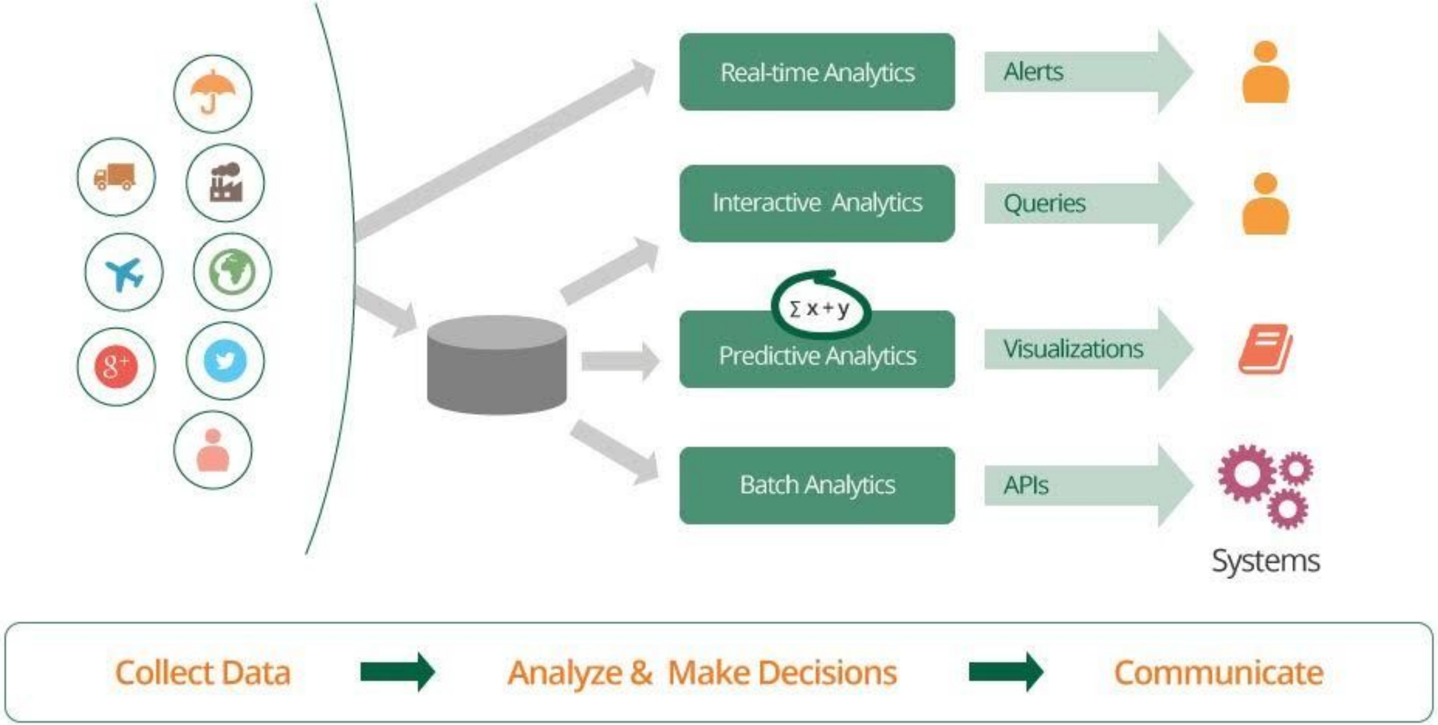
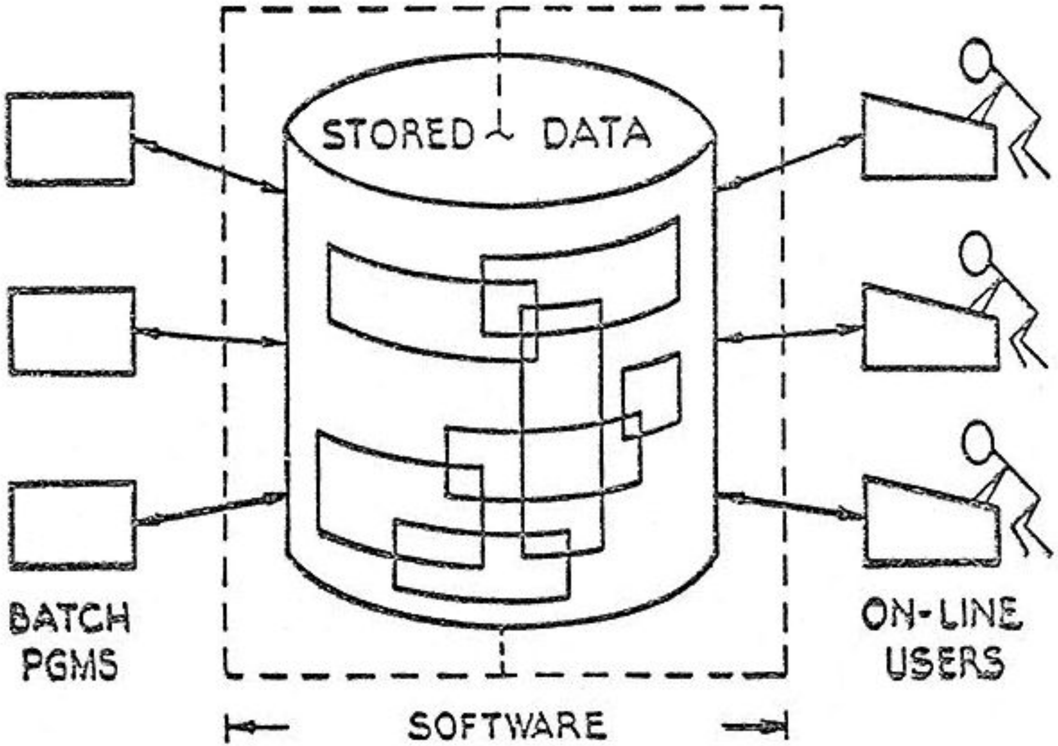
Stream Processing

Analytics



Real-time Analytics
Alerts
Interactive Analytics
Queries
Σ x + y
Predictive Analytics
Visualizations
Batch Analytics
APIs
Systems
Collect Data
Analyze & Make Decisions
Communicate

A DATABASE SYSTEM



The Stream Model

- The system cannot store the entire stream
  - Google queries, mouse clicks, sensor measurements
- Input tuples enter at a rapid rate, at one or more input ports
- Low latency (ms) requirements (Real-time processing)

The 8 Requirements of Real-Time Stream Processing (Stonebraker 2005)

1. Keep the data moving
2. Query using SQL on streams
3. Handle stream imperfections (out-of-order data)
4. Generate predictable outcomes
5. Integrate stored and streaming data
6. Guarantee data safety and availability
7. Partition and scale applications automatically
8. Process and respond instantaneously

1. Keep the data moving

- Process messages ‘in-stream’
  - No requirement to store them to perform any operation or sequence of operations.
- Push model

2. Query using SQL on streams

- Historically, general purpose languages (C++ or Java)
  - long development cycles
  - high maintenance costs.

In contrast, it is very much desirable to process moving real-time data using a **high-level language** such as SQL.

3. Handle stream imperfections (out-of-order data)

The third requirement is to have built-in mechanisms to provide resiliency against stream ‘imperfections’ including **missing** and **out-of-order** data which are commonly present in real-world data streams.

4. Generate predictable outcomes

Time series data must be processed in a predictable manner to ensure the results of processing are **deterministic** and **repeatable**.

5. Integrate stored and streaming data

The fifth requirement is to have the capability to efficiently store, modify, and access state information, and combine it with live streaming data. For **seamless integration**, the system should use a **uniform language** when dealing with either type of data.

Lambda Architecture!

6. Guarantee data safety and availability

The sixth requirement is to ensure that the applications are up and **available**, and the **integrity** of the data maintained at all times, despite failures.

