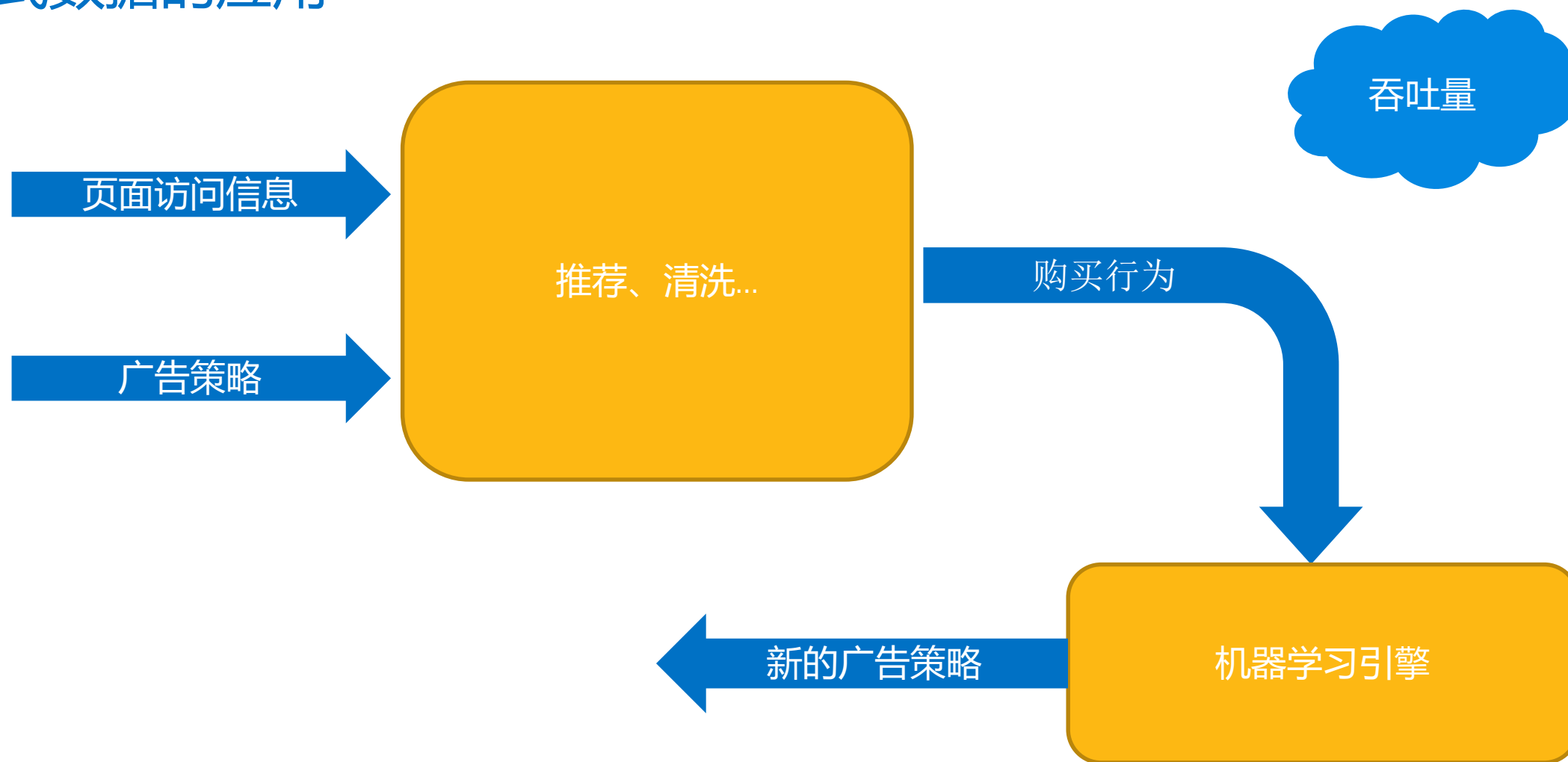
*Other names and brands may be claimed as the property of others.

什么是流式数据？

流式数据的应用



流式数据的应用



流式数据处理与生俱来的复杂性

复杂的数据

复杂多样的数据格式
(json, parquet, avro, ...)

脏数据，延迟，乱序

复杂的处理

与批数据互操作

机器学习

数据流上的交互式查询

复杂的系统

复杂多样的存储系统
(Kafka, S3, Kinesis, RDBMS, ...)

系统崩溃

Spark如何处理流式数据？

Structured Streaming *

基于Spark SQL*引擎构建的流处理系统

fast, scalable, fault-tolerant

丰富而统一的高级API

deal with complex data and complex workloads

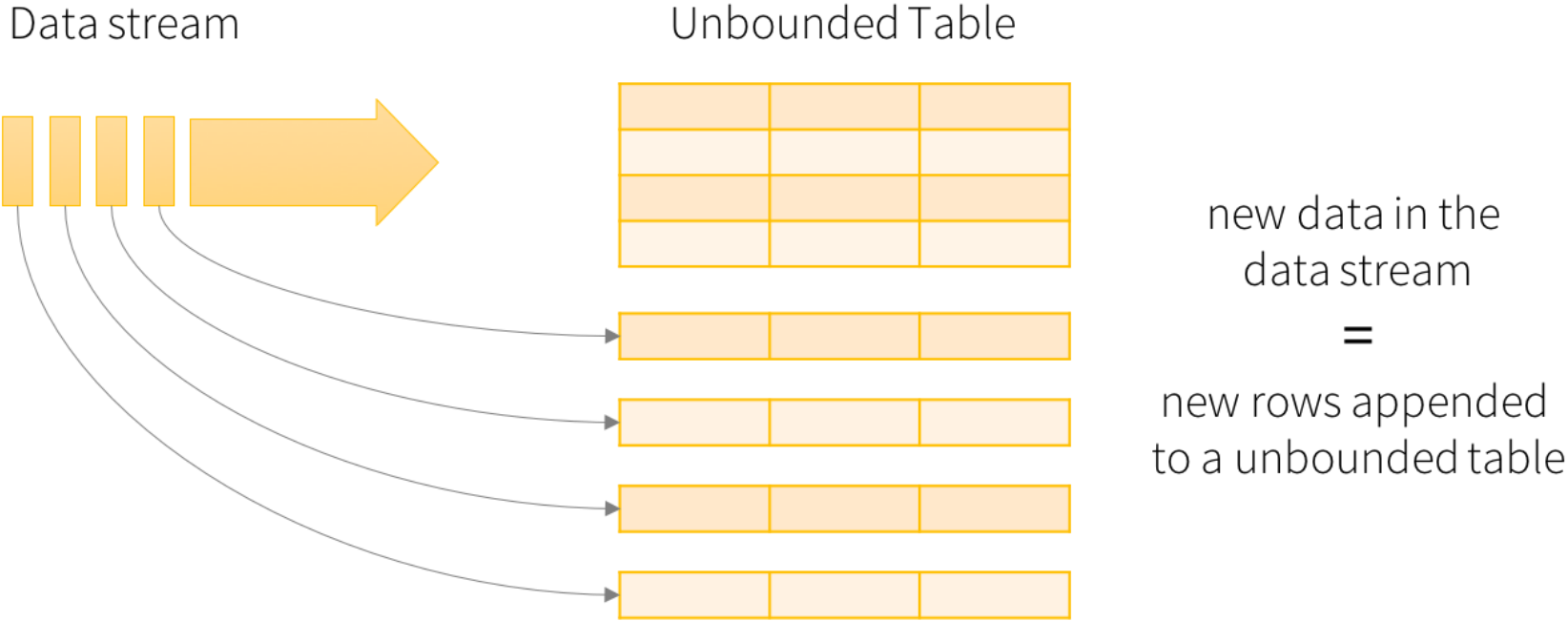
丰富的数据源支持

integrate with many storage system

*Other names and brands may be claimed as the property of others.

