

📄 Apache流计算框架详细对比 (/a/1190000004593949)

[storm](https://segmentfault.com/t/storm/blogs) (<https://segmentfault.com/t/storm/blogs>)   [streaming](https://segmentfault.com/t/streaming/blogs) (<https://segmentfault.com/t/streaming/blogs>)   [spark-streaming](https://segmentfault.com/t/spark-streaming/blogs) (<https://segmentfault.com/t/spark-streaming/blogs>)  
[flink](https://segmentfault.com/t/flink/blogs) (<https://segmentfault.com/t/flink/blogs>)

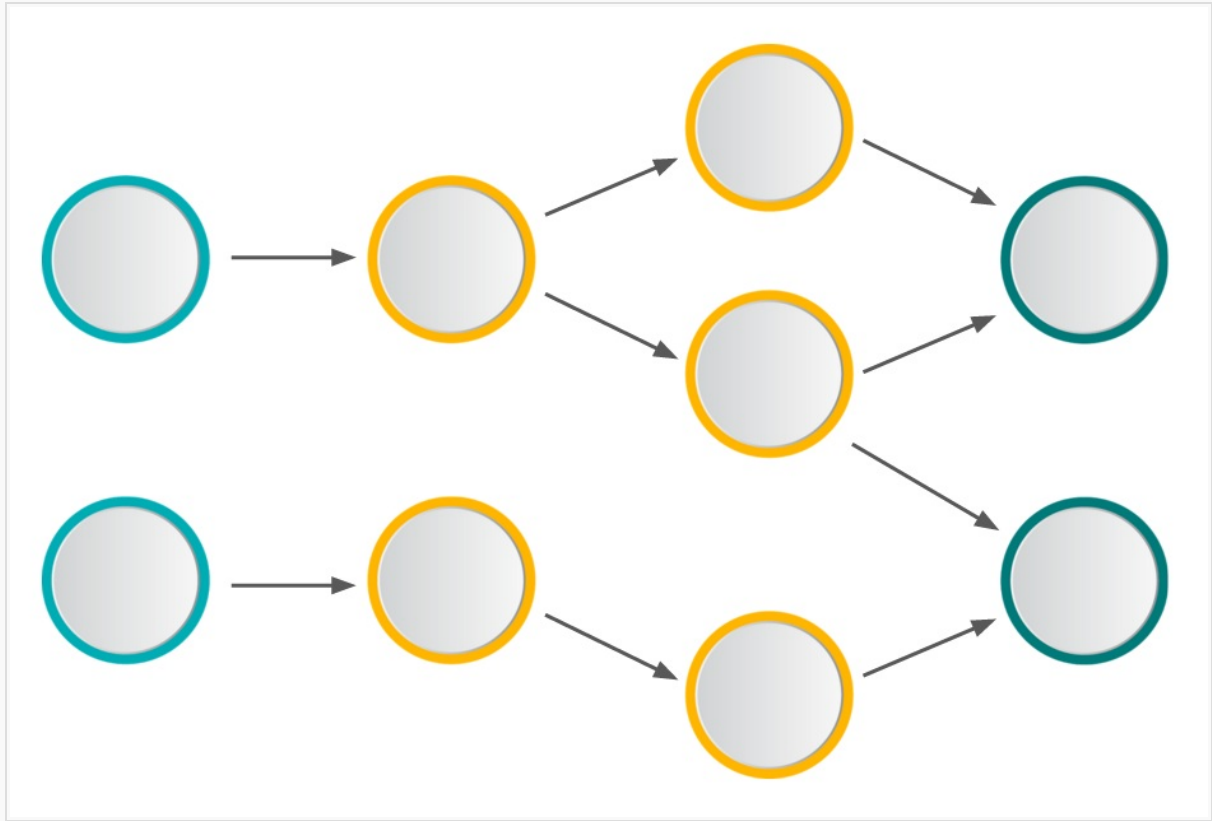
王下邀月熊\_Chevalier (<https://segmentfault.com/u/wxyyxc1992>) 2 天前发布

原文 (<http://www.cakesolutions.net/teamblogs/co...>)

几个月之前我们在这里讨论过<http://www.cakesolutions.net/teamblogs/introducti...>目前对于这种日渐增加的分布式流计算的需求的原因。当然，目前也有很多的各式各样的框架被用于处理这一些问题。现在我们会在这篇文章中进行回顾，来讨论下各种框架之间的相似点以及区别在哪里，还有就是从我的角度分析的，推荐的适用的用户场景。

如你所想，分布式的流处理也就是通常意义上的持续处理、数据富集以及对于无界数据的分析过程的组合。它是一个类似于MapReduce这样的通用计算模型，但是我们希望它能够在毫秒级别或者秒级别完成响应。这些系统经常被有向非循环图(Directed ACyclic Graphs,DAGs)来表示。

DAG主要功能即是用图来表示链式的任务组合，而在流处理系统中，我们便常常用DAG来描述一个流工作的拓扑。笔者自己是从Akka的Stream中的术语得到了启发。如下图所示，数据流经过一系列的处理器从源点流动到了终点，也就是用来描述这流工作。谈到Akka的Streams，我觉得要着重强调下分布式这个概念，因为即使也有一些单机的解决方案可以创建并且运行DAG，但是我们仍然着眼于那些可以运行在多机上的解决方案。



