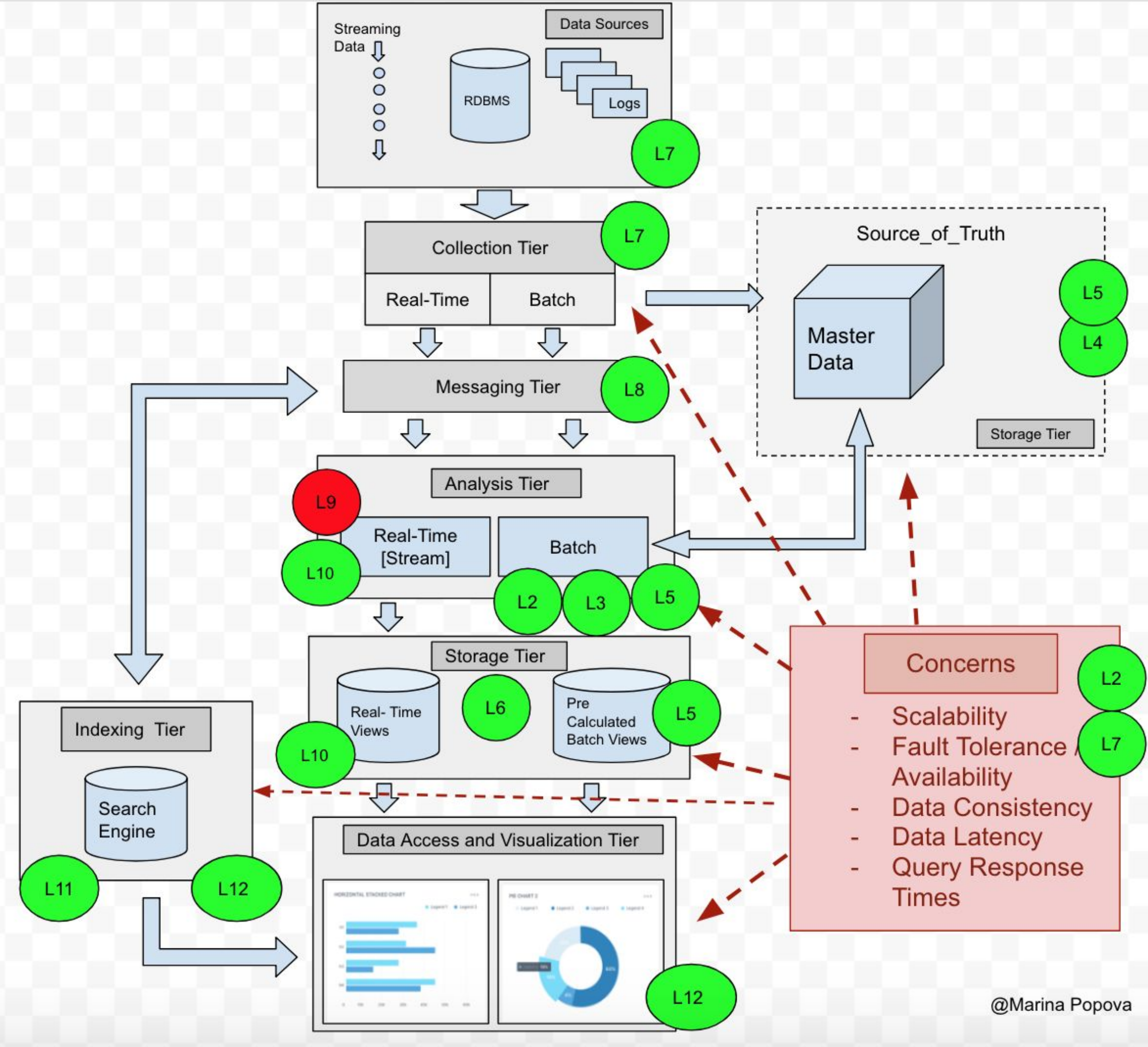
CSCI E-88 Principles Of Big Data Processing

Harvard University Extension, Fall 2019

Marina Popova

Lecture 9 Stream Processing

Where Are We?


Streaming Data
Data Sources
RDBMS
Logs
L7
Collection Tier
L7
Real-Time
Batch
Source_of_Truth
Master Data
L5
L4
Storage Tier
Messaging Tier
L8
Analysis Tier
L9
Real-Time [Stream]
Batch
L10
L2
L3
L5
Storage Tier
Real- Time Views
L6
Pre Calculated Batch Views
L5
L10
Indexing Tier
Search Engine
L11
L12
Data Access and Visualization Tier
L12
Concerns
L2
L7
- Scalability
- Fault Tolerance / Availability
- Data Consistency
- Data Latency
- Query Response Times
@Marina Popova

Agenda

- Admin stuff: midterm, next HW and Lecture topics change

- Stream Processing - main concepts
- Common Requirements and Operations
- Algorithms for Streaming analytics
- Streaming Applications Architecture overview
- Illustration: Spark Streaming

Stream Processing

What is Stream Processing?

Stream vs Batch processing

At-rest vs In-flight data

The data processing we were talking about so far was concerned with the data that was already collected and stored somewhere.
This type of data is often called "at-rest" data

The other type of data is the **"in-flight" or "real-time"** data often referred to as streaming data, which comes into the system all the time, never ends and can be retrieved by your application for as long as the application is up.

"Stream processing " or "real-time processing" terms refer to the type of data processing for this kind of data.

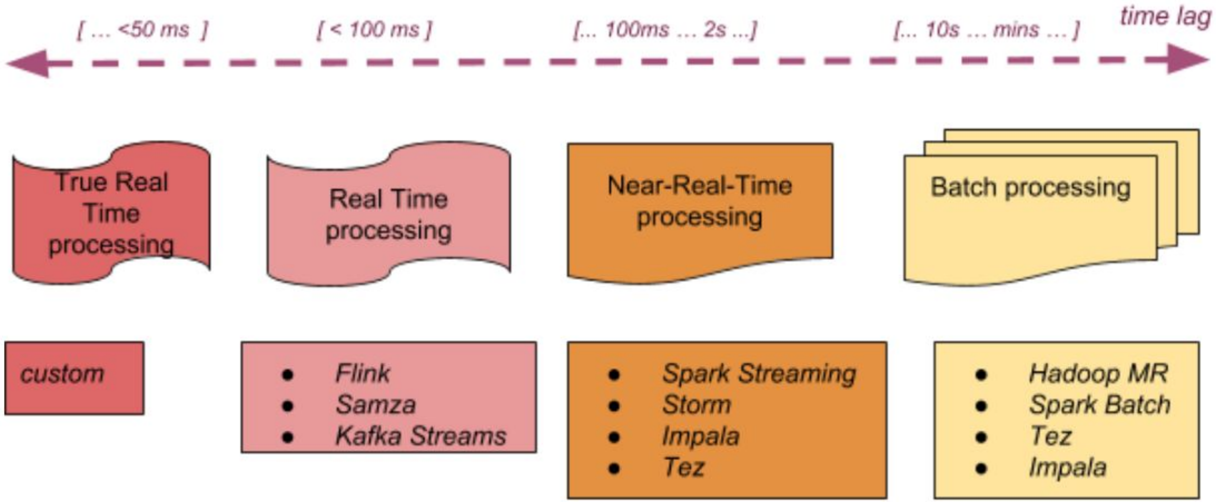
Stream Processing

How "real" is the "real-time" ? We talked about it in the Collection Tier :

- Near-Real-Time (NRT) Event Processing
  - 100/500 ms - 2 sec
- Real Time
  - <100 ms

We are focusing on the NRT processing now

NRT processing tools (Ref: "Hadoop Application Architectures")



time lag

[... <50 ms]

True Real Time processing

custom

[< 100 ms]

Real Time processing

- *Flink*
- *Samza*
- *Kafka Streams*

[... 100ms ... 2s ...]

Near-Real-Time processing

- *Spark Streaming*
- *Storm*
- *Impala*
- *Tez*

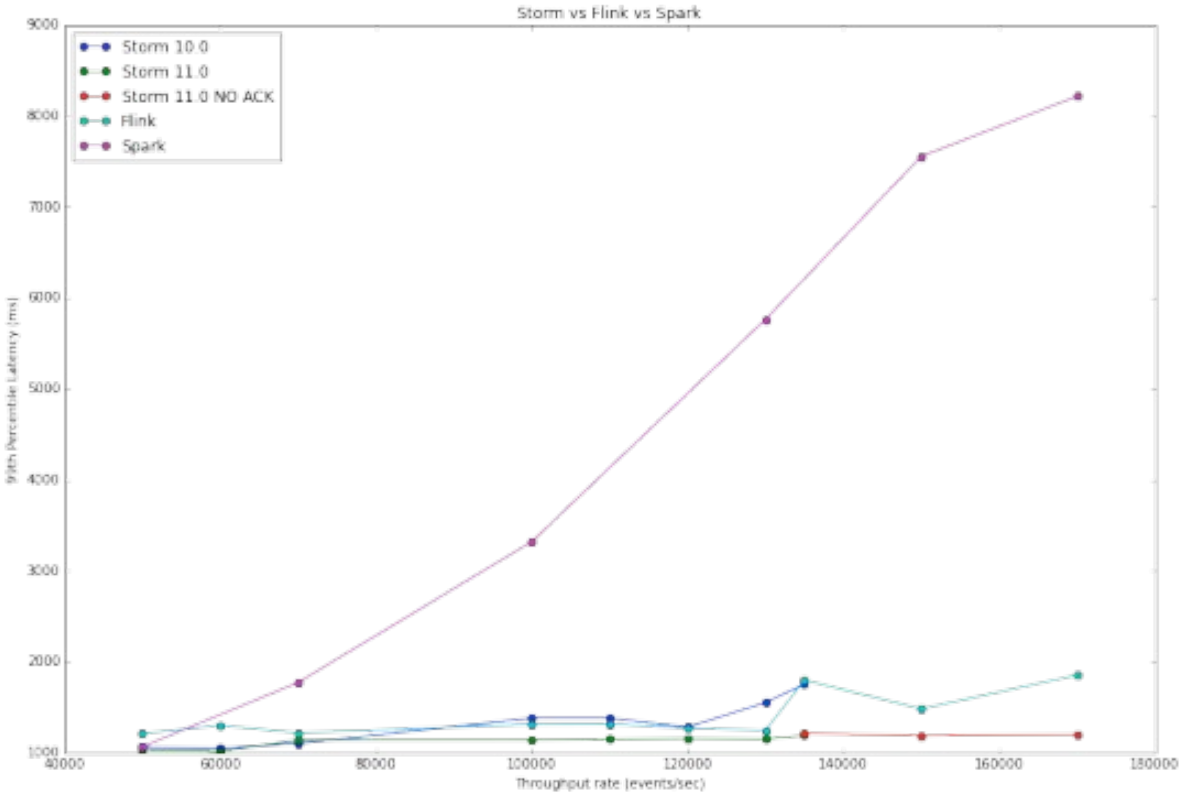
[... 10s ... mins ...]

Batch processing

- *Hadoop MR*
- *Spark Batch*
- *Tez*
- *Impala*

Stream Processing

the bottomline:
there is no good benchmark comparisons of up-to-date performance - numbers differ widely for different use cases!



Yahoo benchmark tests:
https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

@Marina Popova

Stream Processing

Main goal of stream processing:
Process the data as fast as it comes, and provide required analytics insights

The reason streaming processing is so fast is because it analyzes the data **before it hits the disk**. Reading data from disk incurs more latency than reading from RAM. Of course this all comes at a cost. You are only bound by how much data you can fit in the memory

Micro-batching vs Streaming

Strictly speaking, those are not one and the same.
The true streaming means processing event by event, as they arrive. An example of pure streaming frameworks are Storm and Kafka Streams.
On the other hand, Spark streaming is micro-batching, due to processing of events grouped into micro-batches.
It may or may not be important, based on the requirements for the specific processing in your applications

Stream Processing

Query Models:

- Batch systems:
  - **Static Dataset Queries**: query is submitted and results are returned to the client/app
- Streaming systems:
  - **Continuous Queries**: query is registered by the application when it starts, results are computed continuously, and returned to the client all the time (periodically) for as long as the application runs (or wants the results)
  - It is a "one-pass" processing model - you get to touch the data only once; no iterations are possible

Stream Processing

Event-time vs Stream-time

Batch systems: any timestamps associated with events as they are stored in , say, Master datastore, are always event times; thus all time-based aggregations are 100% correct from the time perspective

Streaming systems: this is no longer true….