The simplest way to perform streaming analytics
is not having to *reason* about streaming at all

概念模型



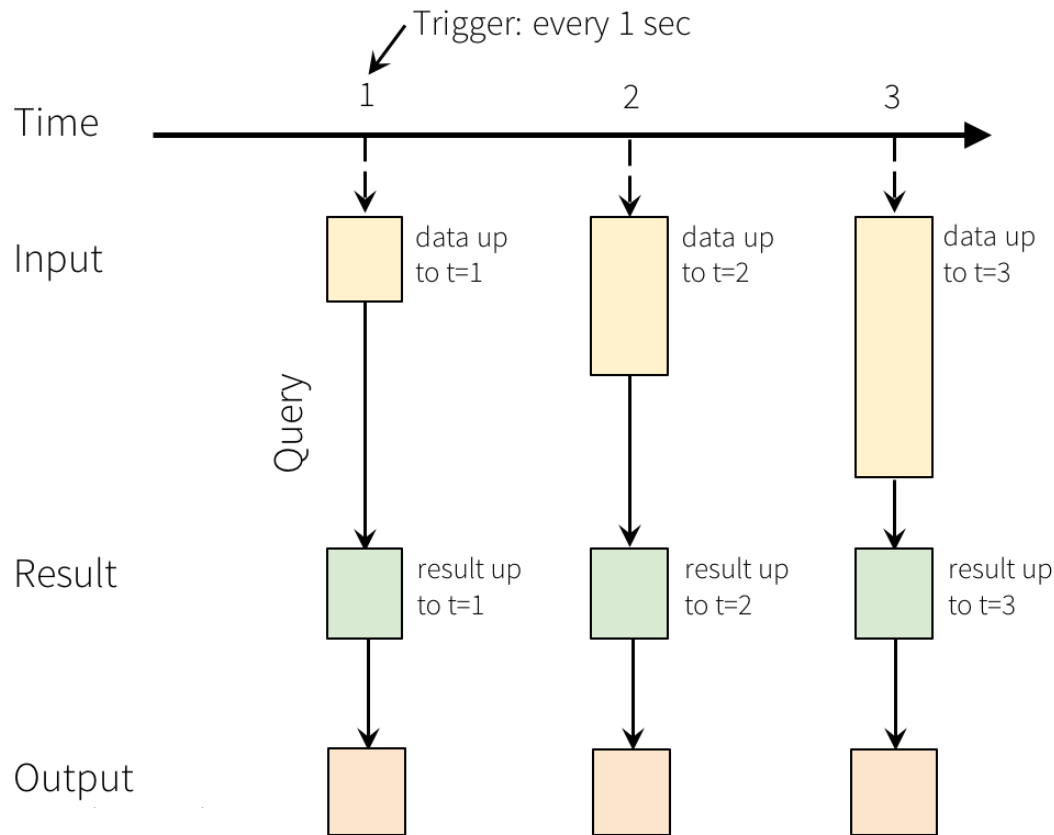
Data stream as an unbounded table

概念模型

将输入流看做是一张输入表

在每一个触发间隙（trigger interval），
输入表逐渐增长

当用户在输入表上应用查询的时候，
结果表随之发生变化



Programming Model for Structured Streaming

概念模型

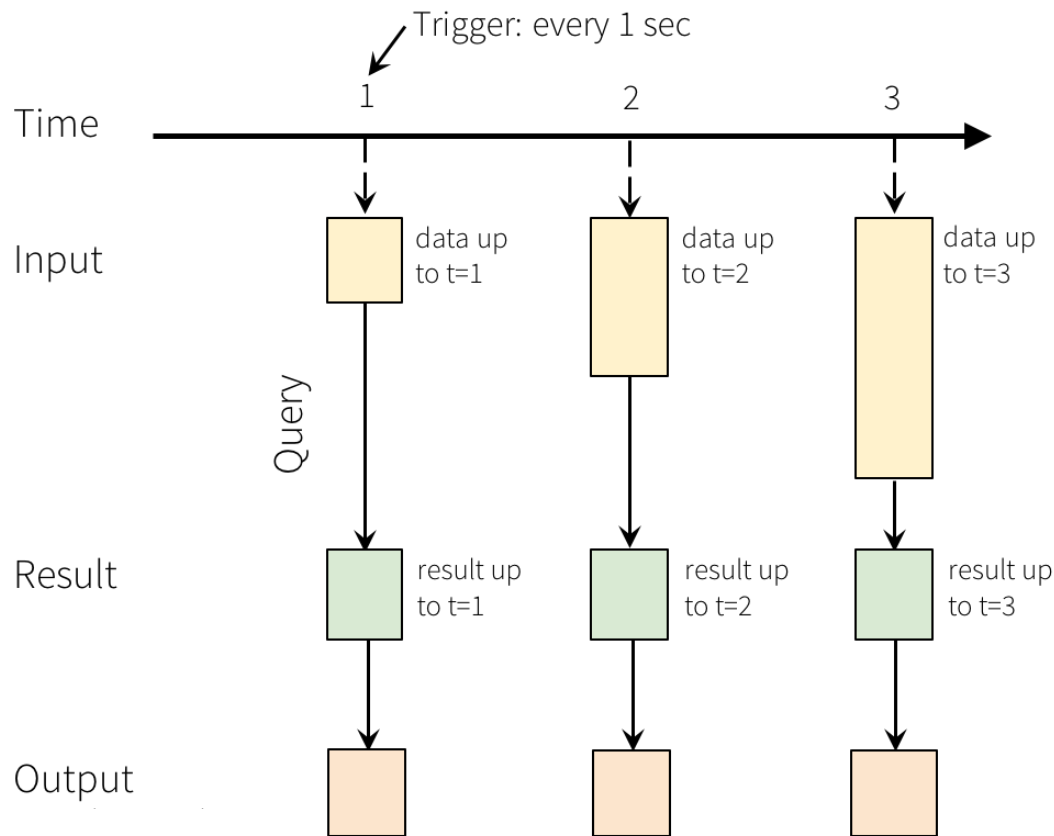
在每一个触发间隙（trigger interval），
我们可以输出特定的结果。

Output mode 定义了
每次触发需要输出的内容

Append mode: 仅输出新行

Complete mode: 输出全部的结果

Update output [2.1.1]: 输出自上次触发以来改变的行

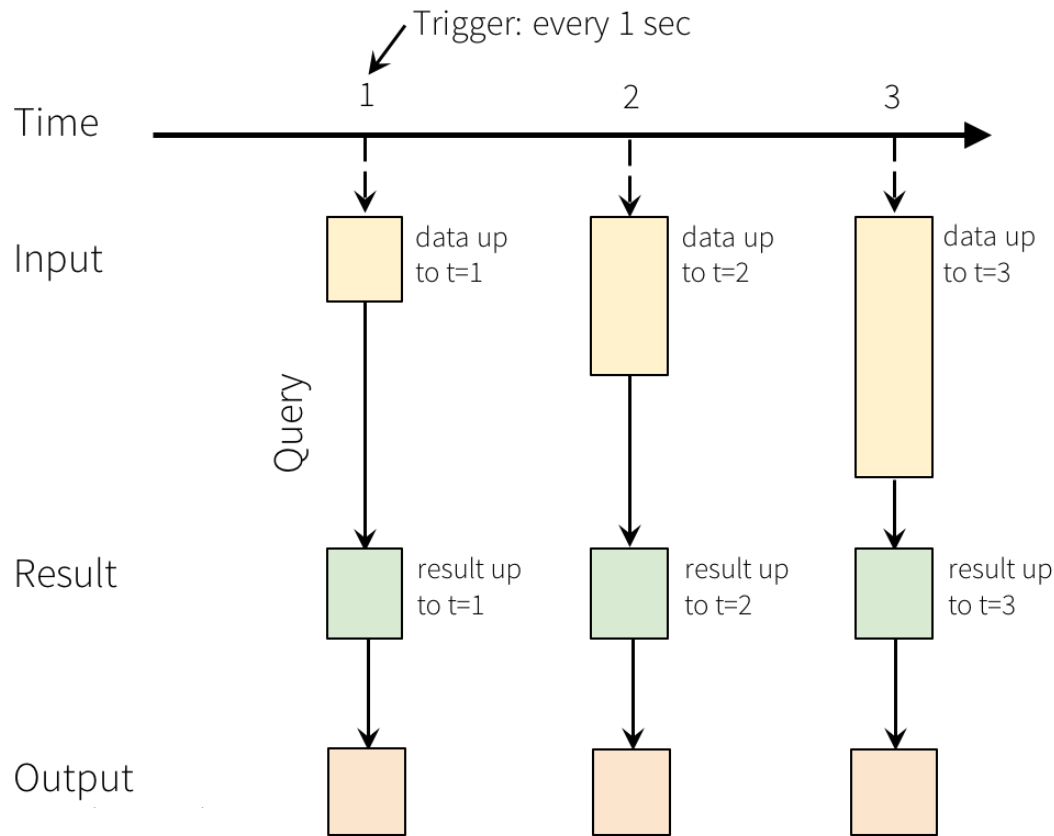


Programming Model for Structured Streaming

概念模型

Spark并不是在每次触发的时候都去处理完整的输入数据。

相反，它会将查询编译成增量式的查询，每次仅仅处理更新的数据。



Programming Model for Structured Streaming

Streaming word count

一个简单的流查询

```
Spark.readStream  
  .format("kafka")  
  .option("subscribe", "input")  
  .load()
```