## Points of Interest

在不同的系统之间进行选择的时候，我们主要关注到以下几点。

- Runtime and Programming model(运行与编程模型)

一个平台提供的编程模型往往会决定很多它的特性，并且这个编程模型应该足够处理所有可能的用户案例。这是一个决定性的因素，我也会在下文中多次讨论。

- Functional Primitives(函数式单元)

一个合格的处理平台应该能够提供丰富的能够在独立信息级别进行处理的函数，像map、filter这样易于实现与扩展的一些函数。同样也应提供像aggregation这样的跨信息处理函数以及像join这样的跨流进行操作的函数，虽然这样的操作会难以扩展。

- State Management(状态管理)

大部分这些应用都有状态性的逻辑处理过程，因此，框架本身应该允许开发者去维护、访问以及更新这些状态信息。

- Message Delivery Guarantees(消息投递的可达性保证)

一般来说，对于消息投递而言，我们有至多一次(at most once)、至少一次(at least once)以及恰好一次(exactly once)这三种方案。

- at most once

At most once投递保证每个消息会被投递0次或者1次，在这种机制下消息很有可能会丢失。

- at least once

At least once投递保证了每个消息会被默认投递多次，至少保证有一次被成功接收，信息可能有重复，但是不会丢失。

- exactly once

exactly once意味着每个消息对于接收者而言正好被接收一次，保证即不会丢失也不会重复。

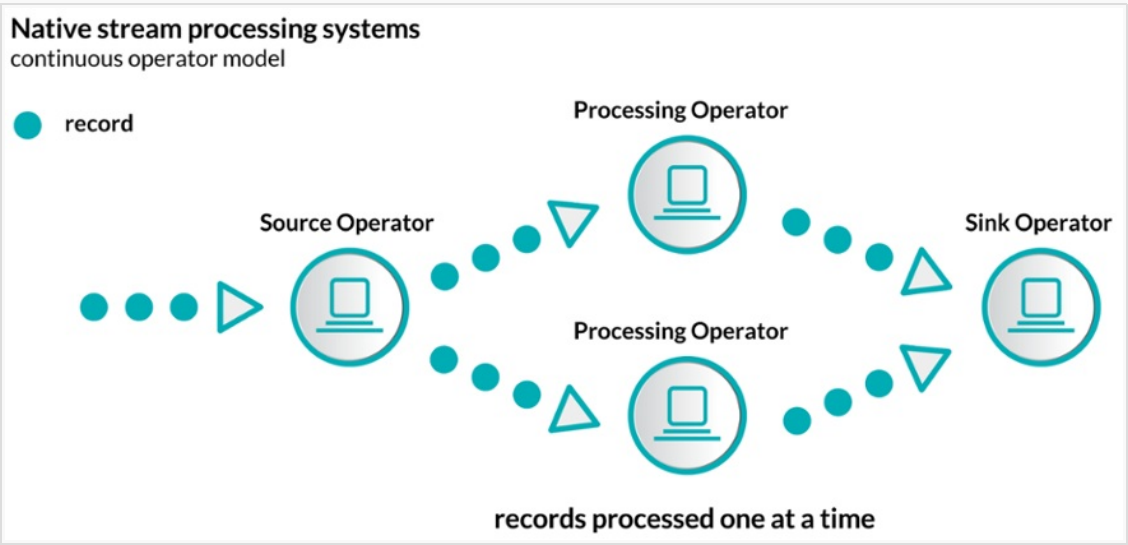
- Failures Handling

在一个流处理系统中，错误可能经常在不同的层级发生，譬如网络分割、磁盘错误或者某个节点莫名其妙挂掉了。平台要能够从这些故障中顺利恢复，并且能够从最后一个正常的状态继续处理而不会损害结果。

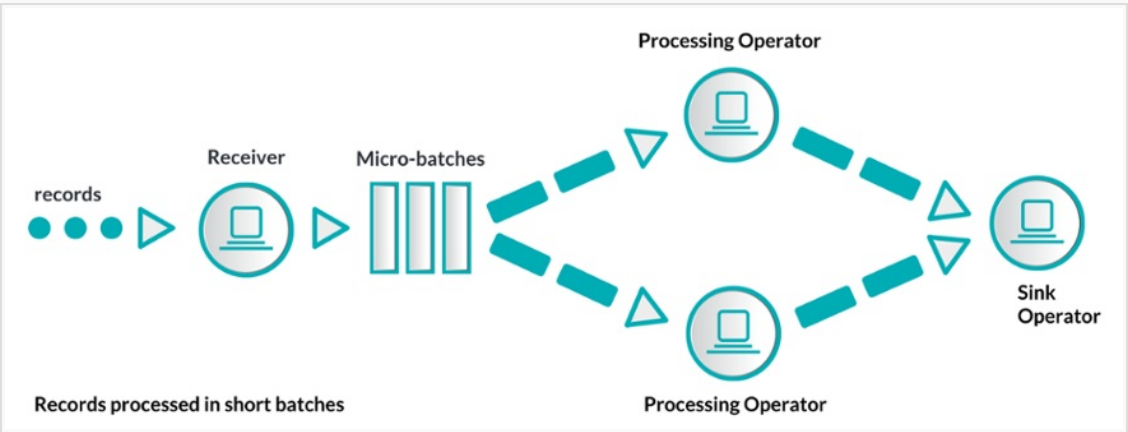
除此之外，我们也应该考虑到平台的生态系统、社区的完备程度，以及是否易于开发或者是否易于运维等等。