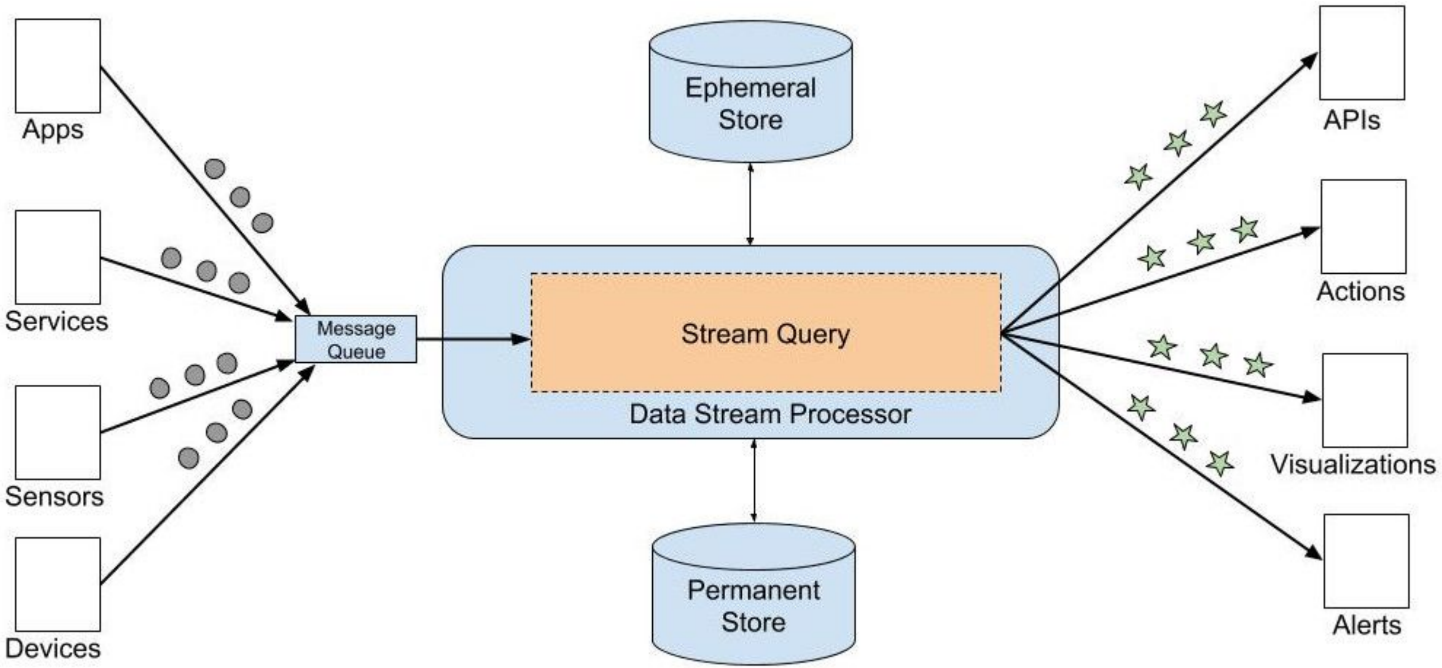
7. Partition and scale applications automatically

Distributed operation is becoming increasingly important given the favourable **price-performance** characteristics of **low-cost commodity** clusters. As such, it should be possible to split an application over multiple machines for **scalability** (as the volume of input streams or the complexity of processing increases) without the developer having to write low-level code.

8. Process and respond instantaneously

The eigth requirement is that a stream processing engine must have a highly-optimized, minimal overhead execution engine to deliver **real-time response** for **high-volume applications**.

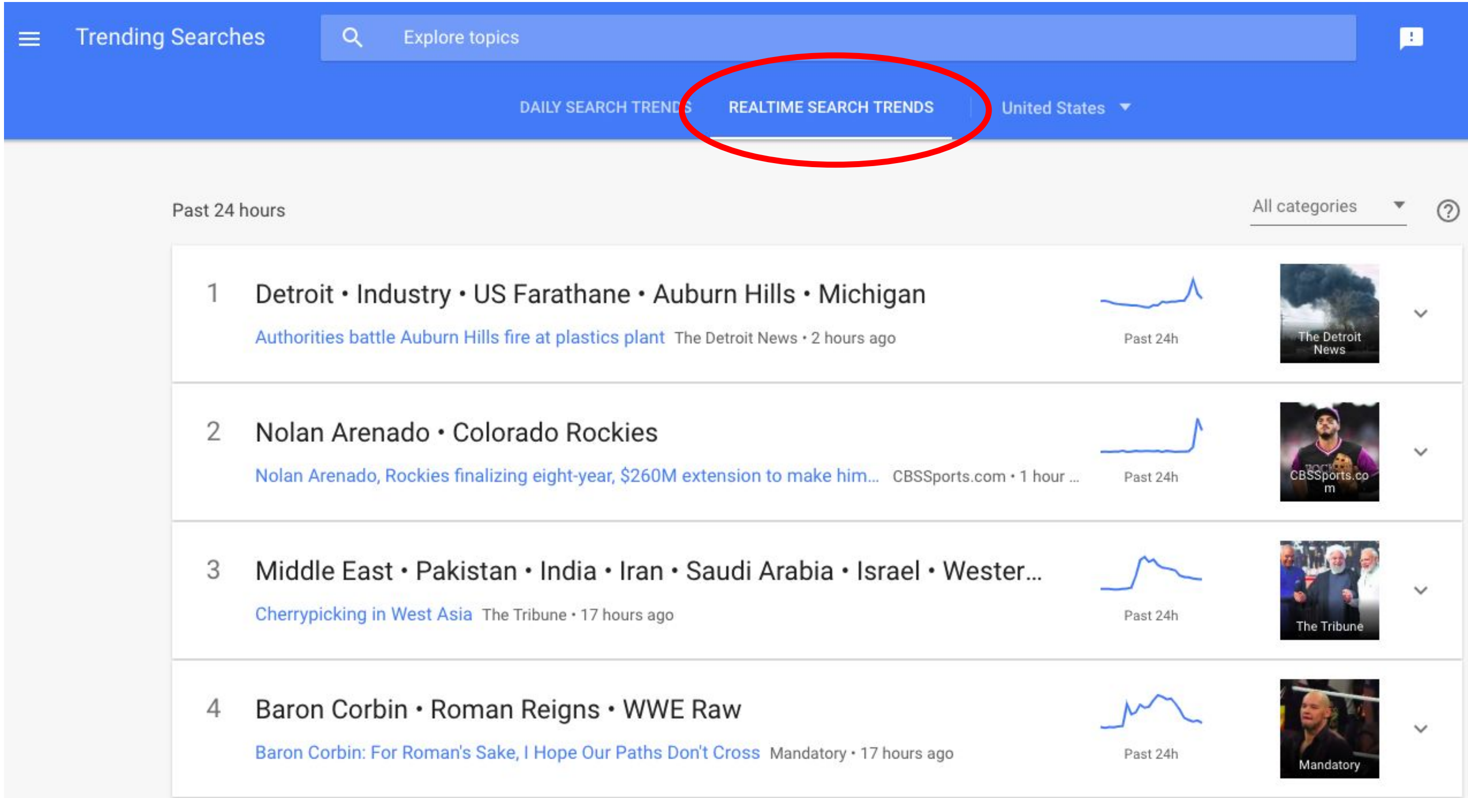
How do we make critical calculations about the stream using a **limited** amount of (secondary) **memory**?



Applications (1)