- Event time: when an event took place
- Stream time: when the event entered the streaming system

They can be different for many reasons:

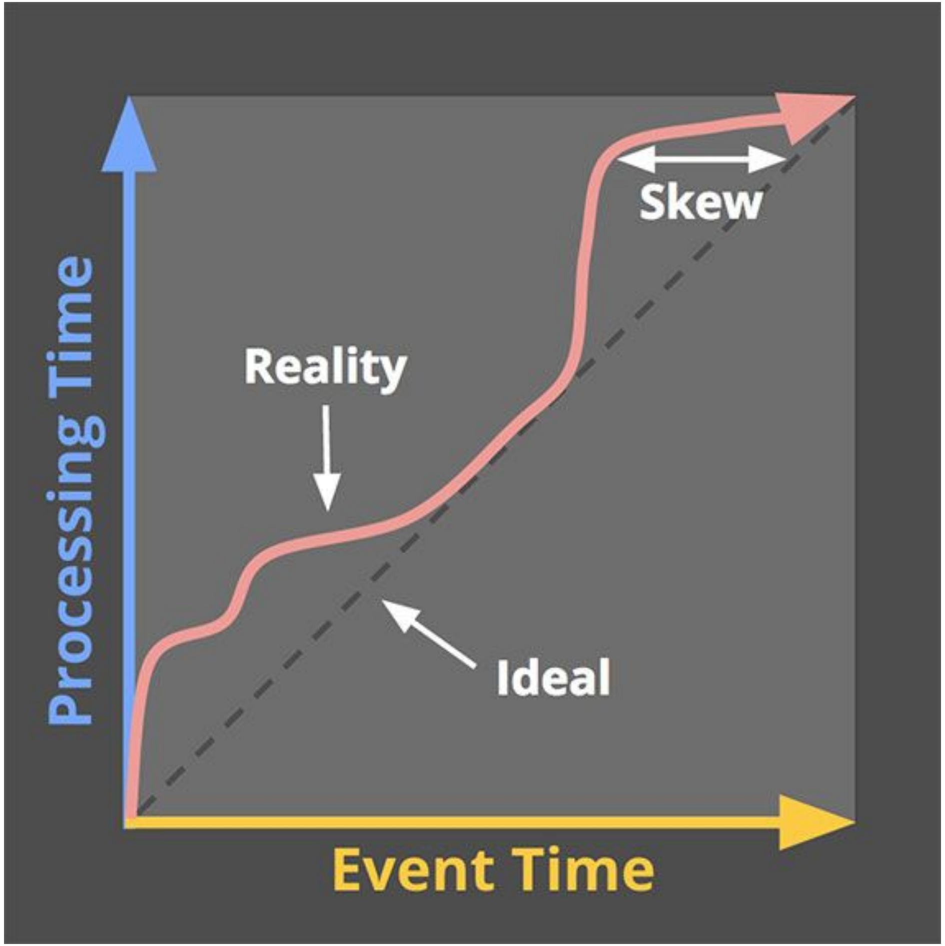
- Delay in the collection tier due to network slowdown or bursts of incoming data
- Out-of-order/old events due to failures and reprocessing in the collection or messaging tiers
- Delays in the actual processing of the event due to slowness of the streaming tier - under increased load

Stream Processing

This difference is called **"time skew"** and it has direct impact on some of the streaming computations

Some frameworks support one more type of "time":

Ingestion time: a hybrid of processing and event time. It assigns wall clock timestamps to records as soon as they arrive in the system (at the source) and continues processing with event time semantics based on the attached timestamps



@Marina Popova

Stream Processing: Requirements and Common Operations

Most common functionality required in Streaming systems:

- **Windowing operations**

Support for different functions performed over time windows or number of events; Examples: running averages; window-based statistics such as min/max/std etc.

- **Aggregations**

The ability to manage counters. Different types of aggregate operations are often required and supported
We will discuss Windowing and Aggregations operations in more details soon

- **Event-level enrichment**

The ability to modify a given event based on rules or content from an external system or some shared metadata

Stream Processing: Requirements and Common Operations

- **Event-level alerting/validation**

The ability to throw an alert based on a single event arriving in the system; possible in true streaming frameworks

- **Calculated metric based alerting**

Similar to the event-level alerting, but is based on a value of some computed metric; for example: alert me when max number of event for specific IP is higher than 1000

- **Persistence of transient data (state storage)**

The ability to store state during processing - for example, to calculate running averages and other functions

- **Higher-level functions**

Support for functions like sorting, grouping, partitioning and joining data sets

Stream Processing: Windowing Operations

Windowing Operations - Details

"window" - a base unit for windowing operations in streaming systems

- It is needed since there is no other natural boundaries for data - streaming data is endless
- Generally speaking, a window defines a *finite set of elements* on an unbounded stream.
- This set can be based on time, element counts, a combination of counts and time, or some custom logic to assign elements to windows

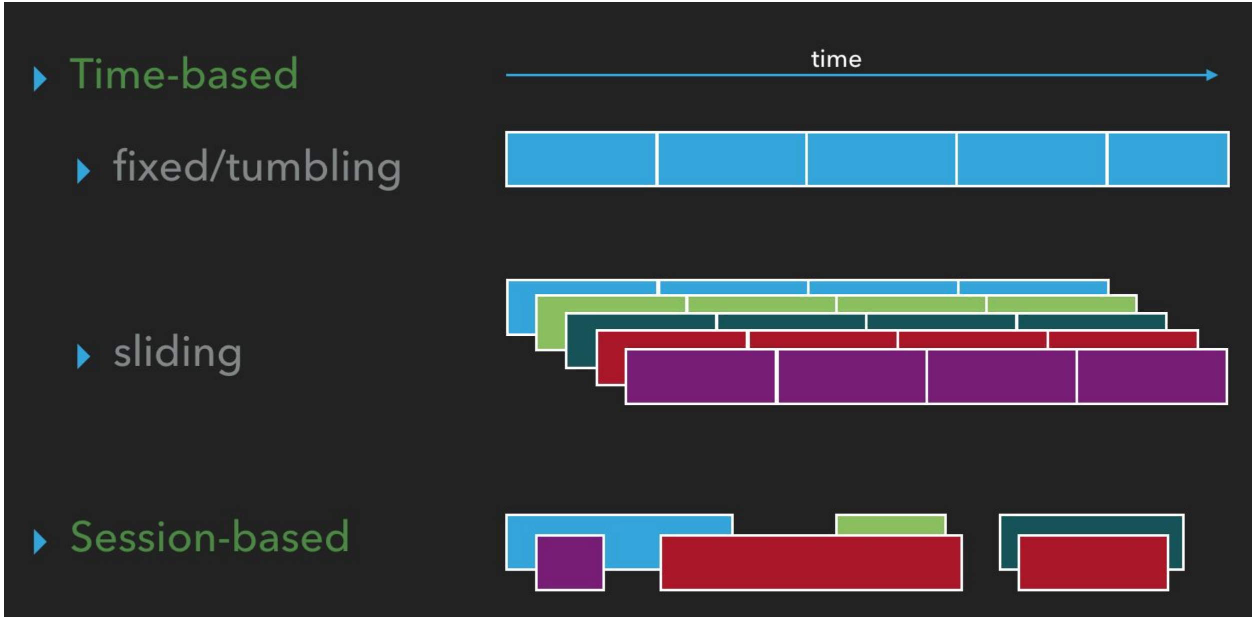
Attributes common for all windowing techniques:

- **Trigger policy**: defines the rules that dictate when it is time to process all data in the window - window is "complete"
- **Eviction policy**: defines the rules to decide if a specific event should be evicted from the window

Stream Processing: Windowing Operations

Types of windows:

Ref: https://softwaremill.com/windowing-in-big-data-streams-spark-flink-kafka-akka/

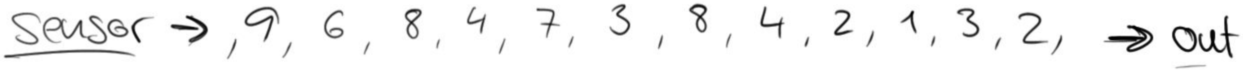


@Marina Popova

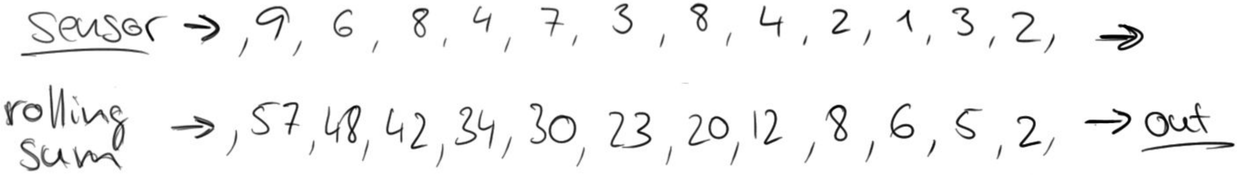
Stream Processing: Windowing Operations

Nice explanation of different windowing techniques: https://flink.apache.org/news/2015/12/04/Introducing-windows.html

Example: traffic sensor that counts every 15 seconds the number of vehicles passing a certain location. The resulting stream could look like:



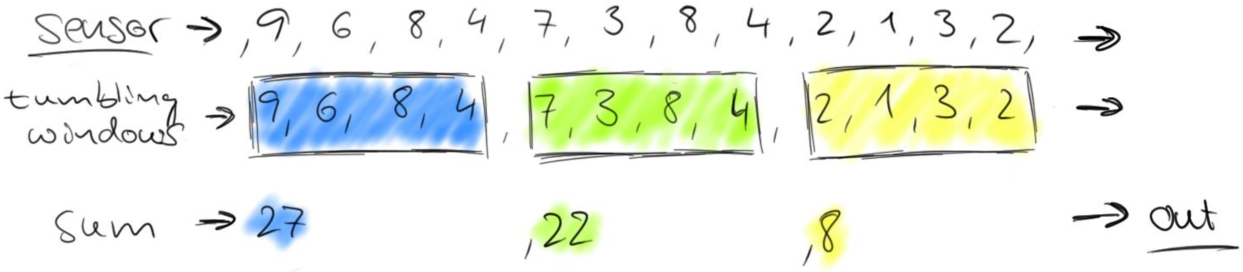
Without using any windows, we can only ask "Ad-hoc" queries like: get a rolling sum of all vehicles passed **so far**:
By computing rolling sums, we return for each input event an updated sum record. This would yield a new stream of partial sums:


sensor → , 9, 6, 8, 4, 7, 3, 8, 4, 2, 1, 3, 2, →
rolling sum → , 57, 48, 42, 34, 30, 23, 20, 12, 8, 6, 5, 2, → out

@Marina Popova

Stream Processing: Windowing Operations

Tumbling windows
In the above example, some important information such as variation over time is lost.
Hence, we might want to rephrase our question and ask for the number of cars that pass the location **every minute**.
This requires us to group the elements of the stream into finite sets, each set corresponding to 60 seconds.
This operation is called a *tumbling windows* operation:



Sensor → ,9, 6, 8, 4, 7, 3, 8, 4, 2, 1, 3, 2, →
tumbling windows → 9, 6, 8, 4 , 7, 3, 8, 4 , 2, 1, 3, 2 →
sum → 27 ,22 ,8 → out

@Marina Popova

Stream Processing: Windowing Operations

Main characteristics of Tumbling Windows

- Tumbling windows discretize a stream into **non-overlapping windows**.
- Eviction policy: window is full (60 sec in the example)
- Trigger policy can be of two types:
  - time-based (length of the window)
  - count-based
- The above example is the *time-based* tumbling window, with the trigger policy = 60 sec

Stream Processing: Windowing Operations

Sliding Window:

Trigger and Eviction policies are based on time:
Eviction: defined by the window length - duration of time that data is retained and available for processing

Trigger: is defined by the sliding interval: each time sliding interval is reached - our code will be notified to start processing the window

Big difference with the Tumbling windows:
The same event can fall into multiple windows!