## RunTime and Programming Model

运行环境与编程模型可能是某个系统的最重要的特性，因为它定义了整个系统的呈现特性、可能支持的操作以及未来的一些限制等等。因此，运行环境与编程模型就确定了系统的能力与适用的用户案例。目前，主要有两种不同的方法来构建流处理系统,其中一个叫Native Streaming，意味着所有输入的记录或者事件都会根据它们进入的顺序一个接着一个的处理。



另一种方法叫做Micro-Batching。大量短的Batches会从输入的记录中创建出然后经过整个系统的处理，这些Batches会根据预设好的时间常量进行创建，通常是每隔几秒创建一批。



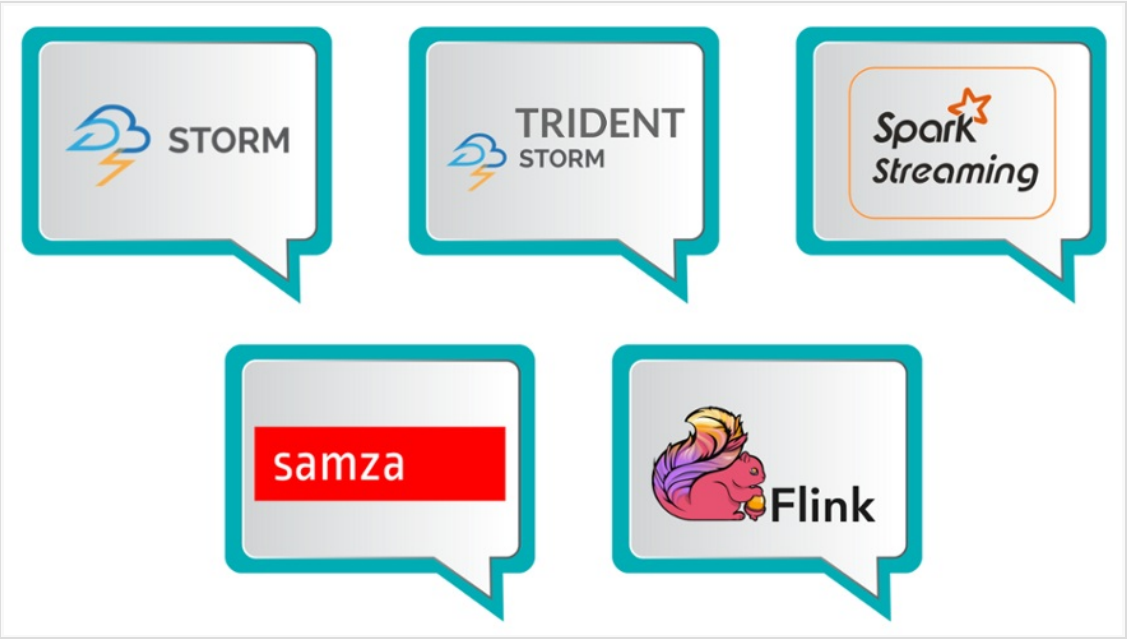
两种方法都有一些内在的优势与不足，首先来谈谈Native Streaming。好的一方面呢是Native Streaming的表现性会更好一点，因为它是直接处理输入的流本身的，并没有被一些不自然的抽象方法所限制住。同时，因为所有的记录都是在输入之后立马被处理，这样对于请求方而言响应的延迟就会优于那种Micro-Batching系统。处理这些，有状态的操作符也会更容易被实现，我们在下文中也会描述这个特点。不过Native Streaming系统往往吞吐量会比较低，并且因为它需要去持久化或者重放几乎每一条请求，它的容错的代价也会更高一些。并且负载均衡也是一个不可忽视的问题，举例而言，我们根据键对数据进行了分割并且想做进一步地处理。如果某些键对应的分区因为某些原因需要更多地资源去处理，那么这个分区往往就会变成整个系统的瓶颈。

而对于Micro-Batching而言，将流切分为小的Batches不可避免地会降低整个系统的变现性，也就是可读性。而一些类似于状态管理的或者joins、splits这些操作也会更加难以实现，因为系统必须去处理整个Batch。另外，每个Batch本身也将架构属性与逻辑这两个本来不应该被糅合在一起的部分相连接了起来。而Micro-Batching的优势在于它的容错与负载均衡会更加易于实现，它只要简单地在某个节点上处理失败之后转发给另一个节点即可。最后，值得一提的是，我们可以在Native Streaming的基础上快速地构建Micro-Batching的系统。

而对于编程模型而言，又可以分为Compositional(组合式)与Declarative(声明式)。组合式会提供一系列的基础构件，类似于源读取与操作符等等，开发人员需要将这些基础构件组合在一起然后形成一个期望的拓扑结构。新的构件往往可以通过继承与实现某个接口来创建。另一方面，声明式API中的操作符往往会被定义为高阶函数。声明式编程模型允许我们利用抽象类型和其他其他的精选的材料来编写函数式的代码以及优化整个拓扑图。同时，声明式API也提供了一些开箱即用的高等级的类似于窗口管理、状态管理这样的操作符。下文中我们也会提供一些代码示例。