数据源

- 指定数据来源
- 内建多种格式的支持
File/Kafka/Socket/Pluggable
- 使用union操作符将多种输入源合并起来

一个简单的流查询

```
Spark.readStream  
  .format("kafka")  
  .option("subscribe", "input")  
  .load()  
  .groupBy("value")  
  .agg (count("*"))
```

} 转换操作

- 可以使用DataFrames, Datasets or SQL 等编程接口
- Catalyst自动推算如何将这此转换增量地执行
- 内部的处理满足exactly-once语义

一个简单的流查询

```
Spark.readStream  
  .format("kafka")  
  .option("subscribe", "input")  
  .load()  
  .groupBy("value")  
  .agg (count("*"))  
  .writeStream  
  .format("kafka")  
  .option("topic", "output")
```

输出端

- 接受每个batch的输出
- 当输出是满足事务性的时候，可以保证 exactly once 语义（比如Files）
- 使用foreach执行任意用户自定义的操作

一个简单的流查询

```
Spark.readStream  
  .format("kafka")  
  .option("subscribe", "input")  
  .load()  
  .groupBy("value")  
  .agg (count("*"))  
  .writeStream  
  .format("kafka")  
  .option("topic", "output")  
  .trigger("5 second")  
  .outputMode("update")
```



触发器


- 设定触发频率
- 不指定意味着系统将尽快的处理

输出模式

- Complete – 输出全部结果
- Update – 输出改变行
- Append (默认) – 仅输出新行

一个简单的流查询

```
Spark.readStream  
  .format("kafka")  
  .option("subscribe", "input")  
  .load()  
  .groupBy("value")  
  .agg (count("*"))  
  .writeStream  
  .format("kafka")  
  .option("topic", "output")  
  .trigger("5 second")  
  .outputMode("update")  
  .option("checkpointLocation", "path")  
  .start()
```



Checkpoint

- 跟踪查询执行的进度
- 失败的时候重启查询

统一的 API – Dataset / Stream

Static =
bounded data

Streaming =
unbounded data



统一的API!



使用DataFrames做批处理

```
Input = spark.read  
    .format("json")  
    .load("path")
```

```
Result = input  
    .select("device", "signal")  
    .where("signal > 15")
```

```
Result.write  
    .format("parquet")  
    .save("path")
```

从json文件中创建Input DF

通过查询特定的设备创建 Result DF

将结果写入到parquet文件

使用DataFrames做流处理

```
Input = spark.readStream  
    .format("json")  
    .load("path")
```

```
Result = input  
    .select("device", "signal")  
    .where("signal > 15")
```

```
Result.writeStream  
    .format("parquet")  
    .start("path")
```

从json文件中创建Input DF

通过查询特定的设备创建 Result DF
Query没有任何改变

将结果写入到parquet文件

Continuous Aggregations

```
input.agg(avg("signal"))
```

持续不断的计算所有设备信号的均值

```
input.groupBy("device-type")  
  .agg("signal")
```

持续不断的计算每种设备信号的均值

Continuous Windowed Aggregations

```
input.groupBy(  
    $"device-type",  
    window($"event-time-col"), "10 min")  
    .avg("signal")
```

使用事件时间机制持续不断的计算
过去10分钟每种设备信号的均值

基于事件时间的处理可以**同样地**应用在流或者批处理任务中

Query Management

```
val query = df.writeStream  
  .format("console")  
  .outputMode("append")  
  .start()
```

```
query.stop()  
query.awaitTermination()  
query.exception()  
query.explain
```

query 对象用来监控和管理流式查询

- 停止查询
- 获取状态
- 获取错误信息

系统中可以有多个处于活动状态的查询过程

每个Query对象有一个唯一的名字用来跟踪对应状态

Joining streams with static data

```
val streamDS = spark  
  .readStream  
  .json("s3://logs")
```

```
val staticDS = spark  
  .read  
  .jdbc("jdbc://", "history-info")
```

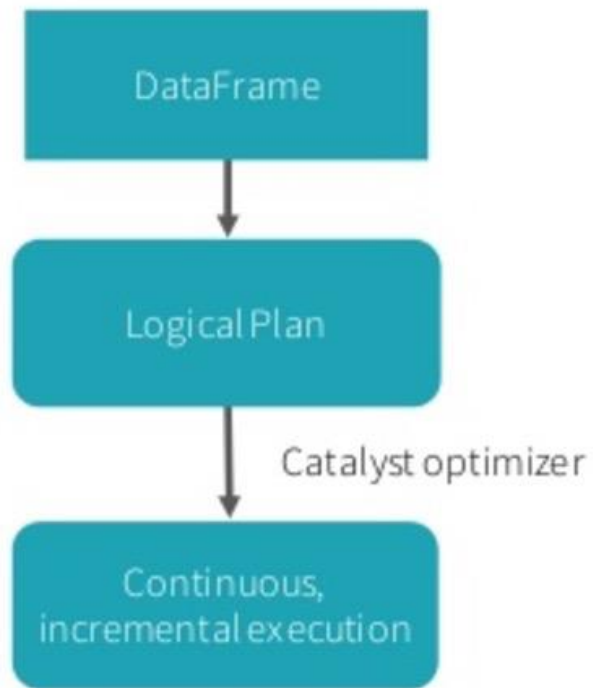
```
streamDS.join(staticDS, "customer_id")  
  .groupBy($"customer_name", hour($"time"))  
  .count()
```

Structured Streaming 使用 DataFrame 接口，
可以直接连接静态的数据表

Query Execution

Logically: 将流看成是对表的操作

Physically: Spark自动地将Query按流的方式执行



Structured Streaming 中的一些高级话题

Event Time

- 许多应用案例需要使用event time来聚合统计信息
E.g. 一小时内的各系统错误数量
- 多种挑战
E.g. 从数据中提取事件时间，处理延迟到达或者乱序的数据
- DStream APIs 并不能完美地支持event-time