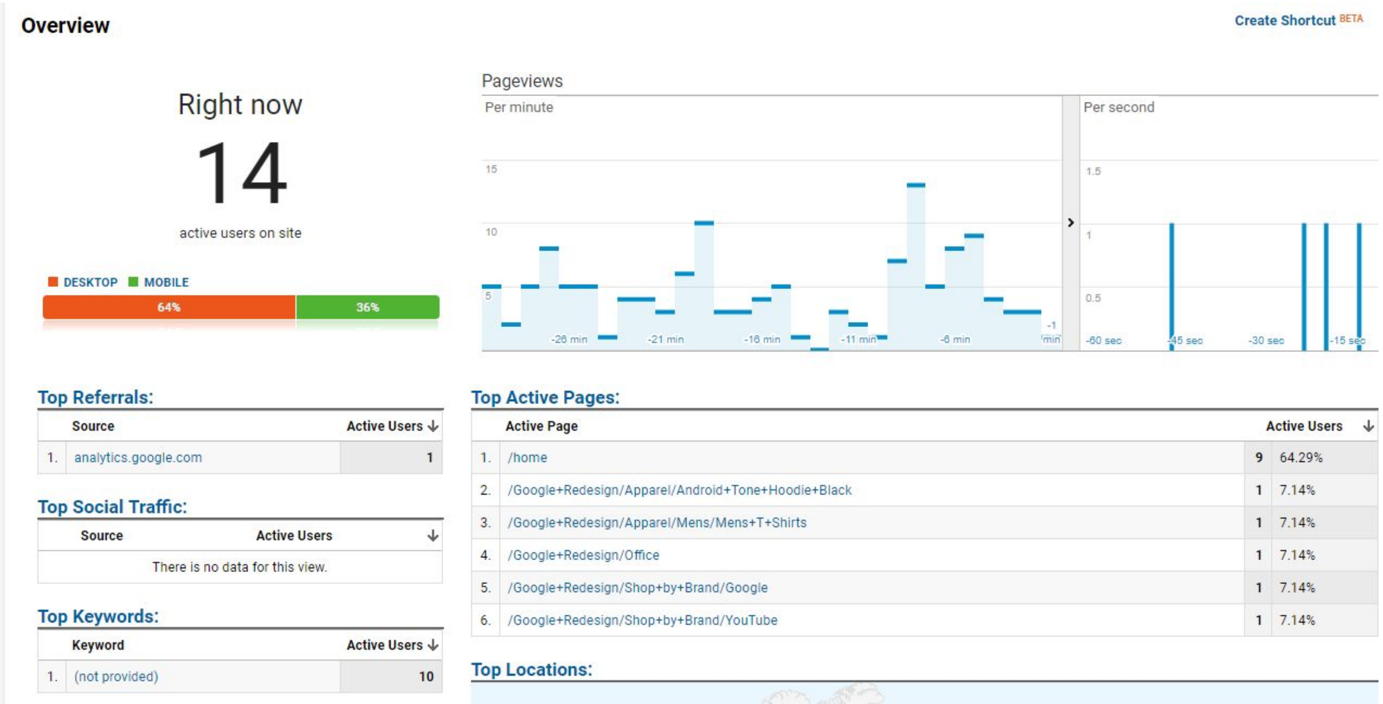
Mining query streams

Google wants to know what queries are more frequent today than yesterday



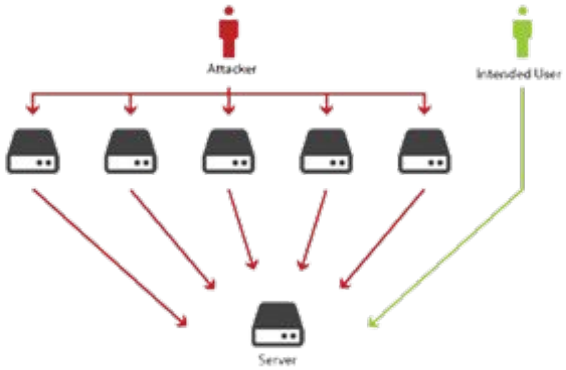
Applications (2)

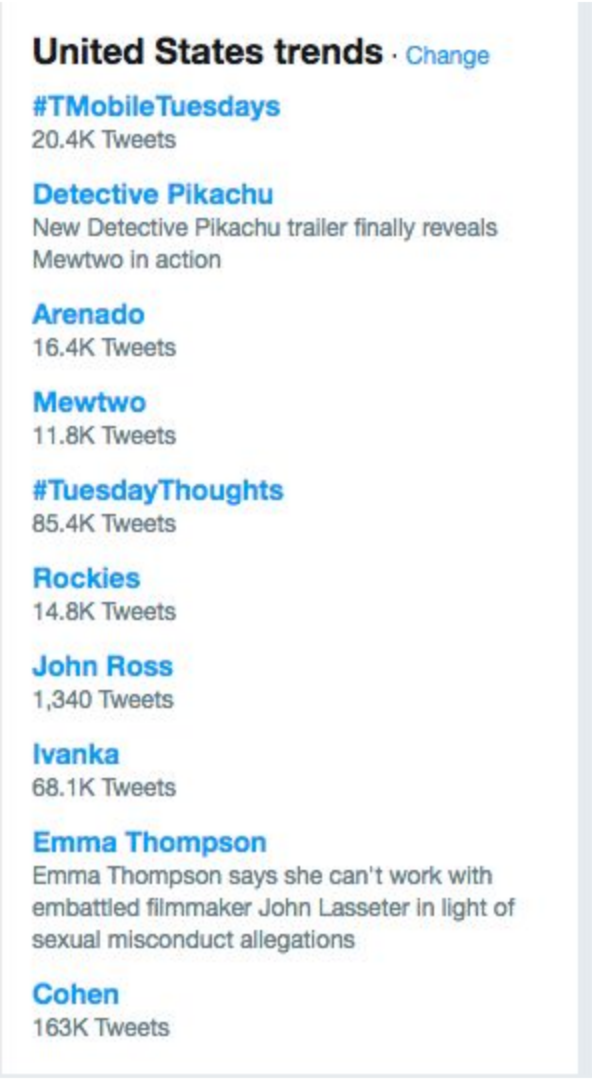
- Mining click streams
  - Google Analytics wants to know if a page is getting an unusual number of hits in the past hour



Applications (3)

- Mining social network news feeds
- Sensor networks
  - Many sensors feeding into a central controller
- Telephone call records
- IP packets monitored at a switch
  - Gather information for optimal routing
  - Detect-denial-of-service attacks





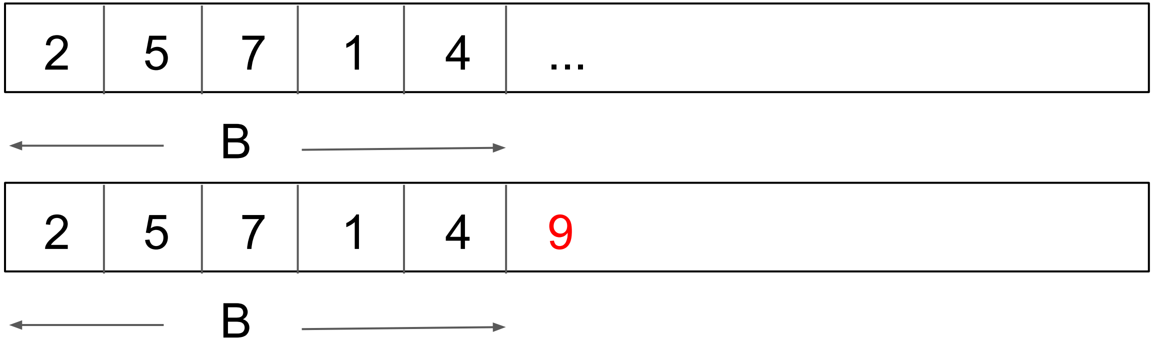
Data stream problems

- Sampling data from a stream
- Queries over sliding windows
- Filtering a data stream
- Counting distinct elements
- Finding frequent elements

We need **algorithms + systems** to tackle such problems!

Reservoir Sampling

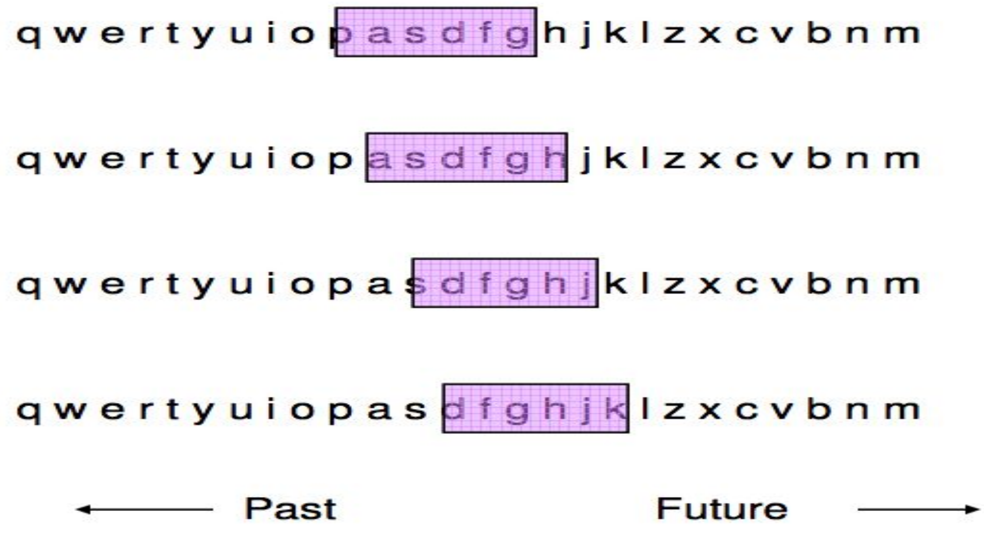
1. Ensure each item is in sample with equal probability B/n
2. Store all the first B elements of the stream
3. Suppose we have seen n-1 elements and now the n-th element arrives (n > B)
4. With probability B/n pick the n-th element, else discard it
5. If we pick the n-th element, then it replaces one of the B elements in the sample, picked at random



Sliding Windows

Queries are about a window of length N – the N most recent elements received.