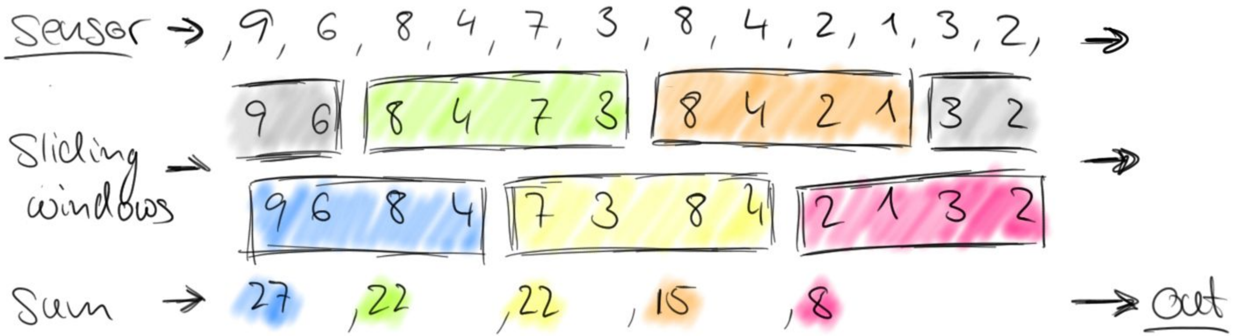
Using the example above, we may want to compute smoothed aggregates. For example, we can compute every thirty seconds the number of cars passed in the last minute:

Stream Processing: Windowing Operations

Using the earlier example:

compute smoothed aggregates.

For example, we can compute every thirty seconds the number of cars passed in the last minute:


Sensor → ,9, 6, 8, 4, 7, 3, 8, 4, 2, 1, 3, 2, →
9 6
8 4 7 3
8 4 2 1
3 2
Sliding windows →
9 6 8 4
7 3 8 4
2 1 3 2
Sum → 27 ,22 ,22 ,15 ,8 → Out

@Marina Popova

Stream Processing: Windowing Operations

Session windows:

- Can have various sizes and are defined basing on data, which should carry some session identifiers;
- Sessions are typically defined as periods of activity (e.g., for a specific user) terminated by a gap of inactivity.

There can be other variations of the windowing techniques, such as **hopping windows and snapshot windows**, but they are less commonly used.

Most commonly used streaming frameworks support most of the windowing types, especially in their latest releases.

Stream Processing: Windowing Operations

Out-of-order even handling

With event-time based stream processing - events can arrive out of order, for different reasons

We do want to still process them - but up to what point?

Cannot keep all windows forever!

At some point, a window has to be considered "done" and garbage collected.
This is handled by a mechanism called **watermarks.**
A watermark specifies that we assume that all events before X have been observed.

We will discuss watermarking and out-of-order event and state handling in details in the next lecture

Stream Processing: Algorithms and Techniques

Most often, Stream Processing is based on **incremental algorithms** instead of Batch algorithms

Incremental computations are much more complex to implement in the distributed systems, but for Stream processing it may be feasible due to :

- smaller scale of datasets (in comparison to Master datasets)
- tolerance for less accurate calculations (often)

If Stream processing is used as part of the Lambda architecture - additional factors are helping:

- the computed views are transient and will be replaced by batch views eventually

We will review the following most commonly used techniques for processing streaming data:

- Random Sampling
- Counting Distinct Elements
- Frequency
- Membership

Stream Processing: Algorithms and Techniques

Random Sampling

The main question: How to take a random test from a data that is moving very fast, never stops and does not fit into memory?
Use cases: you are interested in trends/statistical results more than in the exact results

Commonly used Algorithms: Reservoir Sampling
Ref: http://www.geeksforgeeks.org/reservoir-sampling/

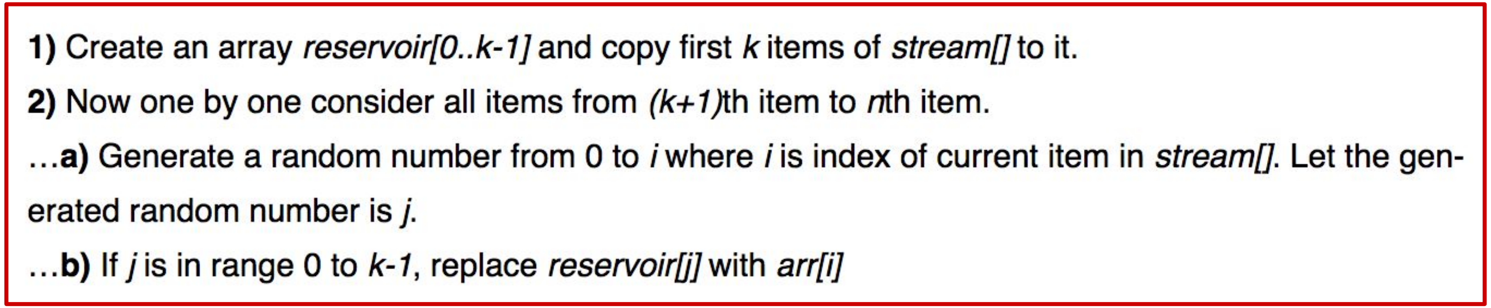
From Wikipedia:
Reservoir sampling is a family of randomized algorithms for randomly choosing ***k*** samples from a list of ***n*** items, where *n* is either a very large or unknown number.
Typically ***n* is large enough that the list doesn't fit into main memory**.
For example, a list of search queries in Google and Facebook.

Stream Processing: Algorithms and Techniques

The main idea is that we can hold a pre-determined number of stream values (the reservoir) and when a new one arrives - we can probabilistically determine:

- whether to add it to our collection
- or to randomly select one of the values already in the reservoir as the random sample

The algorithm:



1) Create an array *reservoir[0..k-1]* and copy first *k* items of *stream[]* to it.

2) Now one by one consider all items from *(k+1)*th item to *n*th item.

…**a)** Generate a random number from 0 to *i* where *i* is index of current item in *stream[]*. Let the generated random number is *j*.

…**b)** If *j* is in range 0 to *k-1*, replace *reservoir[j]* with *arr[i]*

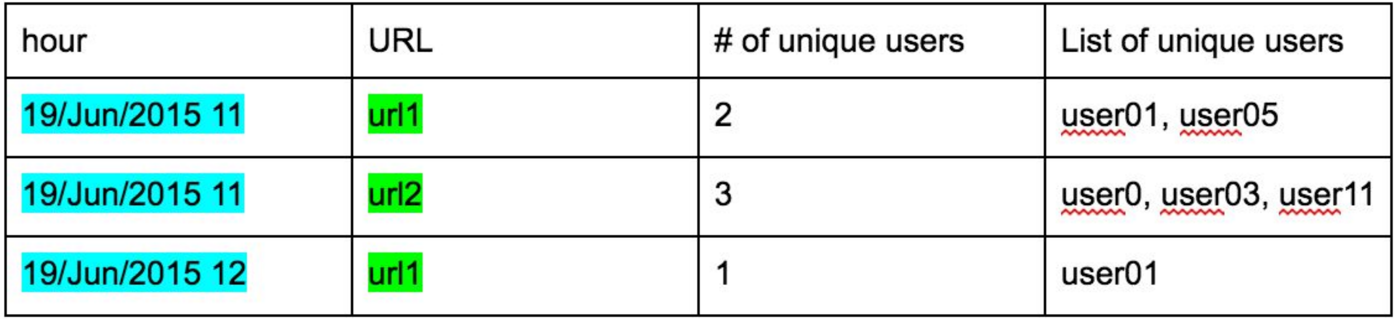
More fun techniques for sampling:
https://www.analyticsvidhya.com/blog/2019/09/data-scientists-guide-8-types-of-sampling-techniques/

@Marina Popova

Stream Processing: Algorithms and Techniques

Counting Distinct Elements - Uniques Counts

Our approach in Batch processing:
create Batch Views that store the list of unique userIDs:



hour	URL	# of unique users	List of unique users
19/Jun/2015 11	url1	2	user01, user05
19/Jun/2015 11	url2	3	user0, user03, user11
19/Jun/2015 12	url1	1	user01

Stream Processing: Algorithms and Techniques

Pros:

- Can calculate unique visitor counts over any time range - with 100% accuracy
- If the cardinality of users is low (only a few distinct users) but the number of clicks is high (hundreds per user) - we are still consolidating the data significantly

Cons:

- What if there are millions of unique visitors per hour ??

Probabilistic approach:

HyperLogLog algorithm: calculate an approximate count of distinct elements == CARDINALITY

Good references for HyperLogLog explanation:

http://www.baeldung.com/java-hyperloglog

http://moderndescartes.com/essays/hyperloglog

papers LogLog counting of large cardinalities by Durand-Flajolet, and HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm by Flajolet et al.

Stream Processing: Algorithms and Techniques

The HyperLogLog (HLL) data structure is a **probabilistic data structure** used to estimate the cardinality of a data set.
It requires ~1KB of data to estimate set cardinalities of up to one billion elements, with a max error rate of 2%

http://stefansavev.com/blog/an-intuitive-explanation-of-hyperloglog-algorithm-for-approximate-distinct-count/
The main idea:
if you, observing a stream of random integers, see an integer which binary representation starts with some known prefix, there is a higher chance that the cardinality of the stream is 2^(size of the prefix).

That is, in a random stream of integers, ~50% of the numbers (in binary) starts with "1", 25% starts with "01", 12,5% starts with "001". This means that if you observe a random stream and see a "001" (size = 3) , there is a higher chance that this stream has a cardinality of 8 (8 = 2^3)

(The prefix "00..1" has no special meaning. It's there just because it's easy to find the most significant bit in a binary number in most processors)

Stream Processing: Algorithms and Techniques

Of course, if you observe just one integer, the chance this value is wrong is high.
That's why the algorithm divides the stream in **"m" independent substreams** and keeps the maximum length of an observed "00...1" prefix of each substream.

Then, *it estimates the final value by taking the mean value of each substream*.

How do you get this 'integer/bit string' for each element? You can map any string (or any object) to a long number or equivalently to a bit string by hashing. Well-chosen hash functions will ensure that distinct strings (objects) will produce distinct hashes in most cases

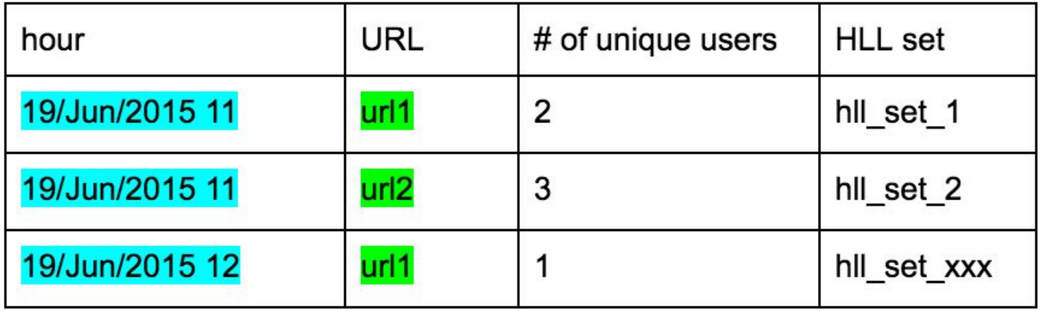
One important feature of this HLL: **results of two calculations can be joined**:
When we take the union of two HLLs created from distinct data sets and measure its cardinality, we will get the same error threshold for the union that we would get if we had used a single HLL and calculated the hash values for all elements of both data sets from the beginning.