基于Event time 的窗口操作

时间窗口是只是group的一种特殊情况

每小时记录的数量

```
df.groupBy(window("timestamp", "1 hour"))  
  .count()
```

每10分钟设备信号的均值

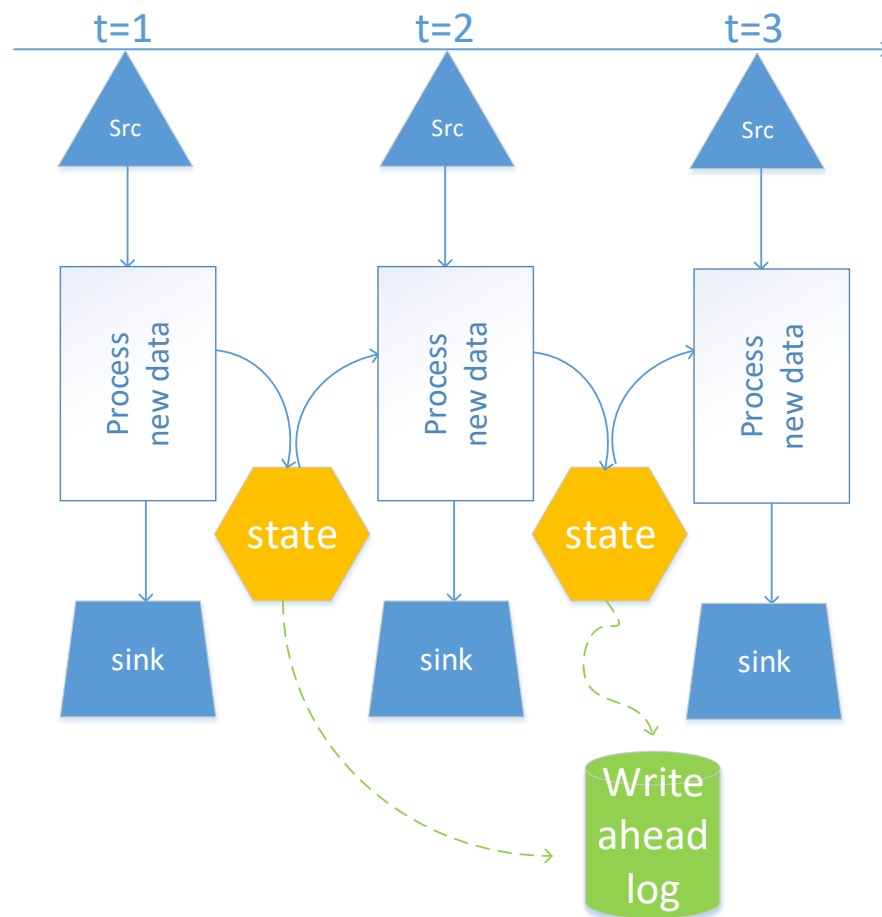
```
df.groupBy(  
    $"device",  
    window("timestamp", "10 minutes"))  
  .avg("signal")
```

完全支持UDAFs！

带状态的聚合

在每次触发操作之间，
聚合必须被保存为**分布式**的状态

- 触发器会首先读取之前的状态并保存更新后的状态
- 状态被保存在内存里并备份到 HDFS/S3(ahead log)
- 具有容错性，exactly-once guarantee



迟到数据的处理

保存状态以允许迟到的数据
去更新旧窗口的count

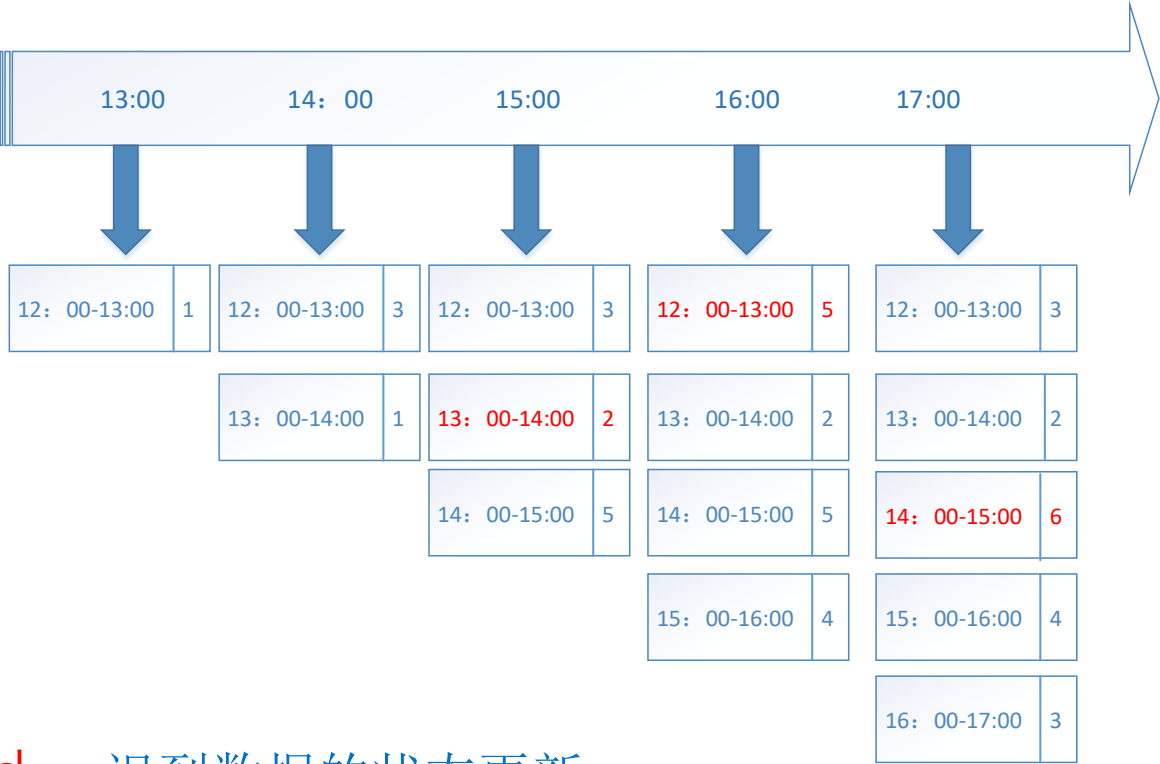


red: 迟到数据的状态更新

迟到数据的处理

保存状态以允许迟到的数据
去更新就窗口的count

但是，如果旧的窗口不被丢弃
状态的大小会无穷地增长



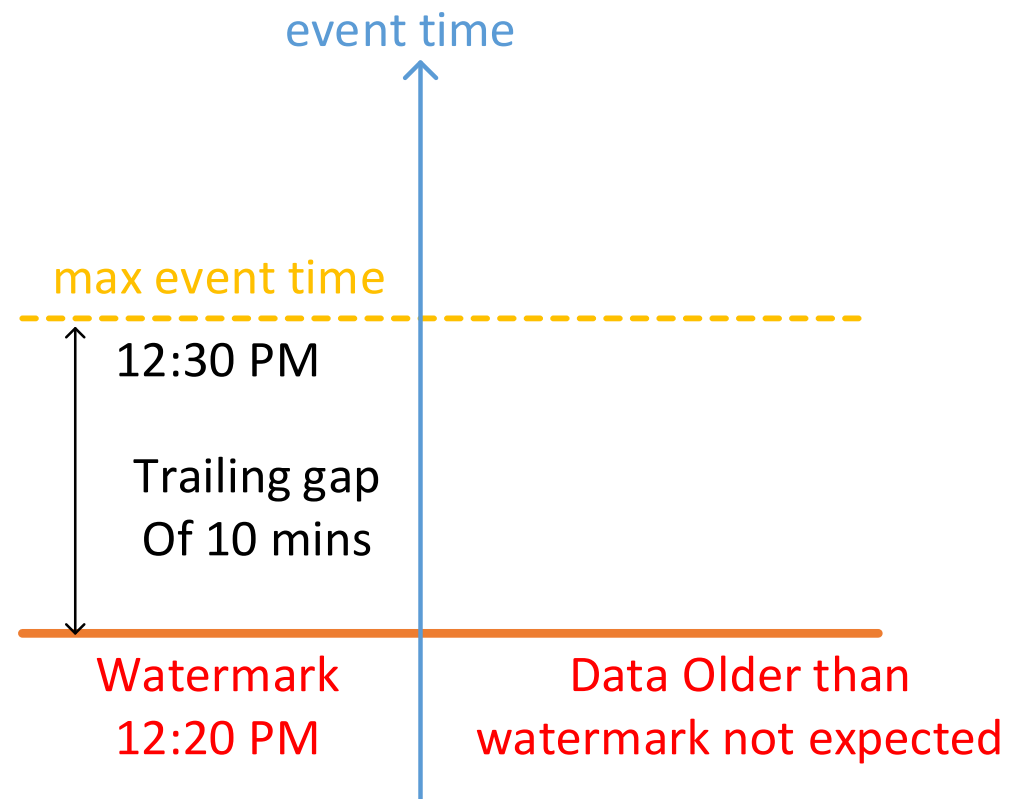
red: 迟到数据的状态更新

Watermarking (水线)