Interesting case: N is so large it cannot be stored in memory.



Counting elements over sliding windows

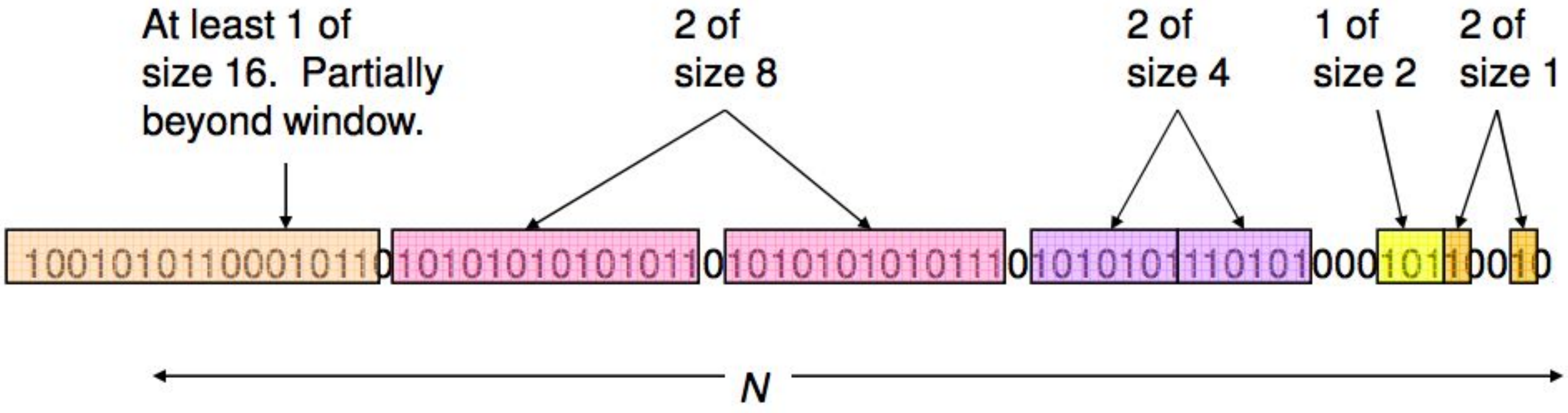
Problem: Given a stream of 0’s and 1’s, answer queries of the form “how many 1’s in the last k bits?” where $k \leq N$.

- Obvious solution: store the most recent N bits.
  - When new bit comes in, discard the N +1st bit.
- You can’t get an exact answer without storing the entire window.
- Real Problem: what if we cannot afford to store N bits?
  - We have an IoT device of limited memory and W = 1 billion
- But we’re happy with an approximate answer.

Exponential Histograms (1)

Key Idea:

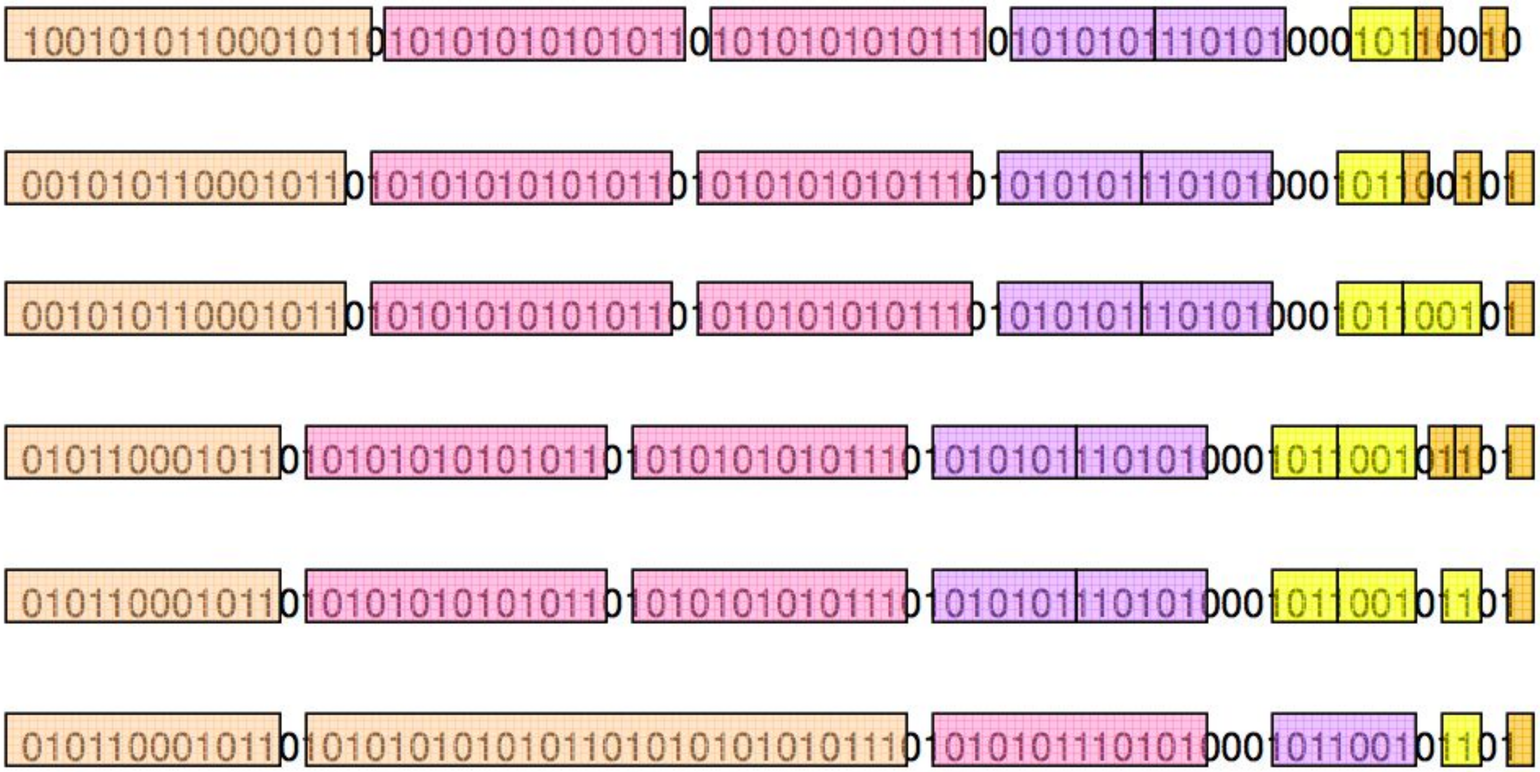
- Summarize blocks of stream with specific numbers of 1's.
- Block sizes (number of 1's) increase exponentially as we go back in time



Exponential Histograms (2)

- Each bit in the stream has a **timestamp**, starting 1, 2, …
- A **bucket** is a record consisting of:
  - The timestamp of its end.
  - The number of 1's between its beginning and end.
  - Constraint: number of 1's must be a power of 2.
- Either one or two buckets with the same power-of-2 number of 1's.
- Buckets do not overlap in timestamps.
- Buckets are sorted by size.
  - Earlier buckets are not smaller than later buckets.
- Buckets disappear when their end-time is > N time units in the past.