Which lands it well into the MR processing pattern

Stream Processing: Algorithms and Techniques

What HLL algorithm interface usually provides:
add(element)
merge(HLL otherSet)

Real-Time (and Batch) View with HLLs:



hour	URL	# of unique users	HLL set
19/Jun/2015 11	url1	2	hll_set_1
19/Jun/2015 11	url2	3	hll_set_2
19/Jun/2015 12	url1	1	hll_set_xxx

When should you use HLL for uniques?

- When cardinality of the dataset is large
- Approximate values are Ok
- There is NO chance of semantic meaning change over time! HLL does not know what elements are within it ...

Stream Processing: Algorithms and Techniques

Frequency: *How many times has stream element X occured?*

Most popular algorithm: **Count-Min Sketch**

Good explanation: https://stackoverflow.com/questions/6811351/explaining-the-count-sketch-algorithm
The simplest explanation I found:
https://redislabs.com/blog/count-min-sketch-the-art-and-science-of-estimating-stuff/

Main idea:

- instead of tracking each unique element(sample) independently, it uses its hash value
- The hash value of an element is used as the index to a constant-sized (parameter d) **array of counters** - when and element 'hits' an index, the corresponding counter is increased
- Not one, but several (the parameter w) different hash functions and their respective arrays are used
- Why? to handle hash collisions for different elements (samples)
- How? by picking the minimum value out of all relevant counters for the sample

Stream Processing: Algorithms and Techniques

Lets illustrate this algorithm by example:

we’ll use small parameter values:
w = 3: we’ll use three hash functions – named h1, h2 and h3 respectively
d = 4: size of one hash function's counter array

Thus, to store the sketch’s counters we’ll use a 3×4 array with a total of 12 elements initialized to 0.

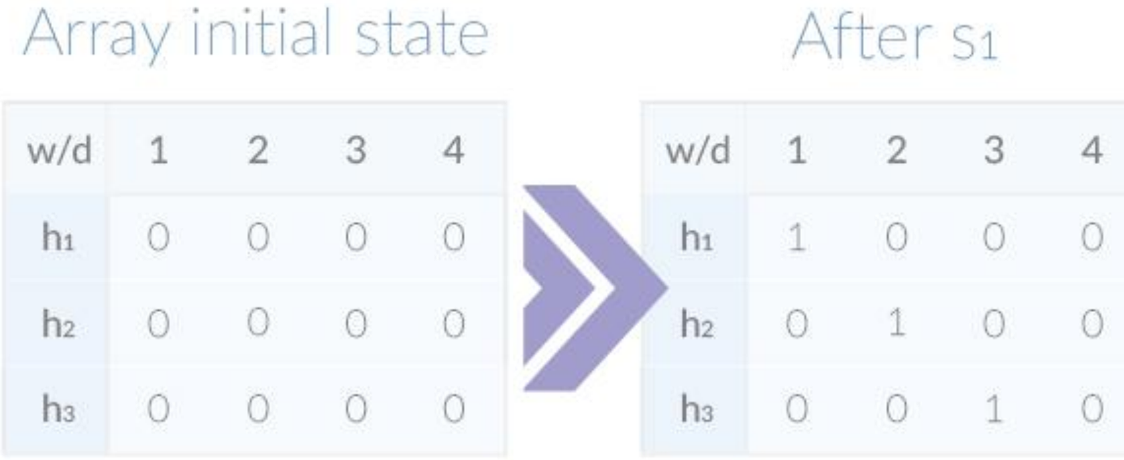
Now we can examine what happens when samples are added to the sketch.
Let’s assume samples arrive one by one and that the hashes for the first sample, s1, are:
h1(s1) = 1, h2(s1) = 2 and h3(s1) = 3.

Stream Processing: Algorithms and Techniques

w = 3: number of hash functions (h1, h2, h3)
d = 4: size of one hash function's counter array

first sample, s1:
h1(s1) = 1
h2(s1) = 2
h3(s1) = 3.

To record s1 in the sketch we'll increment each hash function's counter at the relevant index by 1.
The following table shows the array's initial and current states:



Array initial state

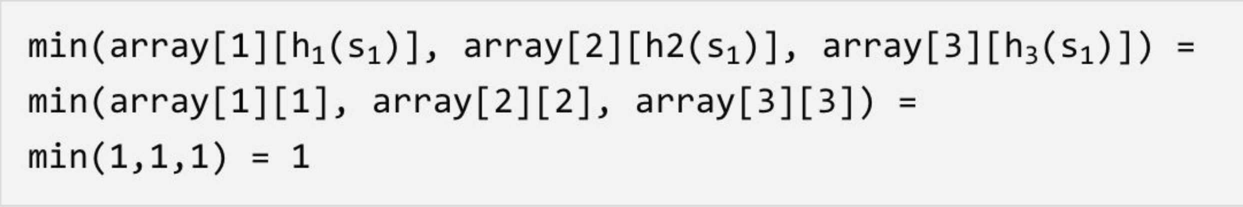
w/d	1	2	3	4
h_1	0	0	0	0
h_2	0	0	0	0
h_3	0	0	0	0

After s_1

w/d	1	2	3	4
h_1	1	0	0	0
h_2	0	1	0	0
h_3	0	0	1	0

Stream Processing: Algorithms and Techniques

The number of observations for a sample (i.e. number of time we saw this element) is the **minimum of all its counters**, so for s1 it is obtained by:



$$\mathtt{min(array[1][}h_1(s_1)\mathtt{)],\ array[2][h2(}s_1\mathtt{)],\ array[3][}h_3(s_1)\mathtt{])\ =}$$

$$\mathtt{min(array[1][1],\ array[2][2],\ array[3][3])\ =}$$

$$\mathtt{min(1,1,1)\ =\ 1}$$

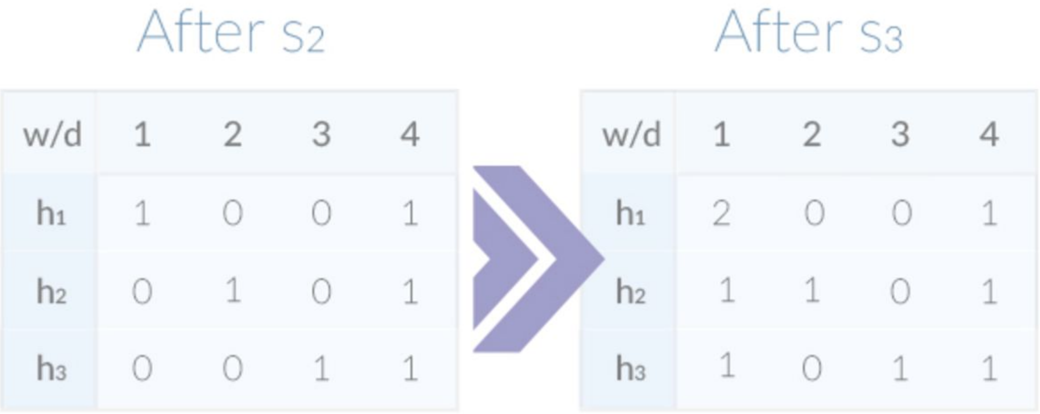
number of observations == how many time have I seen this element??

Stream Processing: Algorithms and Techniques

Lets add our next elements, sample s2 and s3, with the following hash values:

h1(s2) = 4
h2(s2) = 4
h3(s2) = 4

h1(s3) = 1
h2(s3) = 1
h3(s3) = 1



After s_2

w/d	1	2	3	4
h_1	1	0	0	1
h_2	0	1	0	1
h_3	0	0	1	1

After s_3

w/d	1	2	3	4
h_1	2	0	0	1
h_2	1	1	0	1
h_3	1	0	1	1

Stream Processing: Algorithms and Techniques

In our example, almost all of the samples’ hashes map to unique counters, with the one exception:

- collision of h1(s1) and h1(s3)

Because both hashes are the same, h1‘s 1st counter now holds the value **2**.
Since the sketch picks the **minimum of all counters**, the queries for s1 and s3 still return the correct result of 1.

Eventually, however, once enough collisions have occurred, the queries’ results will become less accurate.

CMS’ two parameters – w and d – determine its **space/time requirements** as well as the **probability** and **maximal value** of its error.
For example, a sketch that has 0.01% error rate at probability of 0.01% is created using 10 hash functions and 2000-counter arrays.

Stream Processing: Algorithms and Techniques

Membership: *Has this stream element **ever** occurred in the stream before?*

The data structures and algorithms designed for this specific task are **Bloom Filters**.
As other algorithms, it is probabilistically bound and this probability can be configured

Bloom filters are a way of compactly representing a set of items

Ref: https://prakhar.me/articles/bloom-filters-for-dummies/

Properties of Bloom Filter algorithm:

- Very memory efficient
- False positives are possible (result: element IS in a set - MAY be false)
- False negatives are NOT possible (result: element is NOT in a set - always TRUE)

Stream Processing: Algorithms and Techniques

Algorithm:

An *empty Bloom filter* is a bit array of ***m*** bits, all set to 0.

There must also be ***k*** different hash functions defined, each of which maps or hashes some set element to one of the *m* array positions, generating a uniform random distribution.

Typically, *k* is a constant, much smaller than *m*, which is proportional to the number of elements to be added; the precise choice of *k* and the constant of proportionality of *m* are determined by the intended false positive rate of the filter.

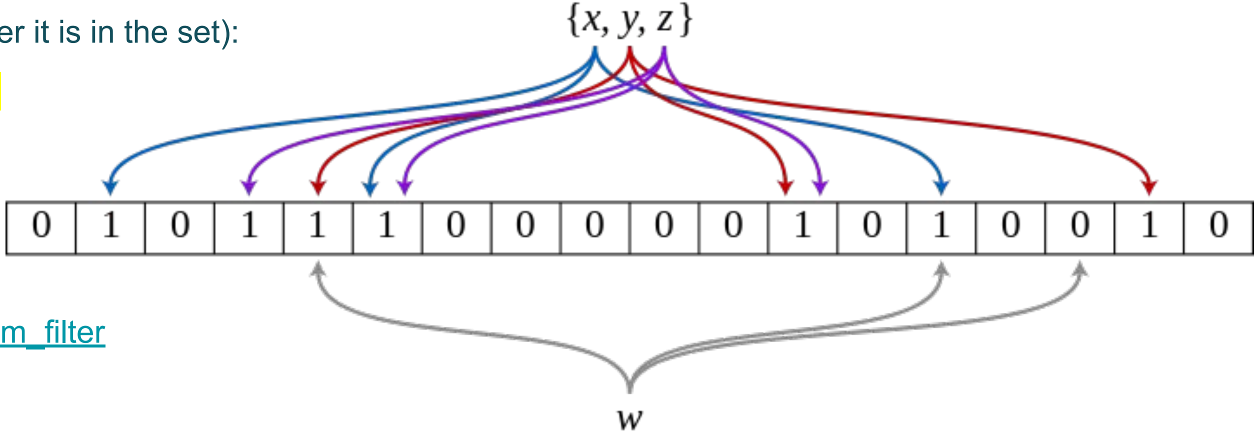
To *add* an element, feed it to each of the *k* hash functions to get *k* array positions. Set the bits at all these positions to 1.

Stream Processing: Algorithms and Techniques

To *query* for an element (test whether it is in the set):

feed it to each of the *k* hash functions

to get *k* array positions.



Ref: https://en.wikipedia.org/wiki/Bloom_filter

If any of the bits at these positions is 0, the element is definitely **NOT** in the set – if it were, then all the bits would have been set to 1 when it was inserted - that's the element **w** in the example

If all are 1, then either the element is in the set, or the bits have by chance been set to 1 during the insertion of other elements, resulting in a false positive. (x, y, z)

In a simple Bloom filter, there is no way to distinguish between the two cases, but more advanced techniques can address this problem.