Watermark [Spark 2.1] - 定义了最大事件时间后的一个阈值，规定了允许处理的最晚数据，并丢弃过时的中间状态

根据可见的最大event time计算

时间间隔用户可以配置

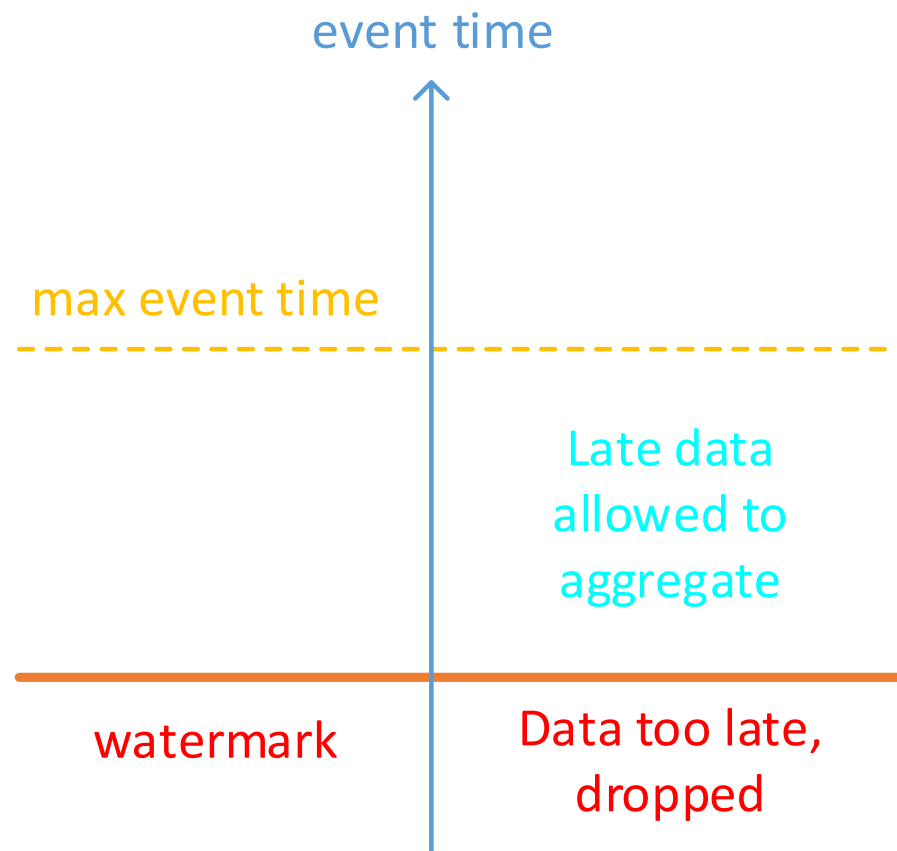


Watermarking (水线)

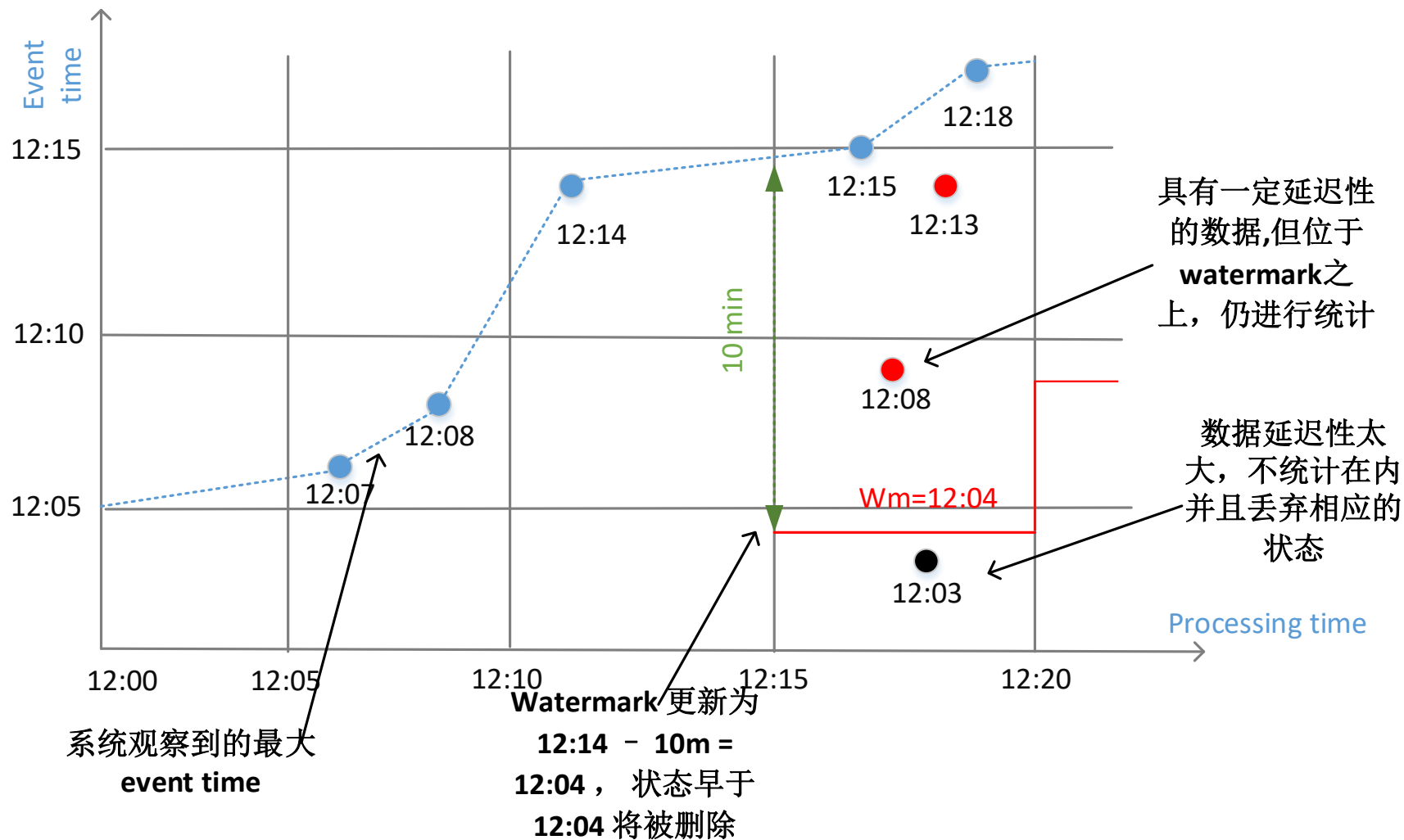
位于watermark之前到来的数据可能有一定延迟性，但允许进行聚合操作

位于watermark之后到来的数据由于延迟性很大，将被丢弃不作处理

晚于watermark的窗口也将被丢弃以节省用于保存中间状态的内存



Watermarking (水线)



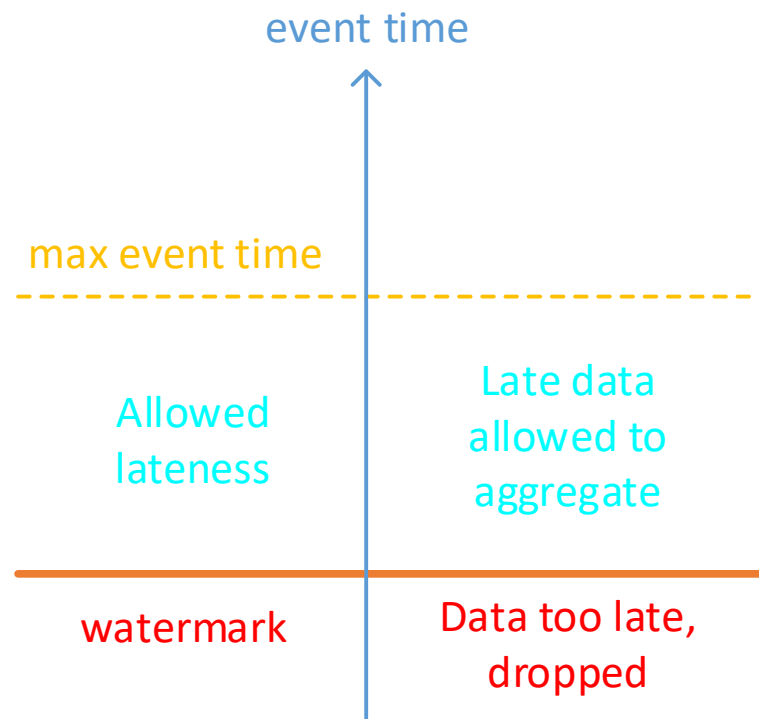
Watermarking (水线)

仅在无状态的操作中使用

(streaming aggs, dropDuplicates, mapGroupsWithState, ...)