Example



Filtering data streams

- Each element of data stream is a tuple
- Given a list of keys S, determine which elements of stream have keys in S
- Obvious solution: store all keys S
  - S may not fit in memory (e.g., millions of filters on the same stream)

Applications:

- Email spam filtering
  - We know 1 billion “good” email addresses
  - If an email comes from one of these, it is NOT spam
- Publish-subscribe
  - People express interest in certain sets of keywords
  - Determine whether each message matches a user’s interest

Bloom Filters

- Create a bit array B of size n
- Use k independent hash functions h1 ,…,hk
- Initialize B to all 0’s
- Hash each element s in S using each function, and set B[hi (s)] = 1 for i = 1,..,k
- When a stream element with key x arrives
  - If B[hi (x)] = 1 for i= 1,..,k, then declare that x is in S
  - Otherwise discard the element
- No false-negatives
  - Is the element in the set? → No/Maybe

Example

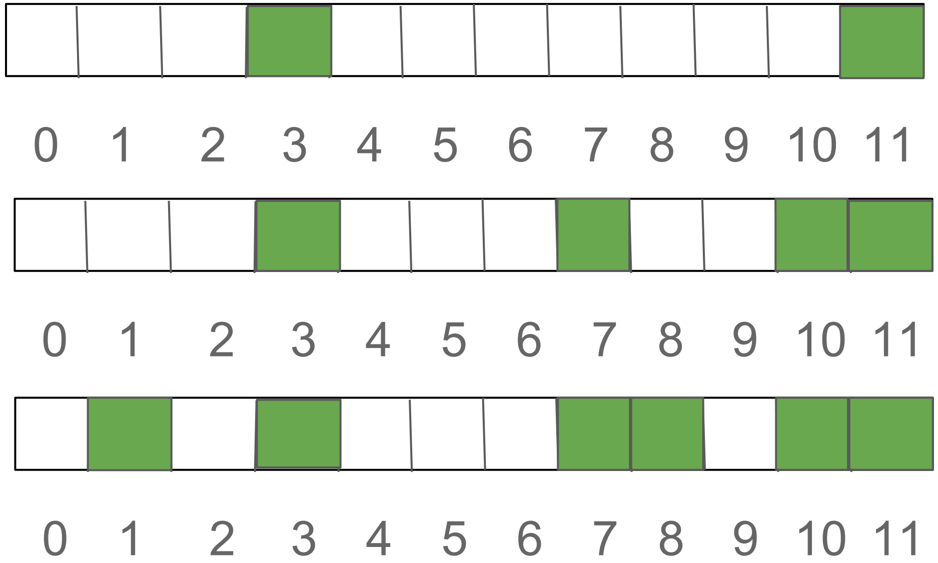
2 hash functions h1, h2

Stream: {5, 4, 15, …}

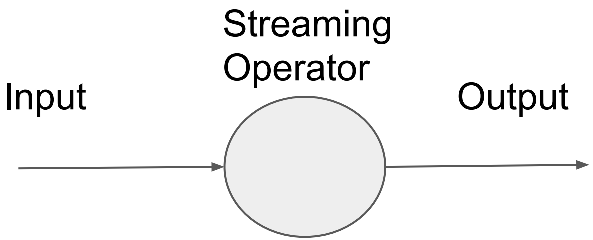
- Input: 5, h1(5) = 3, h2(5) = 11
- Input: 4, h1(4) = 7, h2(4) = 10
- Input: 4, h1(15) = 1, h2(15) = 8

Query: Is 2 in stream?

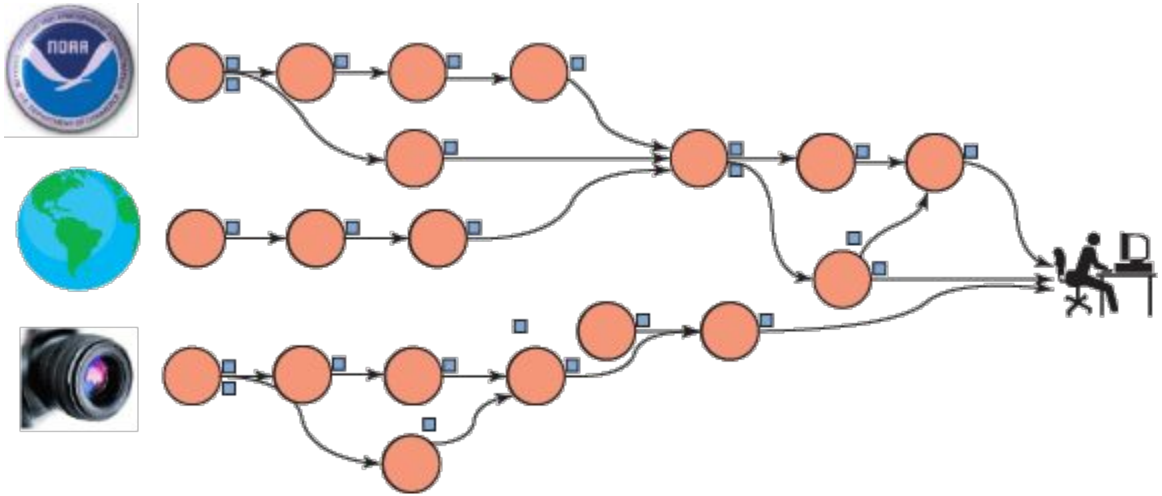
h1(2) = 8 , h2(2) = 9 → NO!



Till now...



Reality



Need for **systems** that handle complex distributed streams

- Scalability
- Fault-tolerance

Deal with scalability

If resources are not enough…

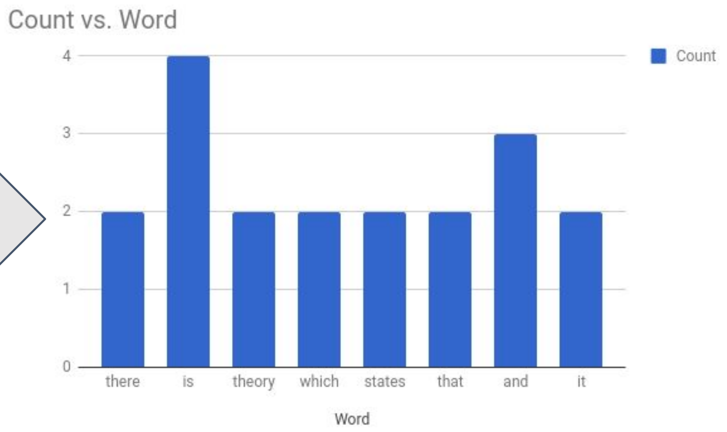
- Scale-Up: Get a **bigger** machine
  - “Huge” machines not accessible to everyone (-)
  - Not easy to migrate a live application (-)
  - Code remains as is (+)

- Scale-Out: Use **more** machines
  - Commodity-hardware (+)
  - Elasticity actions (+)
  - Distributed Algorithm is required (-)

The Word Count Example

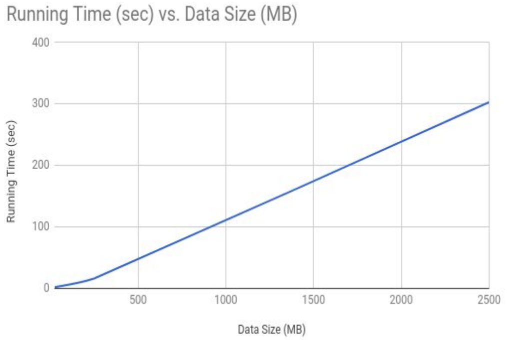
There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable. There is another theory which states that this has already happened.

WordCount



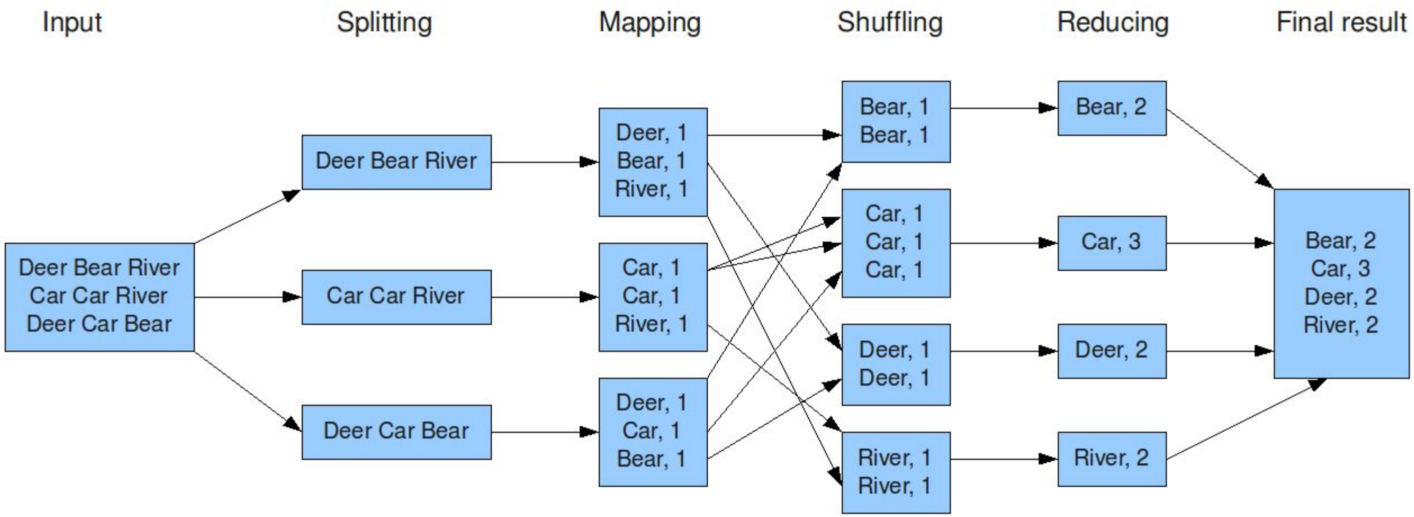
A naive implementation

```python
from collections import defaultdict
import sys

def wordcount(inputfile):
    histo = defaultdict(int)
    with open(inputfile, 'r') as f:
        doclines = f.readlines()
        for line in doclines:
            words = line.split()
            for w in words:
                histo[w.lower()] += 1
        f.close()

    for w in histo:
        print '{} {}'.format(w, histo[w])

if __name__ == '__main__':
    wordcount(sys.argv[1])
```



The bigger the dataset, the more resources we need!