@Marina Popova

Stream Processing: Algorithms and Techniques

The simplest explanation with demo:
https://llimllib.github.io/bloomfilter-tutorial/

Important decisions that affect efficiency and precision of the algorithm:

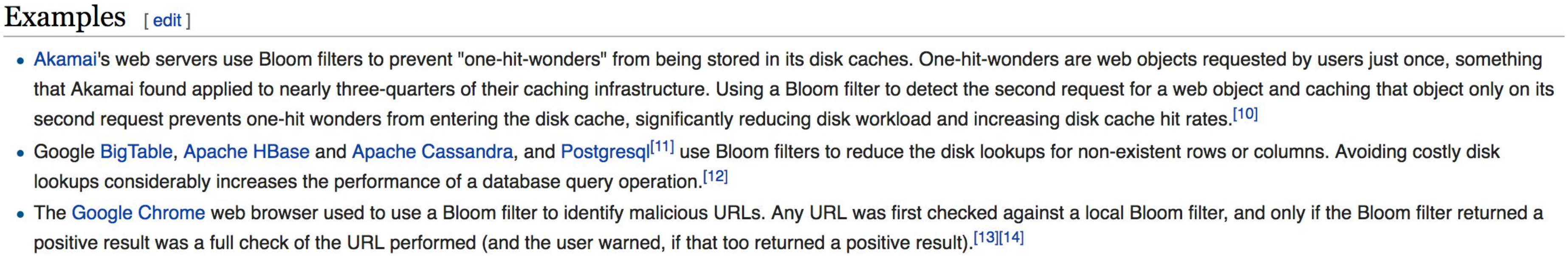
- m - size of the bit array
- K - number of hash functions
- Type of hash functions - fast, independent and uniformly distributed. Examples of fast, simple hashes that are independent enough include **murmur**, the **fnv** series of hashes, and **HashMix**

Stream Processing: Algorithms and Techniques

More properties of Bloom Filters:

- Unlike a standard hash table, a **Bloom filter of a fixed size can represent a set with an arbitrarily large number of elements**; adding an element never fails due to the data structure "filling up". However, the false positive rate increases steadily as elements are added until all bits in the filter are set to 1, at which point ***all* queries** yield a positive result.
- **Union** and intersection of Bloom filters with the same size and set of hash functions can be implemented with bitwise OR and AND operations respectively. The union operation on Bloom filters is lossless in the sense that the resulting Bloom filter is the same as the Bloom filter created from scratch using the union of the two sets.

Examples of Bloom Filter applications: https://en.wikipedia.org/wiki/Bloom_filter#Examples

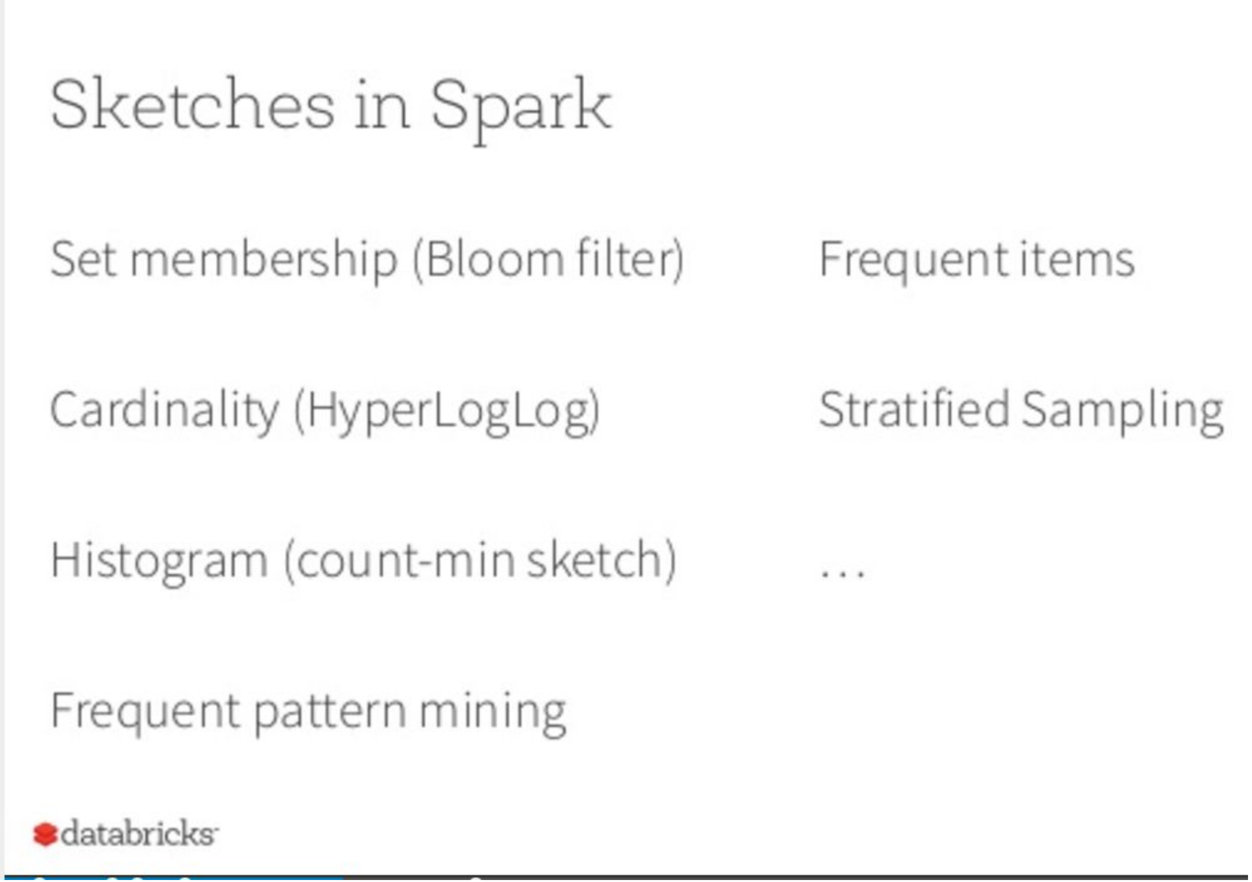


Examples [edit]

- Akamai's web servers use Bloom filters to prevent "one-hit-wonders" from being stored in its disk caches. One-hit-wonders are web objects requested by users just once, something that Akamai found applied to nearly three-quarters of their caching infrastructure. Using a Bloom filter to detect the second request for a web object and caching that object only on its second request prevents one-hit wonders from entering the disk cache, significantly reducing disk workload and increasing disk cache hit rates.[10]
- Google BigTable, Apache HBase and Apache Cassandra, and Postgresql[11] use Bloom filters to reduce the disk lookups for non-existent rows or columns. Avoiding costly disk lookups considerably increases the performance of a database query operation.[12]
- The Google Chrome web browser used to use a Bloom filter to identify malicious URLs. Any URL was first checked against a local Bloom filter, and only if the Bloom filter returned a positive result was a full check of the URL performed (and the user warned, if that too returned a positive result).[13][14]

Stream Processing

What Spark supports
(according to their presentation
at NYC Strata 2015):



Sketches in Spark

- Set membership (Bloom filter)
- Cardinality (HyperLogLog)
- Histogram (count-min sketch)
- Frequent pattern mining
- Frequent items
- Stratified Sampling
- ...

databricks

Stream Processing: Architecture

Stream Processing: high-level architecture

While it might be possible to run stream processing application on one machine only - it will quickly hit the same scaling limitations as batch systems. Why?

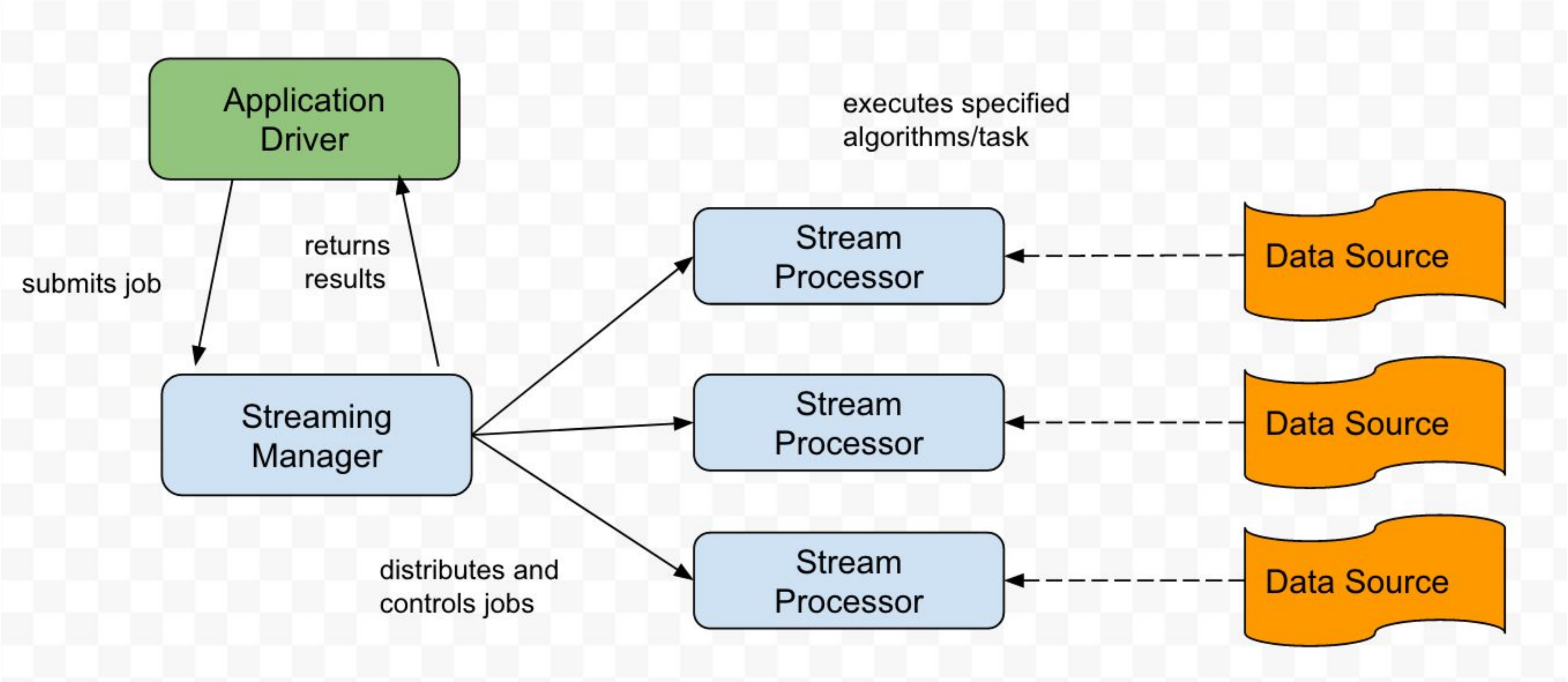
- amount of data
- speed of data ingestion vs processing rate

Thus, streaming applications have to be distributed just as batch processing applications.