## Apache Streaming Landscape

目前已经有了各种各样的流处理框架，自然也无法在本文中全部囊括。所以我必须将讨论限定在某些范围内，本文中是选择了所有Apache旗下的流处理的框架进行讨论，并且这些框架都已经提供了Scala的语法接口。主要的话就是Storm以及它的一个改进Trident Storm，还有就是当下正火的Spark。最后还会讨论下来自LinkedIn的Samza以及比较有希望的Apache Flink。笔者个人觉得这是一个非常不错的选择，因为虽然这些框架都是出于流处理的范畴，但是他们的实现手段千差万别。



- Apache Storm 最初由Nathan Marz以及他的BackType的团队在2010年创建。后来它被Twitter收购并且开源出来，并且在2014年变成了Apache的顶层项目。毫无疑问，Storm是大规模流处理中的先行者并且逐渐成为了行业标准。Storm是一个典型的Native Streaming系统并且提供了大量底层的操作接口。另外，Storm使用了Thrift来进行拓扑的定义，并且提供了大量其他语言的接口。
- Trident 是一个基于Storm构建的上层的Micro-Batching系统，它简化了Storm的拓扑构建过程并且提供了类似于窗口、聚合以及状态管理等没有被Storm原生支持的功能。另外，Storm是实现了至多一次的投递原则，而Trident实现了恰巧一次的投递原则。Trident 提供了 Java, Clojure 以及 Scala 接口。
- 众所周知，Spark是一个非常流行的提供了类似于SparkSQL、Mlib这样内建的批处理框架的库，并且它也提供了Spark Streaming这样优秀地流处理框架。Spark的运行环境提供了批处理功能，因此，Spark Streaming毫无疑问是实现了Micro-Batching机制。输入的数据流会被接收者分割创建为Micro-Batches，然后像其他Spark任务一样进行处理。Spark 提供了 Java, Python 以及 Scala 接口。
- Samza最早是由LinkedIn提出的与Kafka协同工作的优秀地流解决方案，Samza已经是LinkedIn内部关键的基础设施之一。Samza重负依赖于Kafaka的基于日志的机制，二者结合地非常好。Samza提供了Compositional接口，并且也支持Scala。
- 最后聊聊Flink。Flink可谓一个非常老的项目了，最早在2008年就启动了，不过目前正在吸引越来越多的关注。Flink也是一个Native Streaming的系统，并且提供了大量高级别的API。Flink也像Spark一样提供了批处理的功能，可以作为流处理的一个特殊案例来看。Flink强调万物皆流，这是一个绝对的更好地抽象，毕竟确实是这样。

下表就简单列举了上述几个框架之间的特性：

					
Streaming Model	Native	Micro-batching	Micro-batching	Native	Native
API	Compositional		Declarative	Compositional	Declarative
Guarantees	At-least-once	Exactly-once	Exactly-once	At-least-once	Exactly-once
Fault Tolerance	Record ACKs		RDD based Checkpointing	Log-based	Checkpointing
State Management	Not build-in	Dedicated Operators	Dedicated DStream	Stateful Operators	Stateful Operators
Latency	Very Low	Medium	Medium	Low	Low
Throughput	Low	Medium	High	High	High
Maturity	High		High	Medium	Low

## Counting Words

Wordcount就好比流处理领域的HelloWorld，它能够很好地描述不同框架间的差异性。首先看看Storm是如何编写WordCount程序的：

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("spout", new RandomSentenceSpout(), 5);
builder.setBolt("split", new Split(), 8).shuffleGrouping("spout");
builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new Fields("word"));

...

Map<String, Integer> counts = new HashMap<String, Integer>();

public void execute(Tuple tuple, BasicOutputCollector collector) {
    String word = tuple.getString(0);
    Integer count = counts.containsKey(word) ? counts.get(word) + 1 : 1;
    counts.put(word, count);
    collector.emit(new Values(word, count));
}
```

首先来看看它的拓扑定义，在第2行那边是定义了一个Spout，也就是一个输入源。然后定义了一个Bolt，也就是一个处理的组件，用于将某个句子分割成词序列。然后还定义了另一个Bolt用来负责真实的词计算。5，8到12行省略的过程用于定义集群中使用了多少个线程来供每一个组件使用。如你所见，所有的定义都是比较底层的与手动的。接下来继续看看这个8-15行，也就是真正用于WordCount的部分代码。因为Storm没有内建的状态处理的支持，所以我必须自定义这样一个本地状态，和理想的相差甚远啊。下面我们继续看看Trident。

正如我上文中提及的，Trident是一个基于Storm的Micro-Batching的扩展，它提供了状态管理等等功能。

```
public static StormTopology buildTopology(LocalDRPC drpc) {
    FixedBatchSpout spout = ...

    TridentTopology topology = new TridentTopology();
    TridentState wordCounts = topology.newStream("spout1", spout)
        .each(new Fields("sentence"), new Split(), new Fields("word"))
        .groupBy(new Fields("word"))
        .persistentAggregate(new MemoryMapState.Factory(),
            new Count(), new Fields("count"));

    ...