Scaling out


Input
Splitting
Mapping
Shuffling
Reducing
Final result
Deer Bear River
Car Car River
Deer Car Bear
Deer Bear River
Car Car River
Deer Car Bear
Deer, 1
Bear, 1
River, 1
Car, 1
Car, 1
River, 1
Deer, 1
Car, 1
Bear, 1
Bear, 1
Bear, 1
Car, 1
Car, 1
Car, 1
Deer, 1
Deer, 1
River, 1
River, 1
Bear, 2
Car, 3
Deer, 2
River, 2
Bear, 2
Car, 3
Deer, 2
River, 2

Delivery guarantees in streams

At most once (fire and forget): the message is sent, but the sender doesn’t care if it is received or lost

At least once: retransmission of a message will occur until an ack is received.

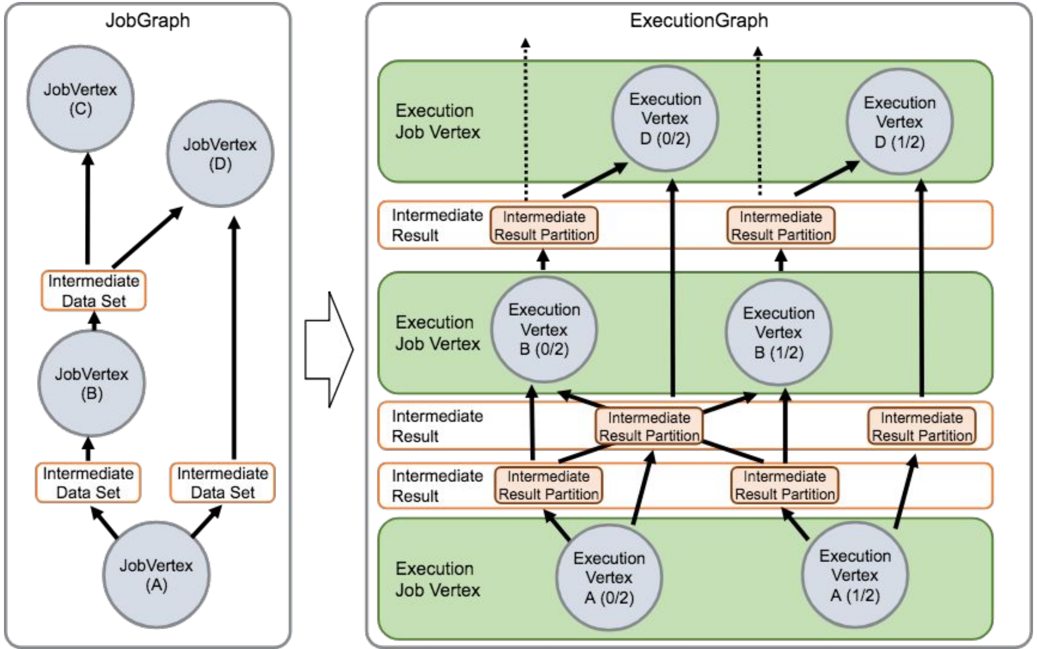
Exactly once: A message is received once and only once

Apache Flink

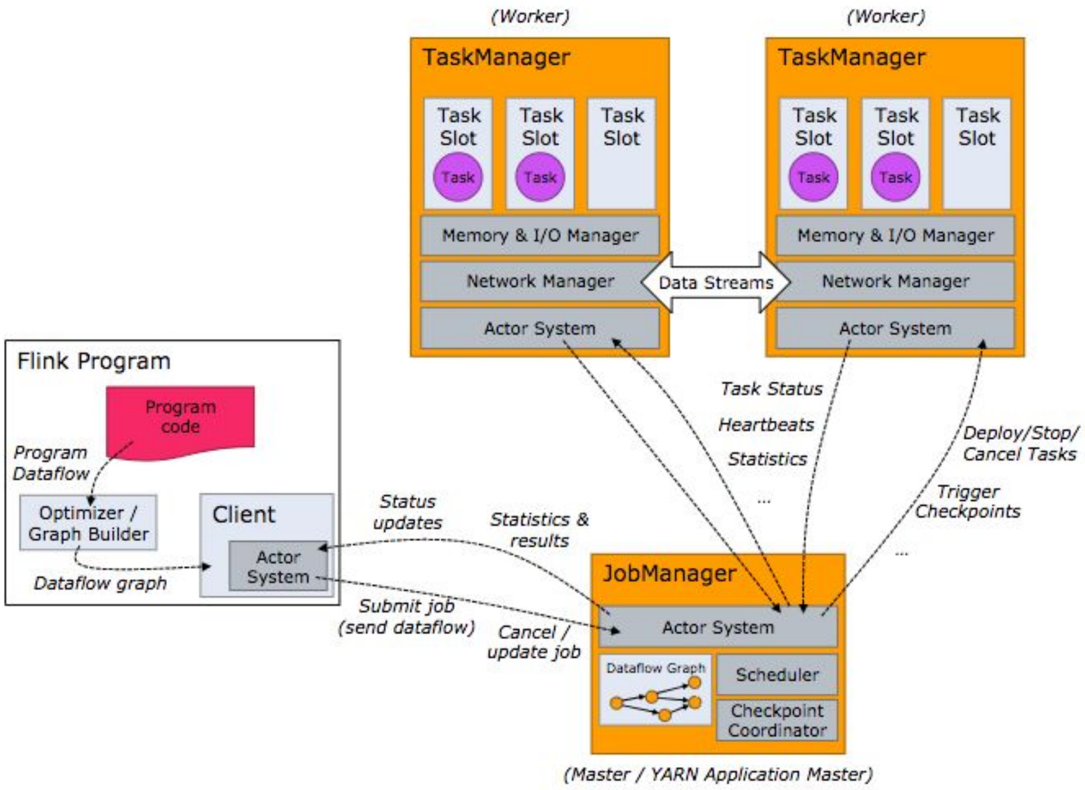
Distributed processing engine for **stateful** computations over unbounded streams.

Applications are parallelized into tasks that are distributed and concurrently executed in a cluster.

Exactly once processing



Architecture



Flink API

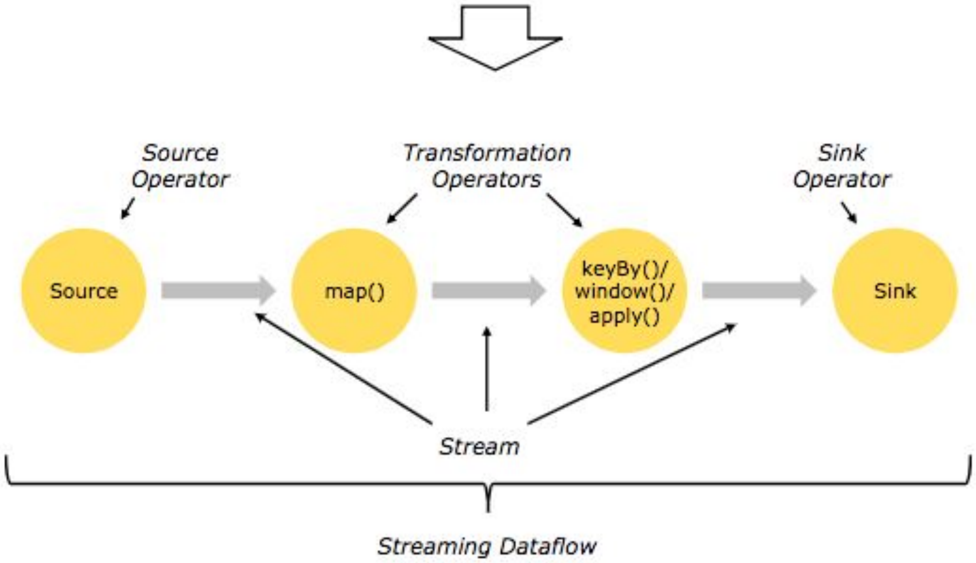
```
DataStream<String> lines = env.addSource(
                                     new FlinkKafkaConsumer<>(…));

DataStream<Event> events = lines.map((line) -> parse(line));

DataStream<Statistics> stats = events
        .keyBy("id")
        .timeWindow(Time.seconds(10))
        .apply(new MyWindowAggregationFunction());

stats.addSink(new RollingSink(path));
```