在有状态的流式查询和批查询中将被忽略

```
val windowedCounts = words
  .withWatermark("timestamp", "10 minutes")
  .groupBy(
    window($"timestamp", "10 minutes"),
    $"word")
  .count()
```



再谈时间有关的几个概念

```
ssm.withWatermark("eventTime", "10 minutes")  
  .groupBy(window("eventTime", "5 minutes"))  
  .count()  
  .writeStream  
  .trigger("10 seconds")  
  .start()
```

再谈时间有关的几个概念

```
ssm.withWatermark("eventTime", "10 minutes")  
  .groupBy(window("eventTime", "5 minutes"))  
  .count()  
  .writeStream  
  .trigger("10 seconds")  
  .start()
```

在时间窗口内统计数据，
流式处理与批处理完全一样

再谈时间有关的几个概念

指定数据的延迟性

```
ssm.withWatermark("eventTime", "10 minutes")  
  .groupBy(window("eventTime", "5 minutes"))  
  .count()  
  .writeStream  
  .trigger("10 seconds")  
  .start()
```

再谈时间有关的几个概念

```
ssm.withWatermark("eventTime", "10 minutes")  
  .groupBy(window("eventTime", "5 minutes"))  
  .count()  
  .writeStream  
  .trigger("10 seconds")  
  .start()
```

更新的频率

任意的状态操作 [Spark 2.2]

mapGroupsWithState/flatMapGroupsWithState

允许用户在自定义的状态上进行用户自定义的操作

支持Processing Time和Event Time的timeout

支持Scala和java 接口

```
df.groupByKey(_._sessionId)
  .mapGroupsWithState
    (timeoutConf)
    (statefunc)
```

```
def statefunc(
  key: K,
  values: Iterator [V],
  state: GroupState [S]): U = {
  // update or remove state
  // set timeouts
  // return mapped values
}
```

自定义状态函数的例子

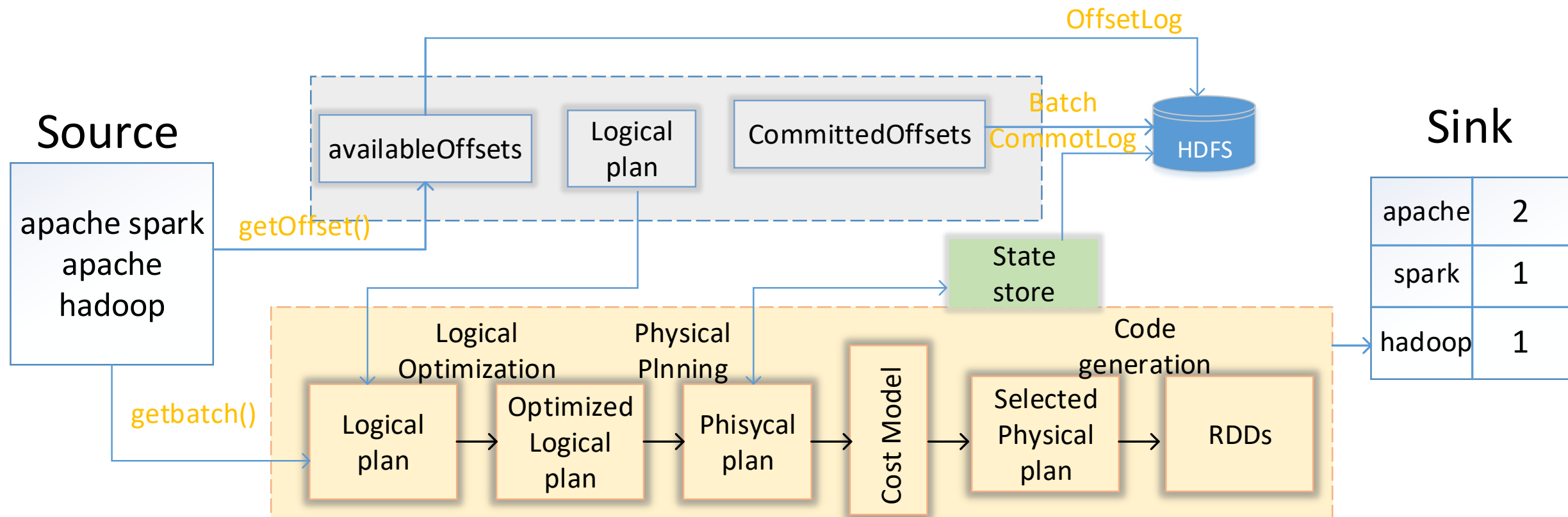
```
case class SessionInfo(numEvents: Int, startTimestampMs: Long, endTimestampMs: Long)
```

```
def stateFunc(sessionId: String, value: Iterator[Event], state: GroupState[SessionInfo]) = {
```

```
    if (state.hasTimedOut) {                // 调用的时候如果超时，则删掉该状态
        state.remove()
        Iterator[SessionUpdate] = ...
    } else if (state.exists) {              // 状态存在的话进行后续处理
        val newState: SessionInfo = ...    // 用户定义如何更新状态
        state.update(newState)             // 设置新的状态
        state.setTimeoutDuration("1 hour") // 配置超时时间
        Iterator.empty
    } else {
        val initialState: SessionInfo = ...
        state.update(initialState)         // 初始化状态
        state.setTimeoutDuration("1 hour") // 配置超时时间
        Iterator.empty
    }
}
```

Structured Streaming *的原理与高可用

*Other names and brands may be claimed as the property of others.



```
val lines = spark.readStream
  .format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load()
  .as[String]
  .filter($"value" === "Strata Hadoop")
  .explain
```

```
== Parsed Logical Plan ==
'Filter ('value = Strata Hadoop)
+- StreamingRelation DataSource(org.apache.spark.sql.Spark
```

```
== Physical Plan ==
Project [_1#10 AS value#15]
+- Filter (isnotnull(_1#10) && (_1#10 = Strata Hadoop))
   +- LocalTableScan [_1#10, _2#11]
```

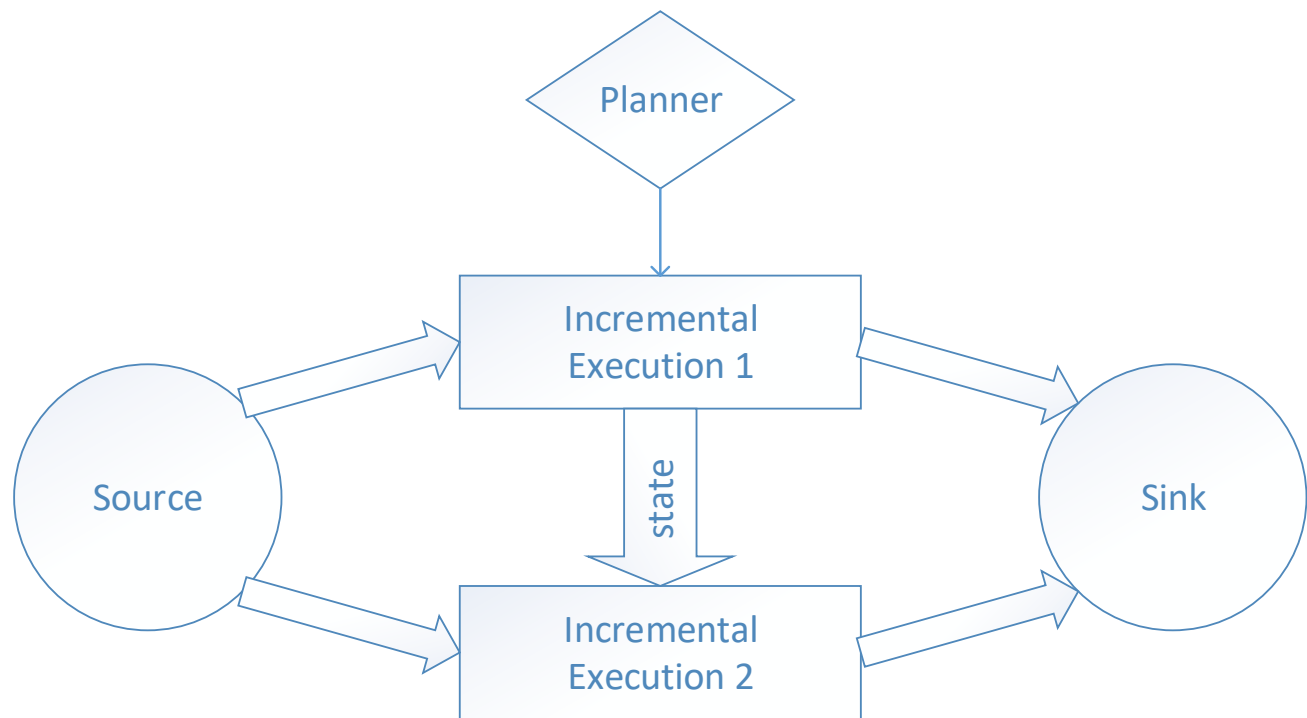
高可用

Structured Streaming *需要 24x7的情况下稳定运行，并且能够在主机出现问题的情况下自动恢复。

- Worker 出现问题：
 - ✓ Spark Core的架构可以原生的处理
- Driver 出现问题：
 - ✓ 由Cluster Manager负责Driver的重启（ Standalone, YARN, MESOS ）
 - ✓ 从checkpoint（ WAL ）读取进度，重新计算