}
```

从代码中就可以看出，在Trident中就可以使用一些上层的譬如 `each`、`groupBy` 这样的操作符，并且可以在Trident中内建的进行状态管理了。接下来我们再看看Spark提供的声明式的接口，要记住，与前几个例子不同的是，基于Spark的代码已经相当简化了，下面基本上就是要用到的全部的代码了：

```
val conf = new SparkConf().setAppName("wordcount")
val ssc = new StreamingContext(conf, Seconds(1))

val text = ...

val counts = text.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)

counts.print()

ssc.start()
ssc.awaitTermination()
```

每个Spark的流任务都需要一个 `StreamingContext` 用来指定整个流处理的入口。`StreamingContext` 定义了Batch的间隔，上面是设置到了1秒。在6-8行即是全部的词统计的计算过程，非常不一样啊。下面再看看Apache Samza，另一个代表性的组合式的API：

```
class WordCountTask extends StreamTask {

    override def process(envelope: IncomingMessageEnvelope, collector: MessageCollector,
        coordinator: TaskCoordinator) {

        val text = envelope.getMessage.asInstanceOf[String]

        val counts = text.split(" ").foldLeft(Map.empty[String, Int]) {
            (count, word) => count + (word -> (count.getOrElse(word, 0) + 1))
        }

        collector.send(new OutgoingMessageEnvelope(new SystemStream("kafka", "wordcount"), counts))

    }
}
```

Topology定义在了Samza的属性配置文件里，为了明晰起见，这里没有列出来。下面再看看Fink，可以看出它的接口风格非常类似于Spark Streaming，不过我们没有设置时间间隔：

```
val env = ExecutionEnvironment.getExecutionEnvironment

val text = env.fromElements(...)
val counts = text.flatMap ( _.split(" ") )
    .map ( (_, 1) )
    .groupBy(0)
    .sum(1)

counts.print()