Source

Transformation

Transformation

Sink

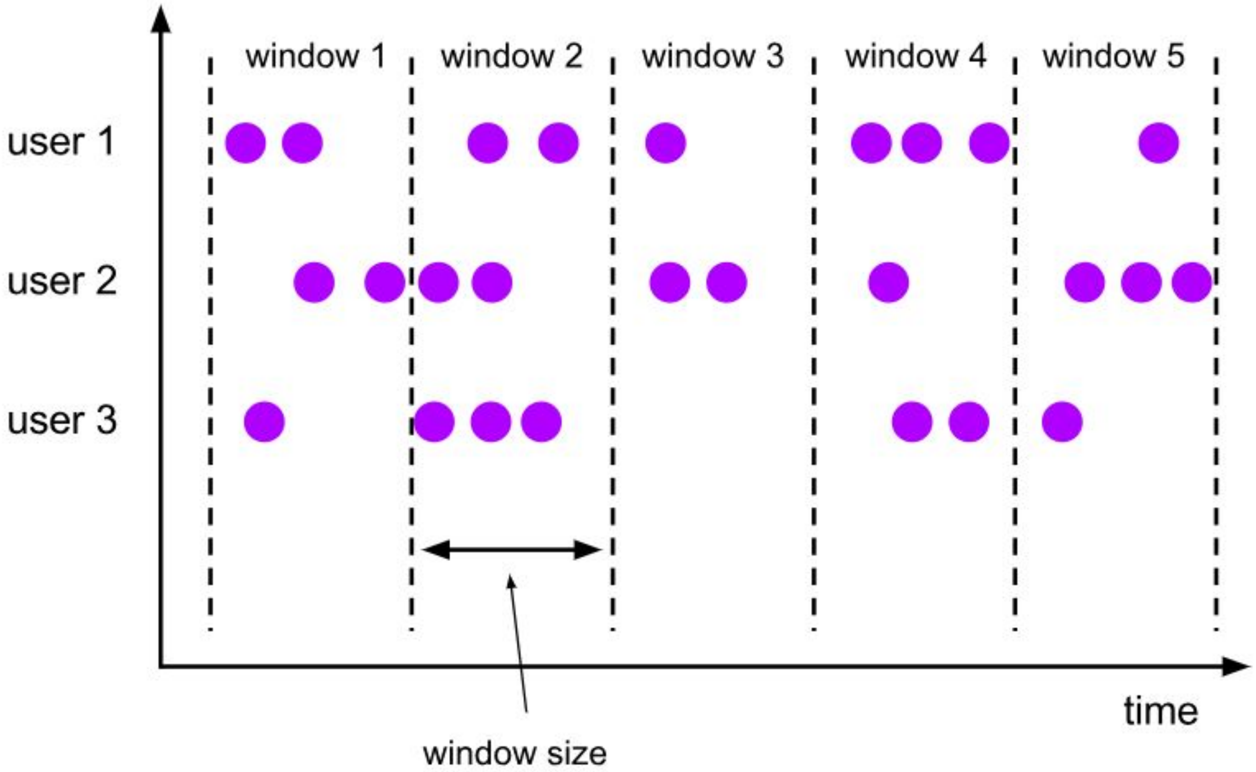


Windows

- Tumbling
- Sliding
- Session

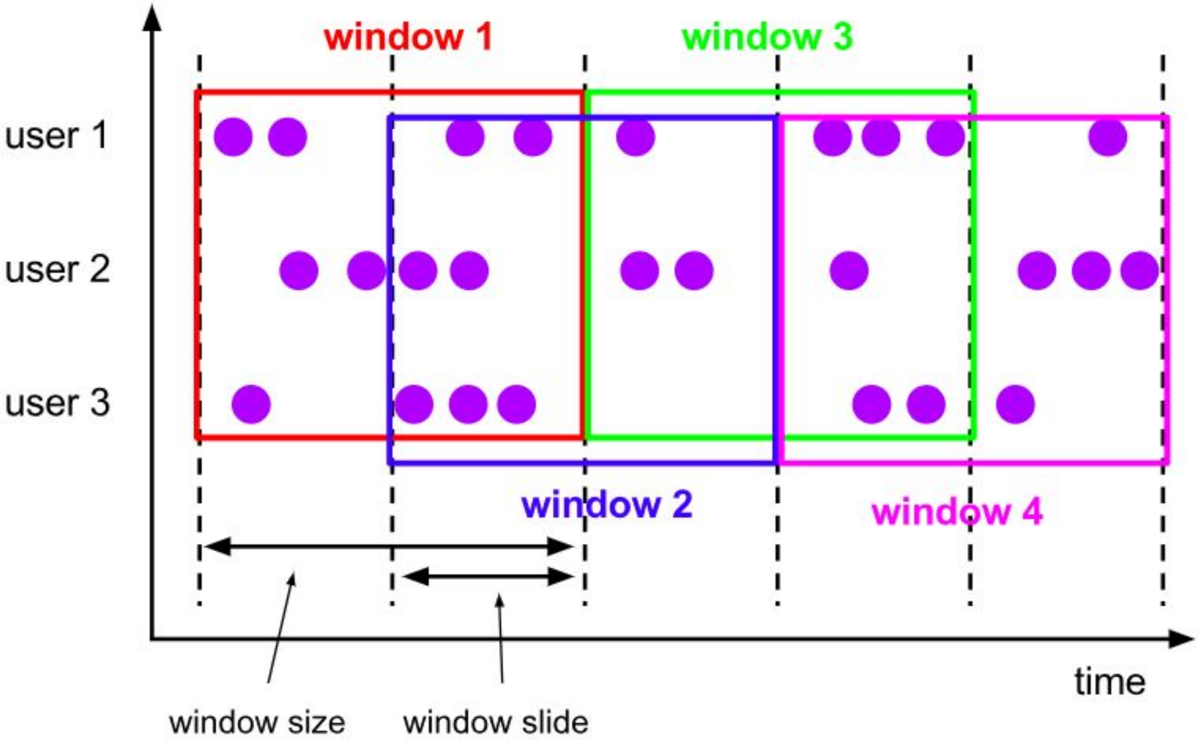
Tumbling Windows

- Configurable size
- Non-overlapping



Sliding windows

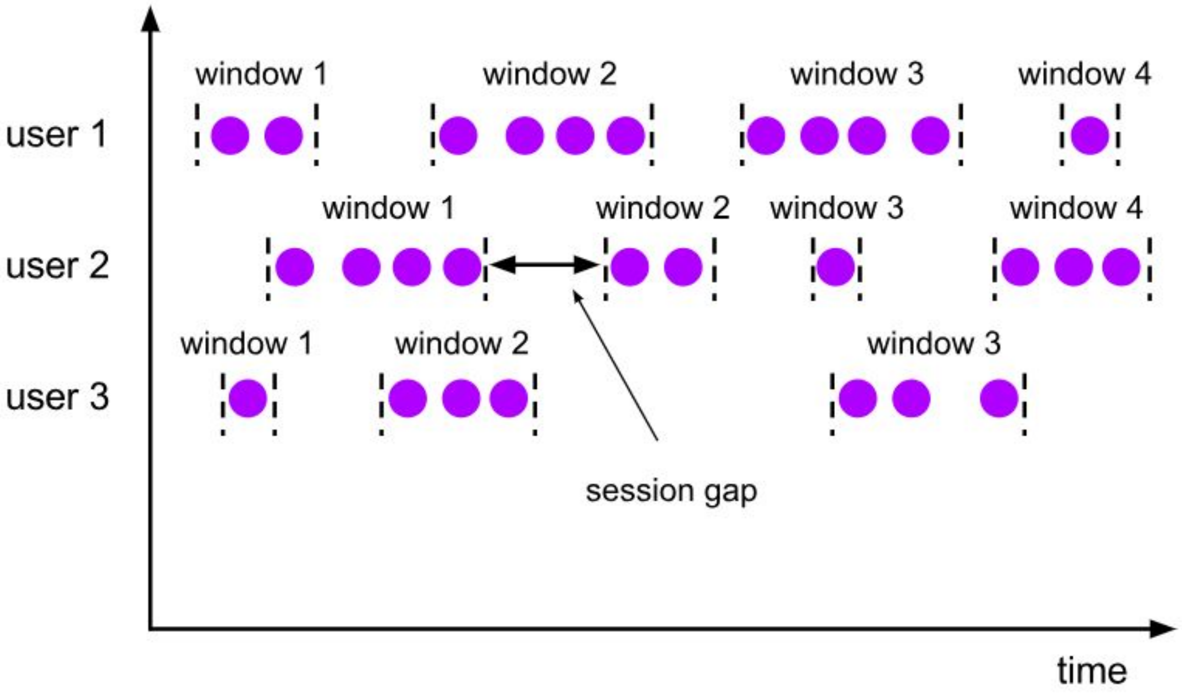
- Configurable size
- Configurable slide