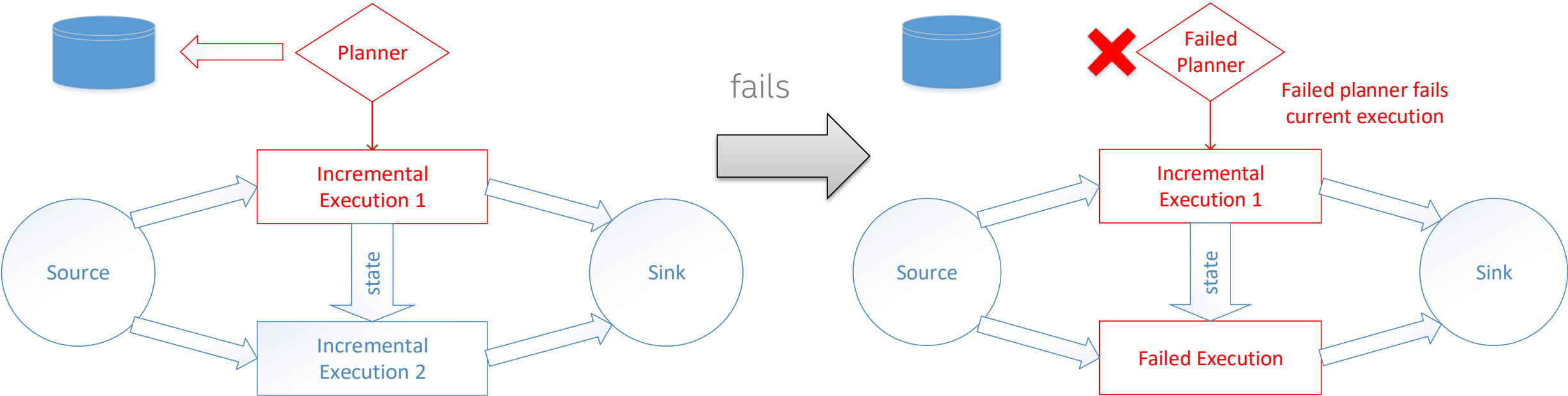
高可用

系统中的数据和元数据都需要是可恢复，可重放的



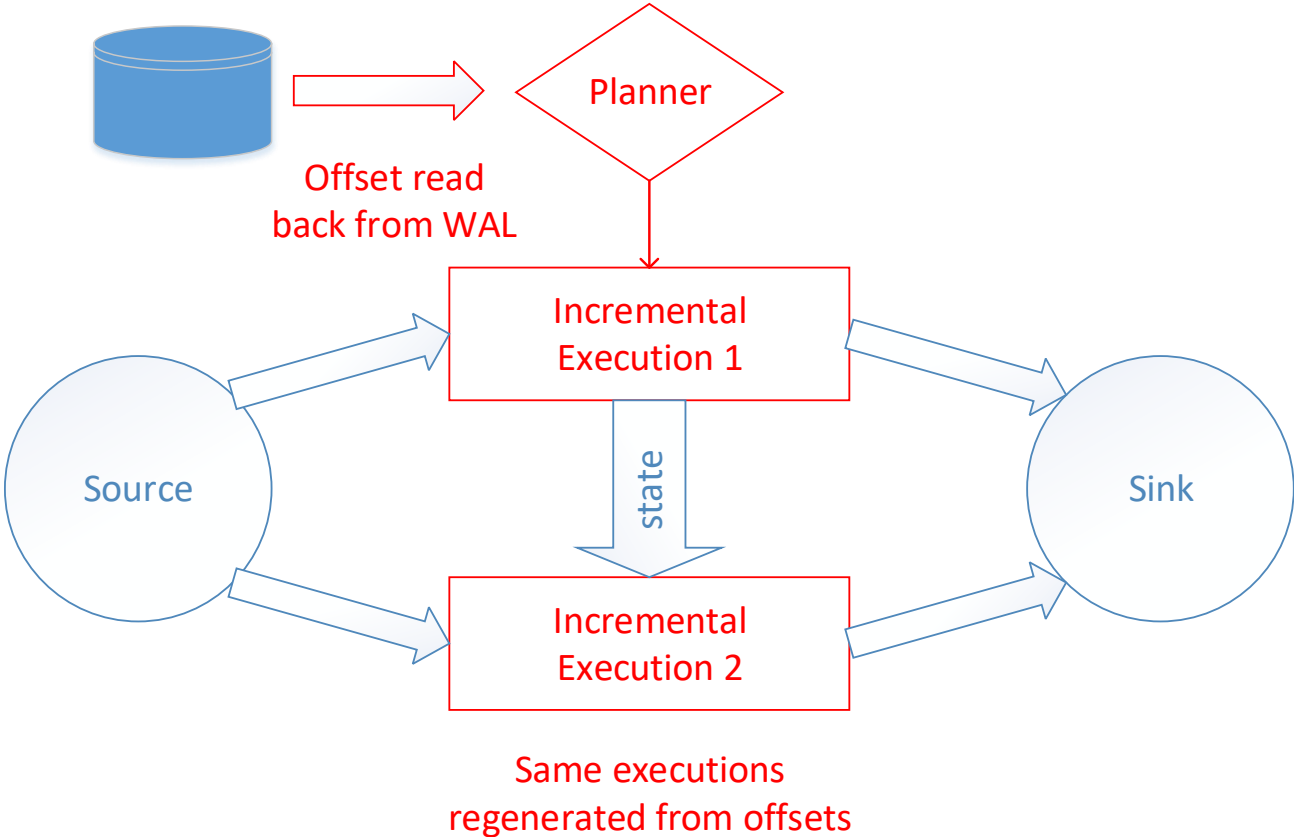
具有容错性的Planner

在执行前，数据offset会被写入具有容错性的WAL



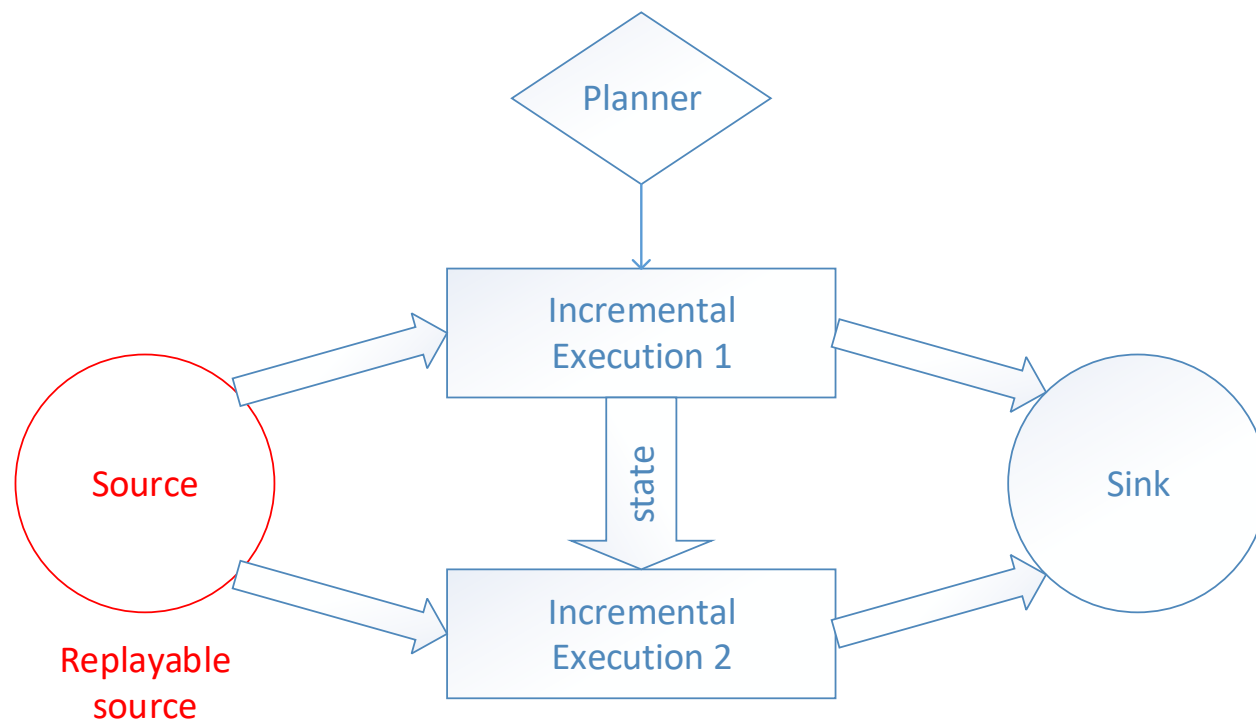
具有容错性的Planner

从WAL中读取数据offset，
重新计算



具有容错性的数据源

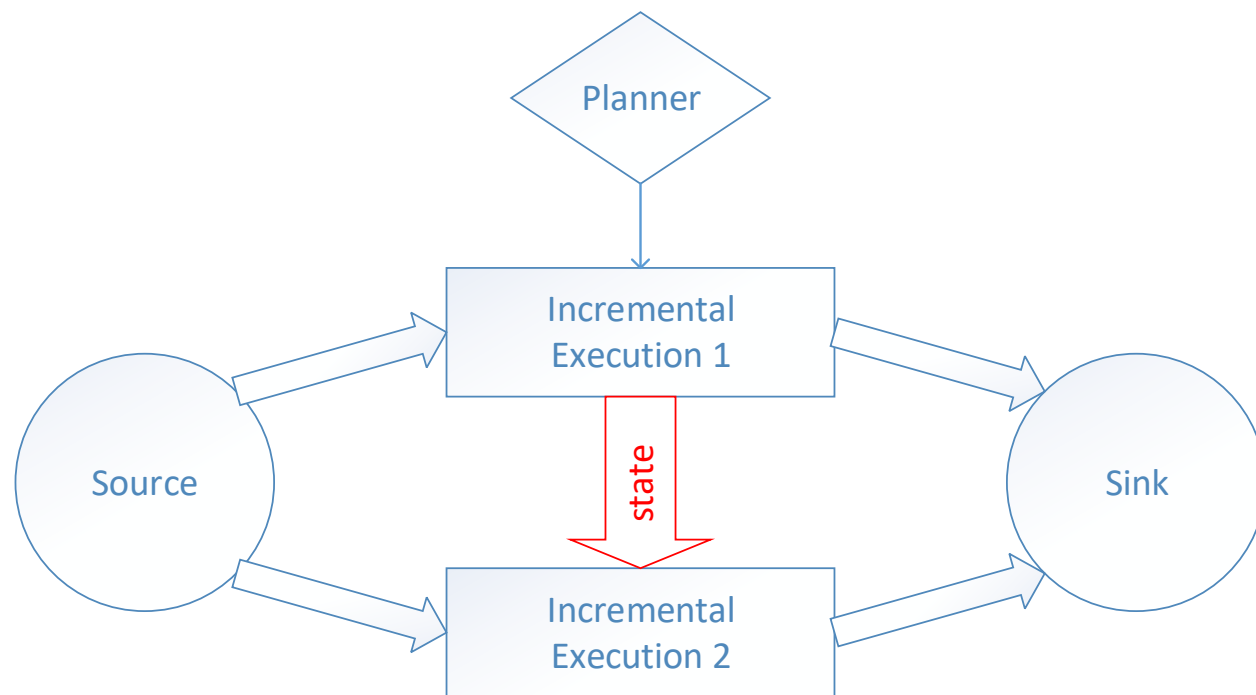
Structured streaming 要求数据源是可重放的（e.g. Kafka, Kinesis, files），并且能够根据planer提供的offset，生成完全一样的数据



具有容错性的状态

Spark 工作节点会在HDFS中保存数据处理的中间状态，包括版本，KV映射

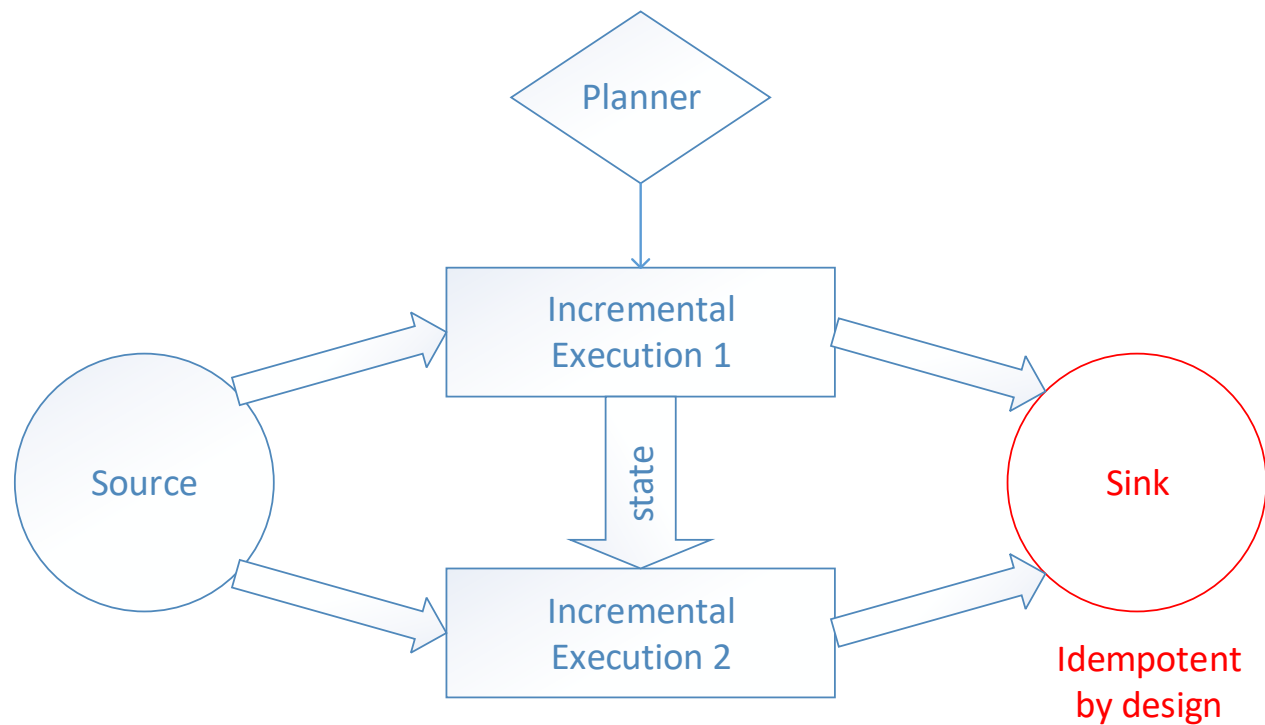
Planner会在查询失败后，确保应用正确的状态版本以进行恢复



State is fault-tolerant with WAL

具有容错性的输出端

具有幂等写入特性，在重新执行查询后，能避免结果的重复输出



高可用性

数据offset保存到WAL
+
状态管理
+
容错性的数据源和输出端
=

**end to end
exactly once !**

Reference

Structured streaming model picture
[streaming-programming-guide.html](https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html)

[https://spark.apache.org/docs/latest/structured-](https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html)

Legal Disclaimer

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

Intel, the Intel logo, Atom, Core, Iris, VTune, Xeon, and Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

* Other names and brands may be claimed as the property of others

© 2017 Intel Corporation.

Thank You !