env.execute("wordcount")
```

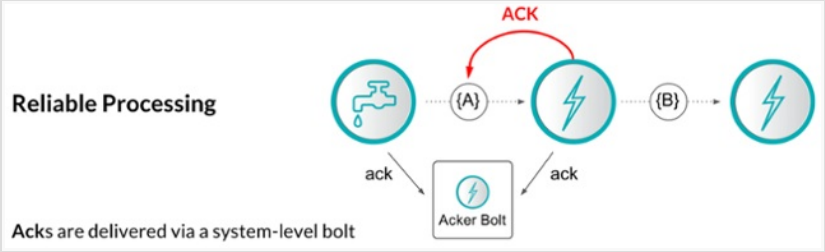
## Fault Tolerance

与批处理系统相比，流处理系统中的容错机制固然的会比批处理中的要难一点。在批处理系统中，如果碰到了什么错误，只要将计算中与该部分错误关联的重新启动就好了。不过在流计算的场景下，容错处理会更加困难，因为会不断地有数据进来，并且有些任务可能需要7\*24地运行着。另一个我们碰到的挑战就是如何保证状态的一致性，在每天结束的时候我们会开始事件重放，当然不可能所有的状态操作都会保证幂等性。下面我们来看看其他的系统是怎么处理的：

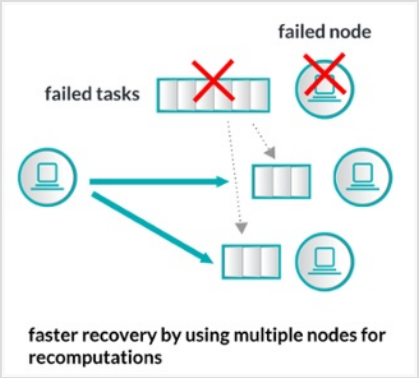
## Storm

Storm使用了所谓的逆流备份与记录确认的机制来保证消息会在某个错误之后被重新处理。记录确认这一个操作工作如下：一个操作器会在处理完成一个记录之后向它的上游发送一个确认消息。而一个拓扑的源会保存有所有其创建好的记录的备份。一旦受到了从Sinks发来的包含有所有记录的确认消息，就会把这些确认消息安全地删除掉。当发生错误时，如果还没有接收到全部的确认消息，就会从拓扑的源开始重放这些记录。这就确保了没有数据丢失，不过会导致重复的Records处理过程，这就属于At-Least投送原则。

Storm用一套非常巧妙的机制来保证了只用很少的字节就能保存并且追踪确认消息，但是并没有太多关注于这套机制的性能，从而使得Storm有较低地吞吐量，并且在流控制上存在一些问题，譬如这种确认机制往往在存在背压的时候错误地认为发生了故障。



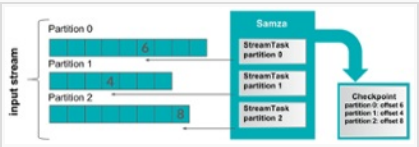
## Spark Streaming



Spark Streaming以及它的Micro-Batching机制则使用了另一套方案，道理很简单，Spark将Micro-Batches分配到多个节点运行，每个Micro-Batch可以成功运行或者发生故障，当发生故障时，那个对应的Micro-Batch只要简单地重新计算即可，因为它是持久化并且无状态的，所以要保证Exactly-Once这种投递方式也是很简单的。

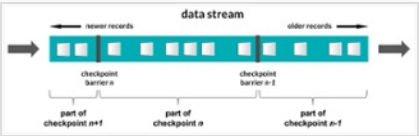
## Samza

Samza的实现手段又不一样了，它利用了一套可靠地、基于Offset的消息系统，在很多情况下指的就是Kafka。Samza会监控每个任务的偏移量，然后在接收到消息的时候修正这些偏移量。Offset可以是存储在持久化介质中的一个检查点，然后在发生故障时可以进行恢复。不过问题在于你并不知道恢复到上一个CheckPoint之后到底哪个消息是处理过的，有时候会导致某些消息多次处理，这也是At-Least的投递原则。



## Flink

Flink主要是基于分布式快照，每个快照会保存流任务的状态。链路中运送着大量的CheckPoint Barrier(检查点障碍，就是分隔符、标识器之类的)，当这些Barrier到达某个Operator的时候，Operator将自身的检查点与流相关联。与Storm相比，这种方式会更加高效，毕竟不用对每个Record进行确认操作。不过要注意的是，Flink还是Native Streaming，概念上和Spark还是相去甚远的。Flink也是达成了Exactly-Once投递原则。



## Managing State

大部分重要的流处理应用都会保有状态，与无状态的操作符相比，这些应用中需要一个输入和一个状态变量，然后进行处理最终输出一个改变了的状态。我们需要去管理、存储这些状态，要保证在发生故障的时候能够重现这些状态。状态的重造可能会比较困难，毕竟上面提到的不少框架都不能保证Exactly-Once，有些Record可能被重放多次。

## Storm



Storm是实践了At-Least投递原则，而怎么利用Trident来保证Exactly-Once呢？概念上还是很简单的，只需要使用事务进行提交Records，不过很明显这种方式及其低效。所以呢，还是可以构建一些小的Batches，并且进行一些优化。Trident是提供了一些抽象的接口来保证实现Exactly-Once，如下图所示，还有很多东西等着你去挖掘。

		State		
Spout		Non-transactional	Transactional	Opaque transactional
	Non-transactional	No	No	No
	Transactional	No	Yes	Yes
	Opaque transactional	No	No	Yes

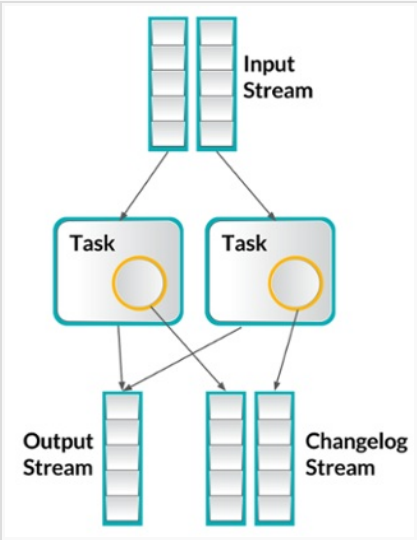
## Spark Streaming

当想要在流处理系统中实现有状态的操作时，我们往往想到的是一个长时间运行的Operator，然后输入一个状态以及一系列的Records。不过Spark Streaming是以另外一种方式进行处理的，Spark Streaming将状态作为一个单独地Micro-Batching流进行处理，所以在对每个小的Micro-Spark任务进行处理时会输入一个当前的状态和一个代表当前操作的函数，最后输出一个经过处理的Micro-Batch以及一个更新好的状态。

		State		
Spout		Non-transactional	Transactional	Opaque transactional
	Non-transactional	No	No	No
	Transactional	No	Yes	Yes
	Opaque transactional	No	No	Yes

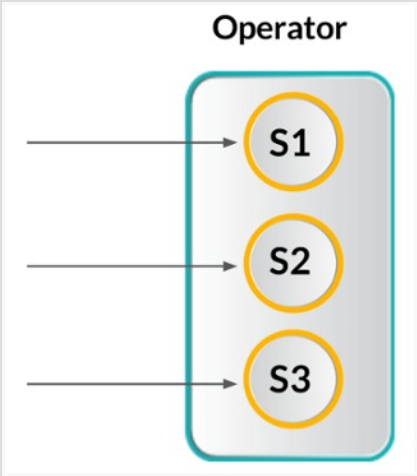
## Samza

Samza的处理方式更加简单明了，就是把它们放到Kafka中，然后问题就解决了。Samza提供了真正意义上的有状态的Operators，这样每个任务都能保有状态，然后所有状态的变化都会被提交到Kafka中。在有需要的情况下某个状态可以很方便地从Kafka的Topic中完成重造。为了提高效率，Samza允许使用插件化的键值本地存储来避免所有的消息全部提交到Kafka。这种思路如下图所示，不过Samza只是提高了At-Least这种机制，未来可能会提供Exactly-Once。



## Flink

Flink提供了类似于Samza的有状态的Operator的概念，在Flink中，我们可以使用两种不同的状态。第一种是本地的或者叫做任务状态，它是某个特定的Operator实例的当前状态，并且这种状态不会与其他进行交互。另一种呢就是维护了整个分区的状态。



## Counting Words with State

# Trident

```
public static StormTopology buildTopology(LocalDRPC drpc) {
    FixedBatchSpout spout = ...