Common Architecture view - very similar to the Batch application architecture:

- Application Driver
- Streaming Manager
- Stream Processor
- Data Sources

Stream Processing: Architecture



@Marina Popova

Stream Processing: Architecture

Key Features of Streaming systems

- Message delivery semantics
    - At-most-once
        - The easiest to support
    - At-least-once
        - Streaming Manager has to keep track of every message sent to Stream Processors, with ACK
        - In case of failure to process/deliver - can resend
        - Streaming job must be idempotent
    - Exactly-once
        - Streaming Manager has to do the above + de-dupe message
        - Streaming job can be non-idempotent

Stream Processing: Architecture

Key Features of Streaming systems

- State management
  - In-memory or persistent (distributed) ?
  - Ability to recover state in the face of failure
- Fault Tolerance - in the face of data loss of resources/servers loss; common approaches:
  - State-machine: manager replicates the streaming job's to N independent nodes and coordinates the replicas by sending the same input to all
  - Rollback recovery/ checkpointing: Stream Processor periodically packages the state of its computations as a checkpoint, and copies it to a different Stream Processor node or some other storage

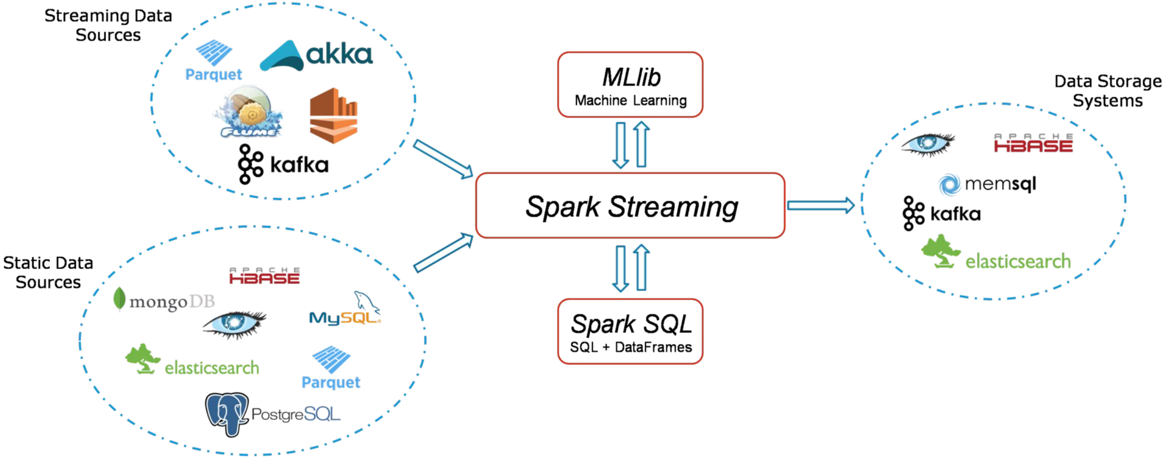
Major Streaming Frameworks:

- Apache Spark Streaming
- Apache Flink
- Apache Storm
- Apache Samza
- Kafka Streams

Spark Streaming

http://spark.apache.org/docs/latest/streaming-programming-guide.html

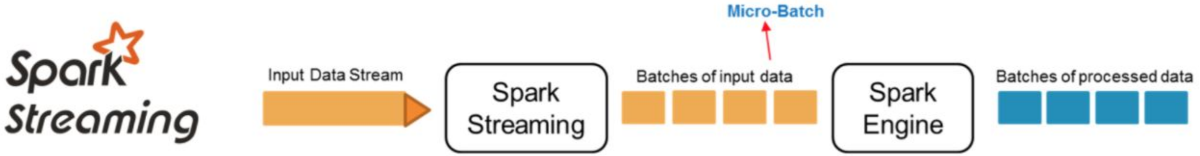
Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.
It can also use the static data sources for initial data load.
A high-level Spark Streaming workflow may look like this:



@Marina Popova

Spark Streaming

- Instead of processing the streaming data one record at a time, Spark Streaming **discretizes the streaming data into small micro-batches** (configurable).
- In other words, Spark Streaming's **Receivers** accept data in parallel and buffer it in the memory of Spark's workers nodes.
- Then the Spark engine runs short tasks (tens of milliseconds) to process the batches and output the results to other systems.
- Each **batch of data is represented as an RDD**. This allows the streaming data to be processed using any Spark code or library
- The fundamental stream unit is **DStream** which is basically a series of the batch RDDs.

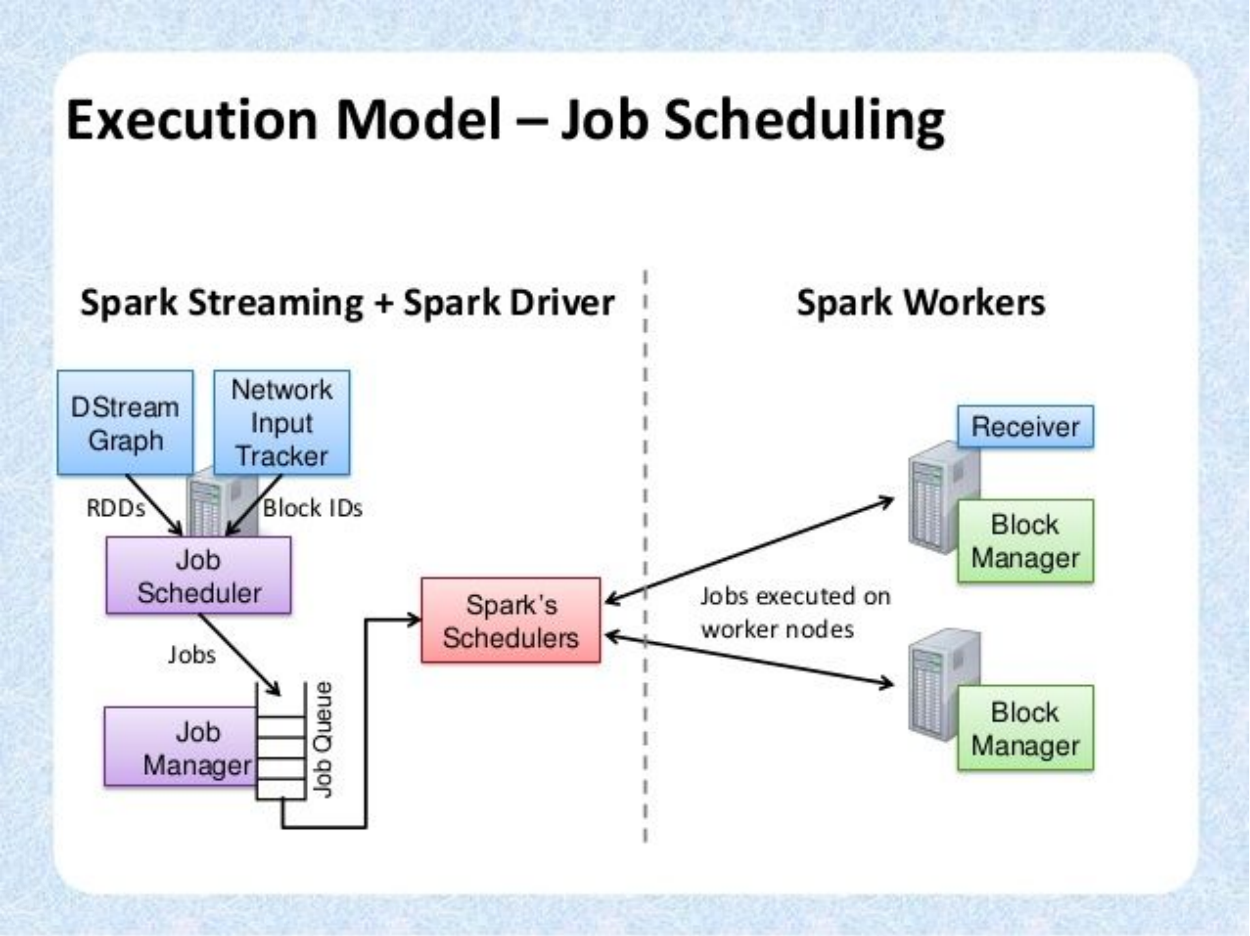


@Marina Popova

Spark Streaming

The High-level Spark Streaming Architecture

very similar to the Spark Batch Architecture



Execution Model – Job Scheduling

Spark Streaming + Spark Driver

Spark Workers

@Marina Popova

Spark Streaming

Main Spark Streaming Components

StreamingContext

- StreamingContext is a wrapper around the SparkContext
- The main addition to SparkContext: batchDuration parameter which determines the goal batch interval
- StreamingContext consumes a stream of data and creates an InputDStream - which is a wrapper around an RDD
- default implementations of sources: fileInputStream, Kafka, Twitter, Akka Actor and ZeroMQ

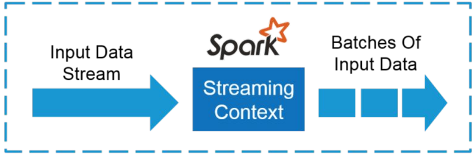
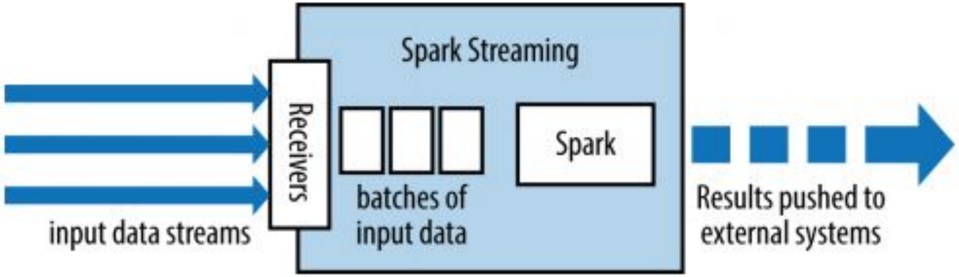


Figure: *Spark Streaming Context*

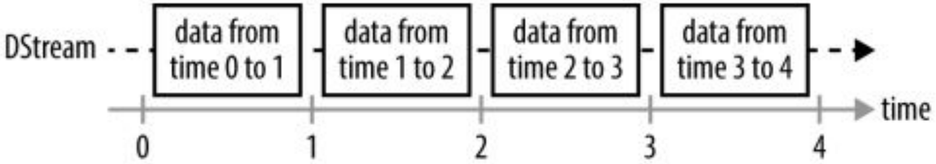
Figure: *Default Implementation Sources*

Spark Streaming

High-level architecture of Spark Streaming



DStream as a continuous series of RDDs



@Marina Popova

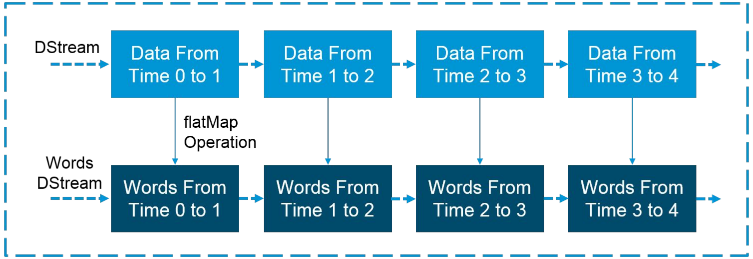
Spark Streaming

DStreams

Discretized Stream (DStream) is the basic abstraction provided by Spark Streaming.
It is a continuous stream of data. It is received from a data source or a processed data stream generated by transforming the input stream.
Internally, a DStream is represented by a continuous series of RDDs and each RDD contains data from a certain interval.

Main difference between RDD and Dstream:

- RDD is an immutable distributed collection
- DStream is an immutable distributed collection of RDDs in a context of the batch window



@Marina Popova

Spark Streaming

More on InputDStreams

InputDStreams are DStreams representing the stream of input data received from streaming sources.

Every InputDStream is associated with a **Receiver** object which receives the data from a source and stores it in Spark's memory for processing.

Spark Streaming provides two categories of built-in streaming sources.

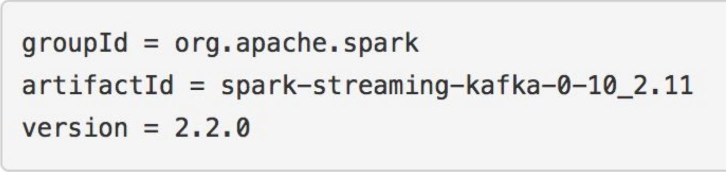
- ***Basic sources*:** Sources directly available in the StreamingContext API. Examples: file systems, and socket connections.
- ***Advanced sources*:** Sources like Kafka, Flume, Kinesis, etc. are available through extra utility classes. These require linking against extra dependencies

If you want to receive **multiple streams of data in parallel** in your streaming application, you can create multiple input DStreams
This will create multiple receivers which will simultaneously receive multiple data streams.
Spark Streaming application needs to be allocated enough cores to process the received data, as well as to run the receiver(s).