    TridentTopology topology = new TridentTopology();

    TridentState wordCounts = topology.newStream("spout1", spout)
        .each(new Fields("sentence"), new Split(), new Fields("word"))
        .groupBy(new Fields("word"))
        .persistentAggregate(new MemoryMapState.Factory(), new Count(), new Fields("count"));

    ...
}
```

在第9行中，我们可以通过调用一个持久化的聚合函数来创建一个状态。

## Spark Streaming

```
// Initial RDD input to updateStateByKey
val initialRDD = ssc.sparkContext.parallelize(List.empty[(String, Int)])

val lines = ...
val words = lines.flatMap(_.split(" "))
val wordDstream = words.map(x => (x, 1))

val trackStateFunc = (batchTime: Time, word: String, one: Option[Int],
    state: State[Int]) => {
    val sum = one.getOrElse(0) + state.getOption.getOrElse(0)
    val output = (word, sum)
    state.update(sum)
    Some(output)
}

val stateDstream = wordDstream.trackStateByKey(
    StateSpec.function(trackStateFunc).initialState(initialRDD))
```

在第2行中，我们创建了一个RDD用来保存初始状态。然后在5, 6行中进行一些转换，接下来可以看出，在8-14行中，我们定义了具体的转换方程，即输入时一个单词、它的统计数量和它的当前状态。函数用来计算、更新状态以及返回结果，最后我们将所有的Bits一起聚合。

## Samza

```
class WordCountTask extends StreamTask with InitableTask {

    private var store: CountStore = _

    def init(config: Config, context: TaskContext) {
        this.store = context.getStore("wordcount-store")
            .asInstanceOf[KeyValueStore[String, Integer]]
    }

    override def process(envelope: IncomingMessageEnvelope,
        collector: MessageCollector, coordinator: TaskCoordinator) {

        val words = envelope.getMessage.asInstanceOf[String].split(" ")

        words.foreach { key =>
            val count: Integer = Option(store.get(key)).getOrElse(0)
            store.put(key, count + 1)
            collector.send(new OutgoingMessageEnvelope(new SystemStream("kafka", "wordcount"),
                (key, count)))
        }
    }
}
```

在上述代码中第3行定义了全局的状态，这里是使用了键值存储方式，并且在5~6行中定义了如何初始化。然后，在整个计算过程中我们都使用了该状态。

## Flink

```
val env = ExecutionEnvironment.getExecutionEnvironment

val text = env.fromElements(...)
val words = text.flatMap ( _.split(" ") )