Spark Streaming

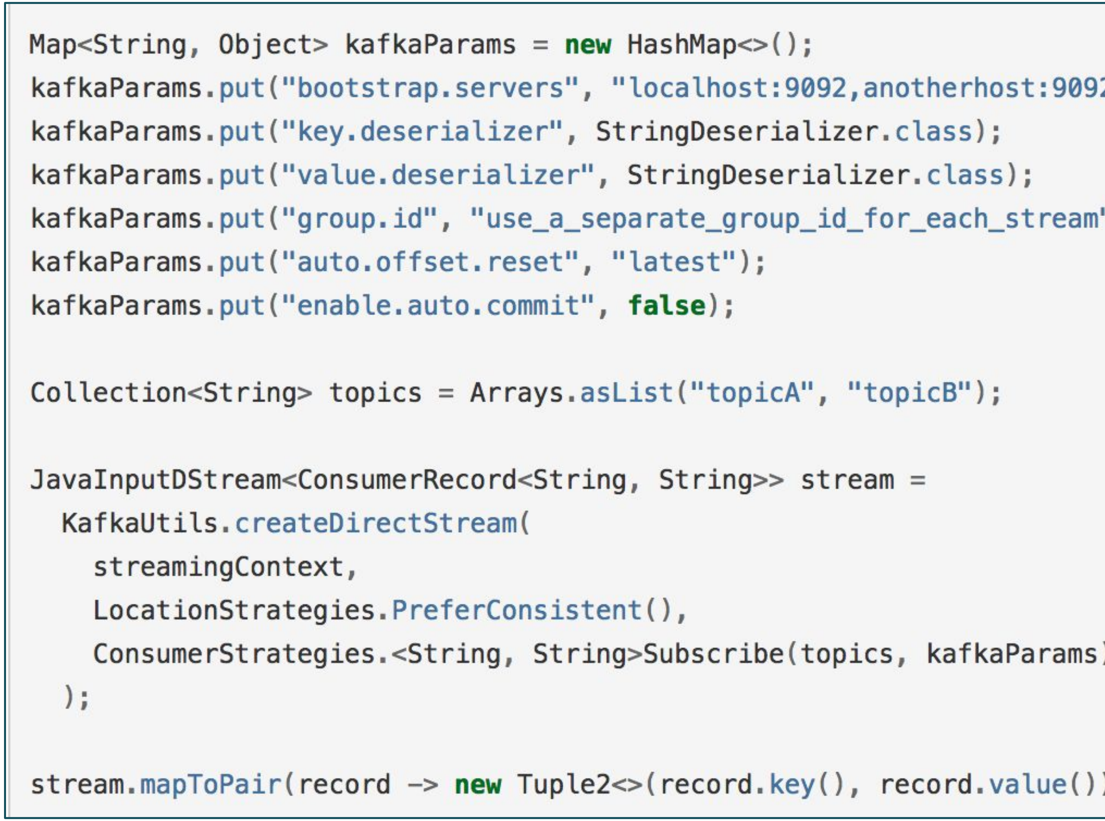
Input DStreams - Kafka source

add dependencies:



```
groupId = org.apache.spark
artifactId = spark-streaming-kafka-0-10_2.11
version = 2.2.0
```

LocationStrategies.PreferConsistent:
will distribute partitions evenly
across available executors



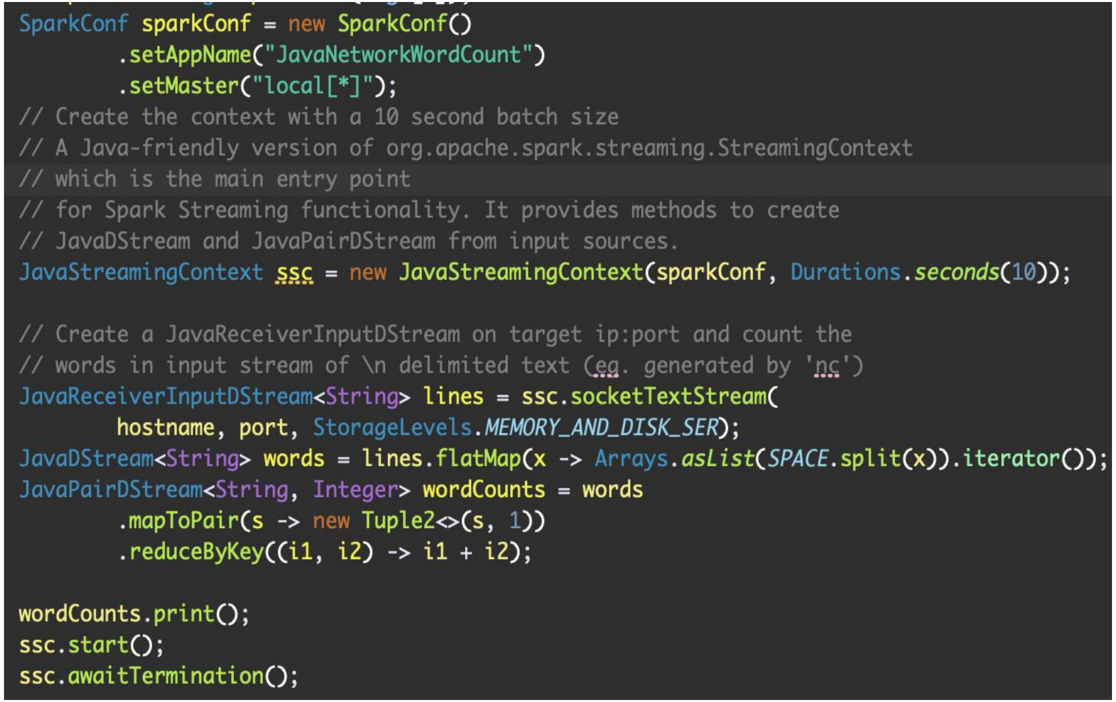
```
Map<String, Object> kafkaParams = new HashMap<>();
kafkaParams.put("bootstrap.servers", "localhost:9092,anotherhost:9092
kafkaParams.put("key.deserializer", StringDeserializer.class);
kafkaParams.put("value.deserializer", StringDeserializer.class);
kafkaParams.put("group.id", "use_a_separate_group_id_for_each_stream"
kafkaParams.put("auto.offset.reset", "latest");
kafkaParams.put("enable.auto.commit", false);

Collection<String> topics = Arrays.asList("topicA", "topicB");

JavaInputDStream<ConsumerRecord<String, String>> stream =
  KafkaUtils.createDirectStream(
    streamingContext,
    LocationStrategies.PreferConsistent(),
    ConsumerStrategies.<String, String>Subscribe(topics, kafkaParams
  );

stream.mapToPair(record -> new Tuple2<>(record.key(), record.value()
```

https://spark.apache.org/docs/latest/streaming-kafka-0-10-integration.html

@Marina Popova

Spark Streaming - implementation steps

- Initialize Spark streaming context
- Define the input sources by creating input DStreams.
- Define the streaming computations by applying transformation and output operations to DStreams.
- Start receiving data and processing it using streamingContext.start().
- Wait for the processing to be stopped (manually or due to any error) using streamingContext.awaitTermination().
- The processing can be manually stopped using streamingContext.stop().



```
SparkConf sparkConf = new SparkConf()
        .setAppName("JavaNetworkWordCount")
        .setMaster("local[*]");
// Create the context with a 10 second batch size
// A Java-friendly version of org.apache.spark.streaming.StreamingContext
// which is the main entry point
// for Spark Streaming functionality. It provides methods to create
// JavaDStream and JavaPairDStream from input sources.
JavaStreamingContext ssc = new JavaStreamingContext(sparkConf, Durations.seconds(10));

// Create a JavaReceiverInputDStream on target ip:port and count the
// words in input stream of \n delimited text (eg. generated by 'nc')
JavaReceiverInputDStream<String> lines = ssc.socketTextStream(
        hostname, port, StorageLevels.MEMORY_AND_DISK_SER);
JavaDStream<String> words = lines.flatMap(x -> Arrays.asList(SPACE.split(x)).iterator());
JavaPairDStream<String, Integer> wordCounts = words
        .mapToPair(s -> new Tuple2<>(s, 1))
        .reduceByKey((i1, i2) -> i1 + i2);

wordCounts.print();
ssc.start();
ssc.awaitTermination();
```

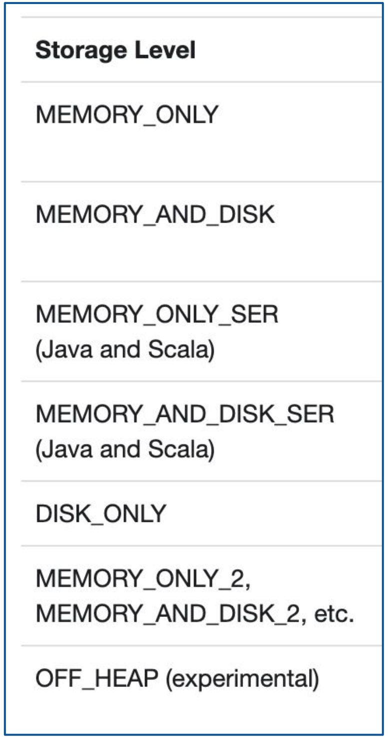
@Marina Popova

Spark Streaming

Spark Streaming simple count example:

This example counts the lines sent every 1 second to Spark Streaming that contain the word *the*:

```
val sc = new SparkContext(sparkConf)
val ssc = new StreamingContext(sc, Seconds(1))
val rawStream = ssc.socketTextStream(host, port,
StorageLevel.MEMORY_ONLY_SER)
rawStream.filter(_.contains("the")).count().print()
ssc.start()
```



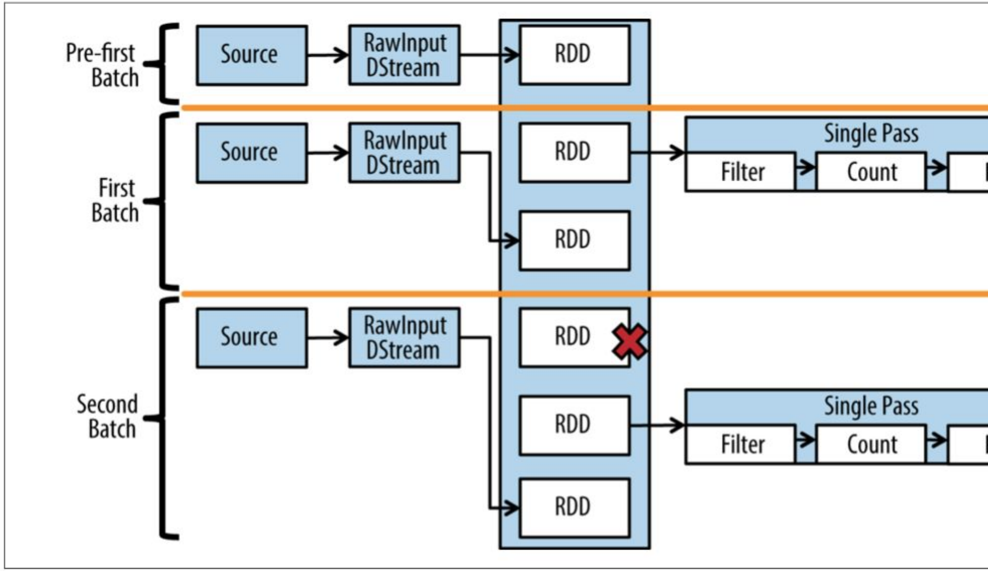
Storage Level
MEMORY_ONLY
MEMORY_AND_DISK
MEMORY_ONLY_SER (Java and Scala)
MEMORY_AND_DISK_SER (Java and Scala)
DISK_ONLY
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.
OFF_HEAP (experimental)

More on persistence of RDDs:
https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#rdd-persistence

Spark Streaming

Points to note:

- The InputDStream is always collecting data and putting it into a new RDD to be processed.
- The DStream that gets executed on is no longer taking in data
- It is an immutable DStream at the point of execution.
- Execution runs on the same engine as Spark
- RDDs within the context of the DStreams are removed when they are no longer needed



Spark Streaming

Operations on DStreams

- Any operation applied on a DStream translates to operations on the underlying RDDs
- Operations can involve more than one DStream

Types of Operations supported by DStreams:

- Stream operations (similar to regular RDD operations) - Stateless!
    - Transformations (lazy) - allow the data from the input DStream to be modified similar to RDDs
        - map, filter, reduce
    - Output operations (trigger execution)
- Windowed Operations - specific to DSstreams - can work across batches
- Stateful Operations - can use results/state from previous batches

Spark Streaming

Transformations on DStreams

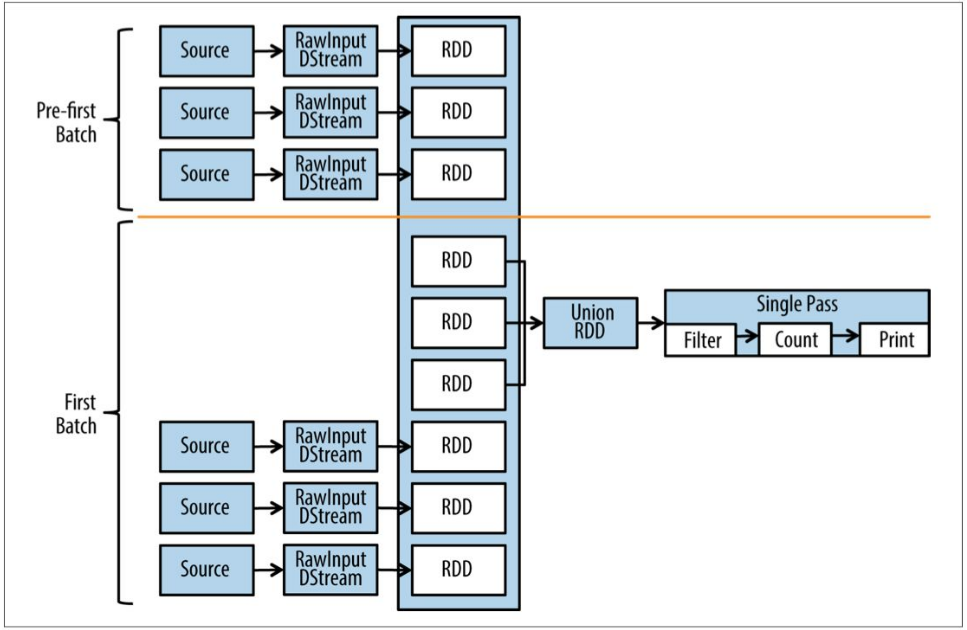
filter(*func*)	Return a new DStream by selecting only the records of the source DStream on which *func* returns true.
map(*func*)	Return a new DStream by passing each element of the source DStream through a function *func*.
reduce(*func*)	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function *func* (which takes two arguments and returns one). The function should be associative and commutative so that it can be computed in parallel.
countByValue()	When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream
join(*otherStream*, [*numTasks*])	When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.
updateStateByKey(*func*)	Return a new "state" DStream where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key. This can be used to maintain arbitrary state data for each key.

@Marina Popova

Spark Streaming

Example:
Join three InputDStreams before we do our filter and counts.

This demonstrates the process of taking multiple input streams and performing a union so we can treat it as a single stream



```
val sc = new SparkContext(sparkConf)
val ssc = new StreamingContext(sc, Seconds(1))
val rawStream1 =
  ssc.socketTextStream(host1, port1, StorageLevel.MEMORY_ONLY_SER_2)
val rawStream2 =
  ssc.socketTextStream(host2, port2, StorageLevel.MEMORY_ONLY_SER_2)
val rawStream3 =
  ssc.socketTextStream(host3, port3, StorageLevel.MEMORY_ONLY_SER_2)
val rawStream = rawStream1.union(rawStream2).union(rawStream3)
rawStream.filter(_.contains("the")).count().print()
ssc.start()
```

@Marina Popova

Spark Streaming

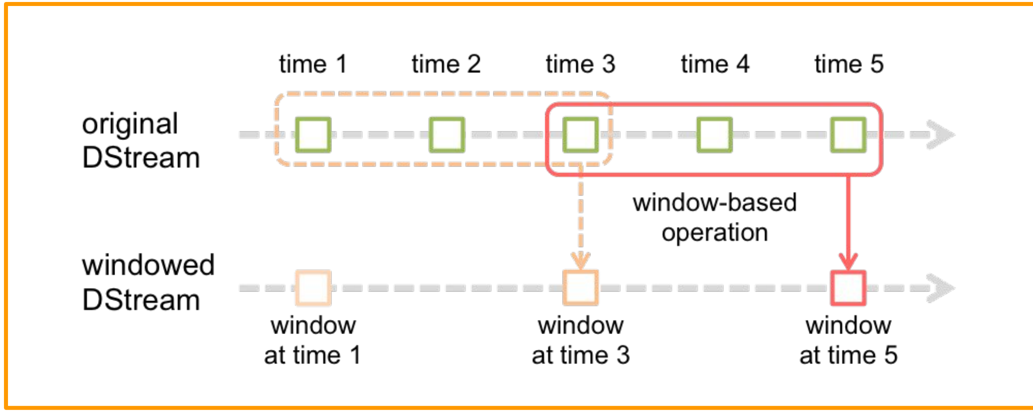
Windowed Operations

allow you to apply transformations over a sliding window of data

any window operation needs to specify two parameters.

- *window length* - The duration of the window (3 in the figure).
- *sliding interval* - The interval at which the window operation is performed (2 in the figure).

These two parameters must be multiples of the batch interval of the source DStream (1 in the figure).



Examples:

- window()
- countByWindow()
- reduceByWindow()
- reduceByKeyAndWindow()

Spark Streaming

Stateful Operations

- updateStateByKey() - iterates over all keys in the state
- mapWithState() - only works on key-value pairs that have values in the current batch
- Spark Structured Streaming - latest APIs

Spark Streaming

UpdateStateByKey Operation

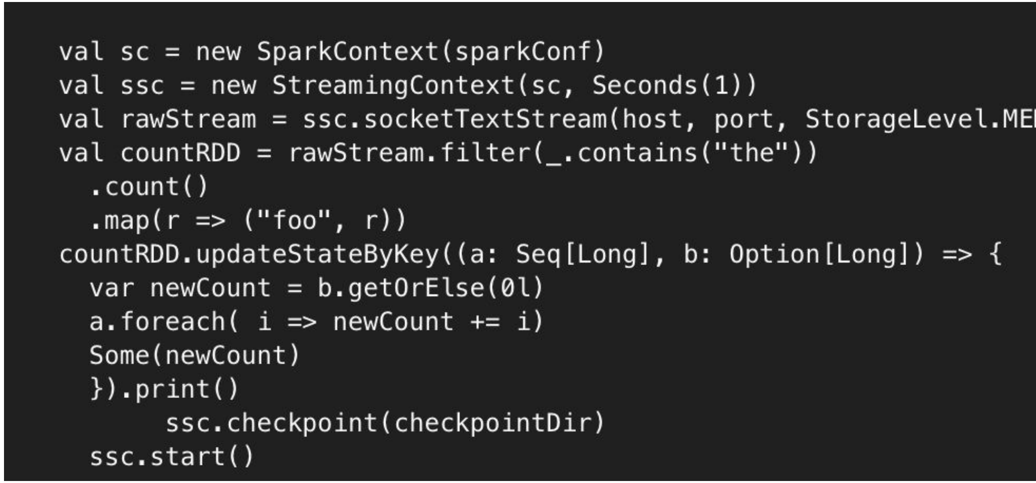
The updateStateByKey operation allows you to maintain arbitrary state while continuously updating it with new information. To use this, you will have to do a few steps:

- Define the state - The state can be an arbitrary data type.
- Define the state update function - Specify with a function how to update the state using the previous state and the new values from an input stream
- Configure checkpoint directory: ssc.checkpoint(checkpointDir)

In every batch, Spark will apply the state update function for all existing keys, regardless of whether they have new data in a batch or not. If the update function returns None then the key-value pair will be eliminated.

Spark Streaming

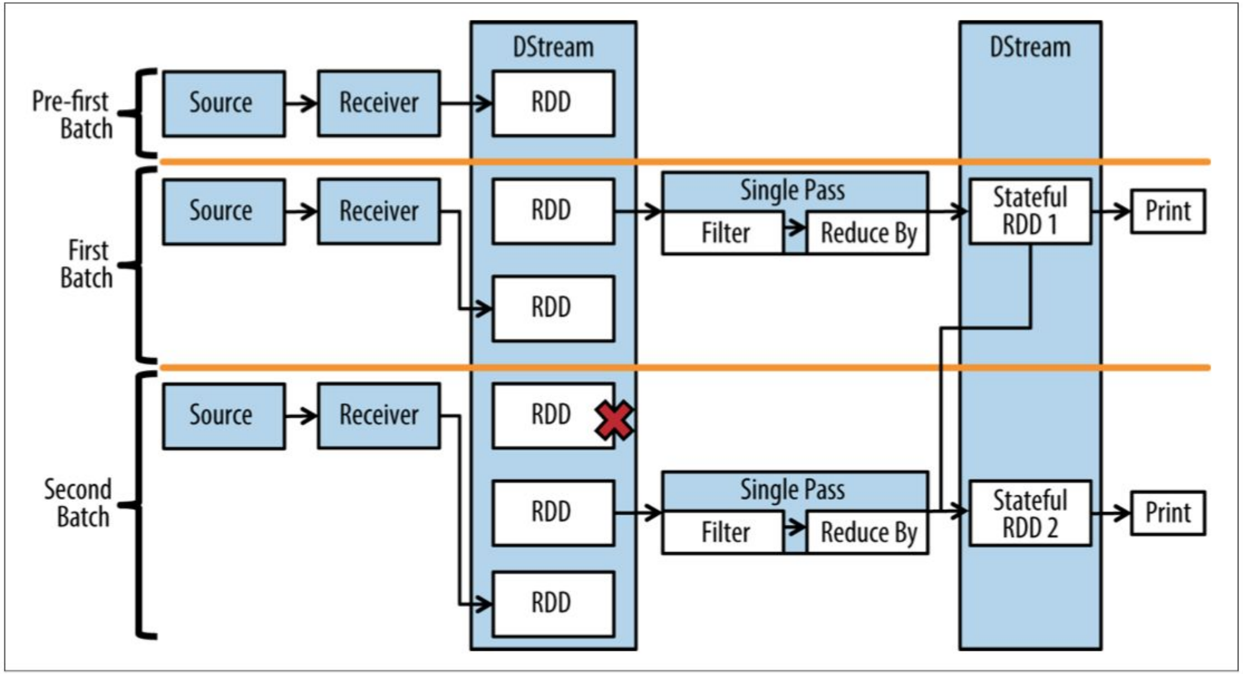
- the function for updateStateByKey() is **based on a key**, so only the values are passed in
- The first value is a Seq[Long], which contains **all new possible values for this key** from this microbatch. It is very possible that this batch didn’t include any values and this will be an empty Seq.
- The second value is an Option[Long], which **represents the value from the last microbatch**. The Option allows it to be empty because no value may have existed.
- the return is an Option[Long]. This allows us the option to remove values populated in past microbatches



```
val sc = new SparkContext(sparkConf)
val ssc = new StreamingContext(sc, Seconds(1))
val rawStream = ssc.socketTextStream(host, port, StorageLevel.ME
val countRDD = rawStream.filter(_.contains("the"))
  .count()
  .map(r => ("foo", r))
countRDD.updateStateByKey((a: Seq[Long], b: Option[Long]) => {
  var newCount = b.getOrElse(0l)
  a.foreach( i => newCount += i)
  Some(newCount)
  }).print()
       ssc.checkpoint(checkpointDir)
  ssc.start()
```

@Marina Popova

Spark Streaming



@Marina Popova