words.keyBy(x => x).mapWithState {
    (word, count: Option[Int]) =>
    {
        val newCount = count.getOrElse(0) + 1
        val output = (word, newCount)
        (output, Some(newCount))
    }
}
```

在第6行中使用了 `mapWithState` 函数，第一个参数是即将需要处理的单次，第二个参数是一个全局的状态。

# Performance

合理的性能比较也是本文的一个重要主题之一。不同的系统的解决方案差异很大，因此也是很难设置一个无偏的测试。通常而言，在一个流处理系统中，我们常说的性能就是指延迟与吞吐量。这取决于很多的变量，但是总体而言标准为如果单节点每秒能处理500K的Records就是个合格的，如果能达到100万次以上就已经不错了。每个节点一般就是指24核附带24或者48GB的内存。

对于延迟而言，如果是Micro-Batch的话往往希望能在秒级别处理。如果是Native Streaming的话，希望能有百倍的减少，调优之后的Storm可以很轻易达到几十毫秒。

另一方面，消息的可达性保证、容错以及状态管理都是需要考虑进去的。譬如如果你开启了容错机制，那么会增加10%到15%的额外消耗。除此之外，以文章中两个WordCount为例，第一个是无状态的WordCount，第二个是有状态的WordCount，后者在Flink中可能会有25%额外的消耗，而在Spark中可能有50%的额外消耗。当然，我们肯定可以通过调优来减少这种损耗，并且不同的系统都提供了很多的可调优的选项。




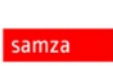

还有就是一定要记住，在分布式环境下进行大数据传输也是一件非常昂贵的消耗，因此我们要利用好数据本地化以及整个应用的序列化的调优。

## Project Maturity(项目成熟度)

在为你的应用选择一个合适的框架的时候，框架本身的成熟度与社区的完备度也是一个不可忽略的部分。Storm是第一个正式提出的流处理框架，它已经成为了业界的标准并且被应用到了像Twitter、Yahoo、Spotify等等很多公司的生产环境下。Spark则是目前最流行的Scala的库之一，并且Spark正逐步被更多的人采纳，它已成功应用在了像Netflix、Cisco、DataStax、Indel、IBM等等很多公司内。而Samza最早由LinkedIn提出，并且正在运行在几十家公司内。Flink则是一个正在开发中的项目，不过我相信它发展的会非常迅速。

## Summary

在我们进最后的框架推荐之前，我们再看一下上面那张图：

					
Streaming Model	Native	Micro-batching	Micro-batching	Native	Native
API	Compositional		Declarative	Compositional	Declarative
Guarantees	At-least-once	Exactly-once	Exactly-once	At-least-once	Exactly-once
Fault Tolerance	Record ACKs		RDD based Checkpointing	Log-based	Checkpointing
State Management	Not build-in	Dedicated Operators	Dedicated DStream	Stateful Operators	Stateful Operators
Latency	Very Low	Medium	Medium	Low	Low
Throughput	Low	Medium	High	High	High
Maturity	High		High	Medium	Low

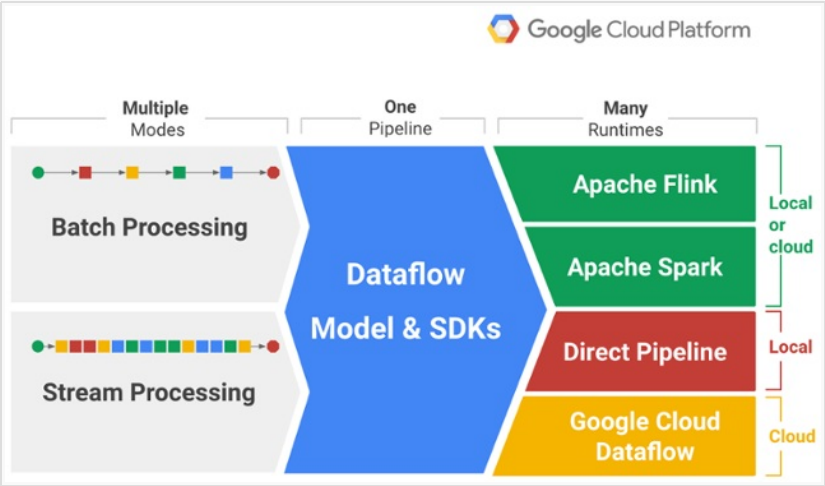
## Framework Recommendations

这个问题的回答呢，也很俗套，具体情况具体分析。总的来说，你首先呢要仔细评估下你应用的需求并且完全理解各个框架之间的优劣比较。同时我建议是使用一个提供了上层接口的框架，这样会更加的开发友好，并且能够更快地投入生产环境。不过别忘了，绝大部分流应用都是有状态的，因此状态管理也是不可忽略地一个部分。同时，我也是推荐那些遵循Exactly-Once原则的框架，这样也会让开发和维护更加简单。不过不能教条主义，毕竟还是有很多应用会需要At-Least-Once与At-Most-Once这些投递模式的。最后，一定要保证你的系统可以在故障情况下很快恢复，可以使用Chaos Monkey或者其他类似的工具进行测试。在我们之前的讨论中也发现这个快速恢复的能力至关重要。

- 对于小型与需要快速响应地项目，Storm依旧是一个非常好的选择，特别是在你非常关注延迟度的情况下。不过还是要谨记容错机制和Trident的状态管理会严重影响性能。Twitter目前正在设计新的流计算系统Heron用来替代Storm，它可以在单个项目中有很好地表现。不过Twitter可不一定会开源它。
- 对于Spark Streaming而言，如果你的系统的基础架构中已经使用了Spark，那还是很推荐你试试的。另一方面，如果你想使用Lambda架构，那Spark也是个不错的选择。不过你一定要记住，Micro-Batching本身的限制和延迟对于你而言不是一个关键因素。
- 如果你想用Samza的话，那最好Kafka已经是你的基础设施的一员了。虽然在Samza中Kafka只是个可插拔的组件，不过基本上所有人都会使用Kafka。正如上文所说，Samza提供了强大的本地存储功能，能够轻松管理数十G的状态数据。不过它的At-Least-Once的投递限制也是很大一个瓶颈。
- Flink目前在概念上是一个非常优秀的流处理系统，它能够满足大部分的用户场景并且提供了很多先进的功能，譬如窗口管理或者时间控制。所以当你发现你需要的功能在Spark当中无法很好地实现的时候，你可以考虑下Flink。另外，Flink也提供了很好地通用的批处理的接口，只不过你需要很大的勇气来将你的项目结合到Flink中，并且别忘了多关注关注它的路线图。

## Dataflow与开源

我最后一个要提到的就是Dataflow和它的开源计划。Dataflow是Google云平台的一个组成部分，是目前在Google内部提供了统一的用于批处理与流计算的服务接口。譬如用于批处理的MapReduce，用于编程模型定义的FlumeJava以及用于流计算的MillWheel。Google最近打算开源这货的SDK了，Spark与Flink都可以成为它的一个运行驱动。





# Conclusion

本文我们过了一遍常用的流计算框架，它们的特性与优劣对比，希望能对你有用吧。

2 天前发布 (/a/1190000004593949)

1 推荐

收藏

## 你可能感兴趣的文章

- LinkedIn 开源其专用于实时数据的处理分布式流处理框架 Samza (<https://segmentfault.com/a/1190000000311130>) 1 收藏, 1.7k 浏览
- Spark Streaming (<https://segmentfault.com/a/1190000003984844>) 1 收藏, 316 浏览
- Suro —— Netflix开源的分布式数据管道系统 (<https://segmentfault.com/a/1190000000371039>) 2 收藏, 2.2k 浏览

## 讨论区

请先 登录 () 后评论



([https://sponsor.segmentfault.com/ck.php?oaparams=2\\_bannerid=7\\_zoneid=2\\_cb=dd639c8033\\_oadest=http://click.aliyun.com/m/3936/](https://sponsor.segmentfault.com/ck.php?oaparams=2_bannerid=7_zoneid=2_cb=dd639c8033_oadest=http://click.aliyun.com/m/3936/))

本文隶属于专栏

某熊的全栈之路 (<https://segmentfault.com/blog/wxyyxc1992>)

知识, 应该在它该在的地方。一个热爱代码，热爱新技术的程序熊。现居南京，想找一个前端的兼职或者团队，有意向的联系QQ：384924552

[王下邀月熊\\_Chevalier \(<https://segmentfault.com/u/wxyyxc1992>\)](https://segmentfault.com/u/wxyyxc1992)  
作者

关注专栏

分享扩散：

...