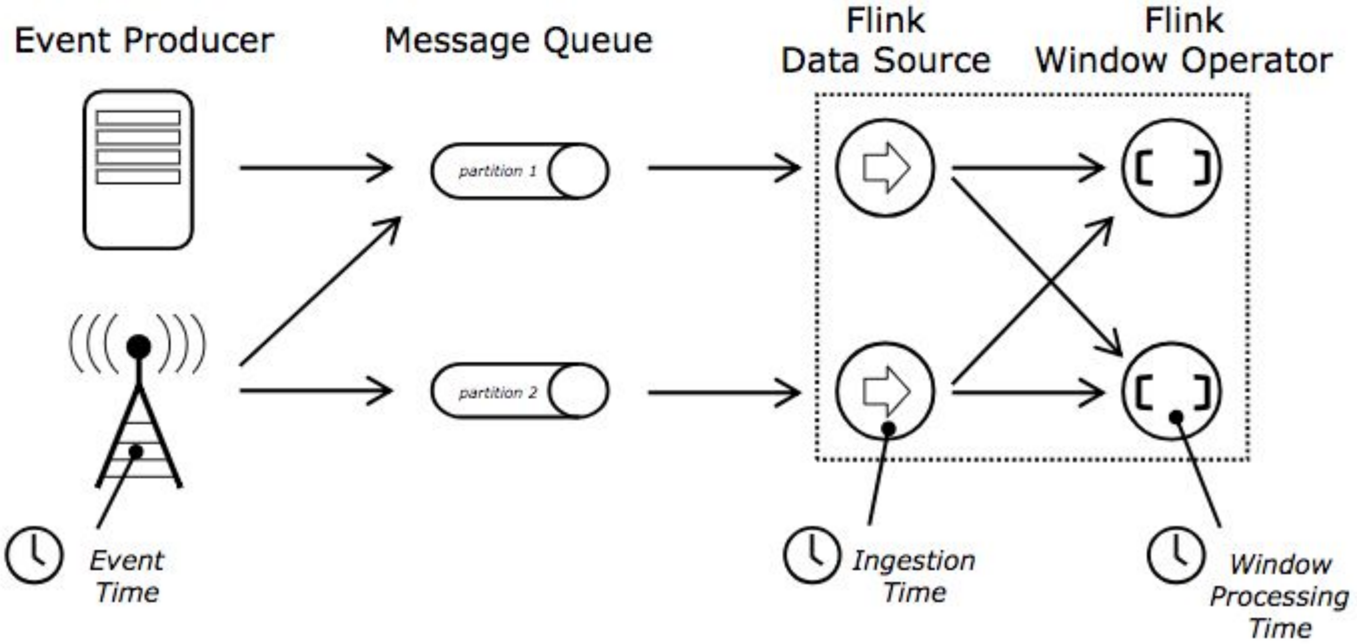
Session Windows

The window closes when a gap of inactivity occurrs

- Configurable incativity gap



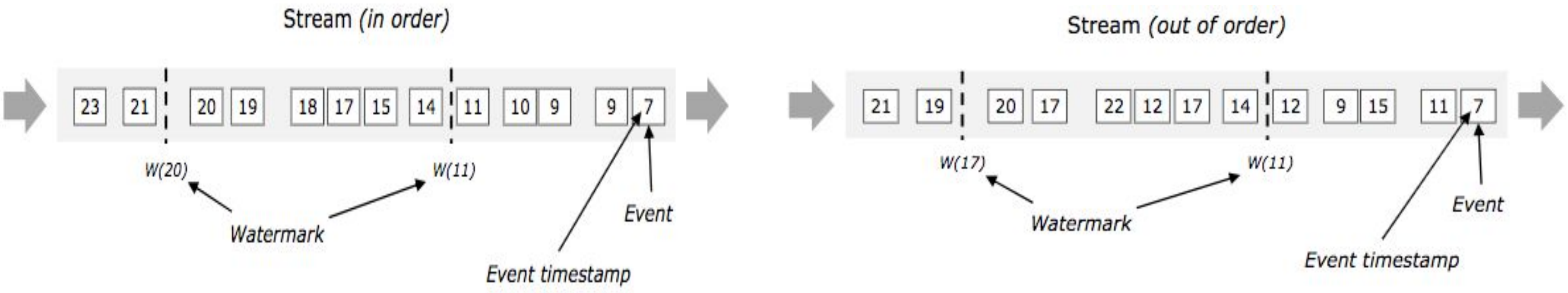
Time in Flink



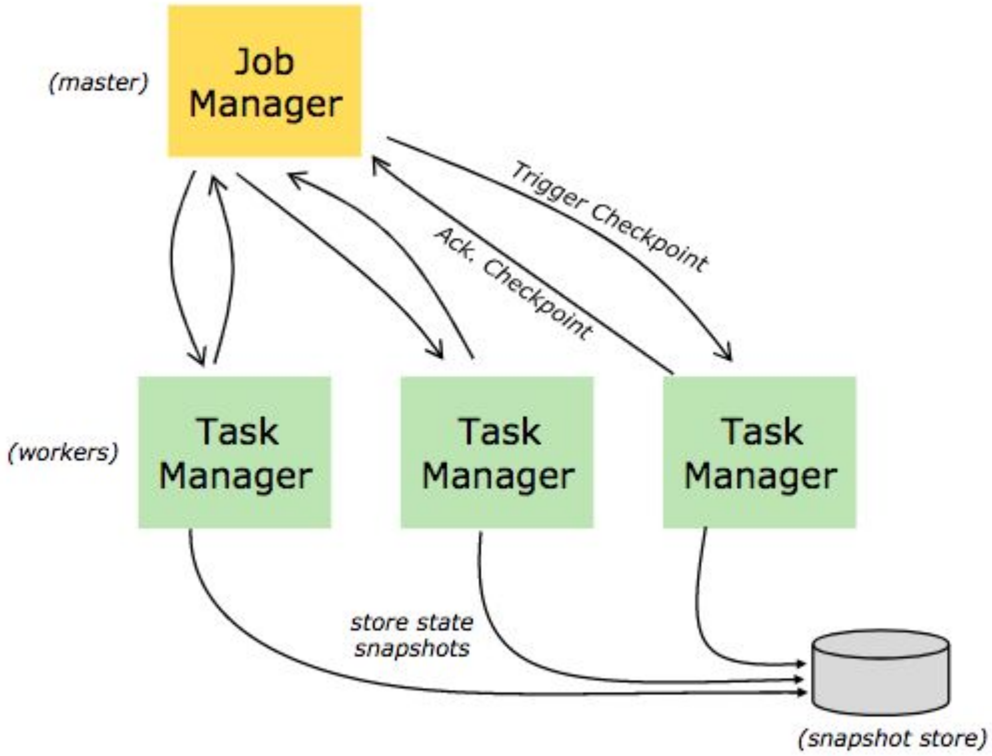

```
env.setStreamTimeCharacteristic (TimeCharacteristic .ProcessingTime);
```

Event Time and Watermarks

- Event time can progress independently of processing time
  - progress through weeks of event time with only a few seconds of processing
- Watermarks
  - part of the data stream and carry a timestamp t
  - event time has reached time t in that stream
  - no more elements from the stream with a timestamp t’ <= t

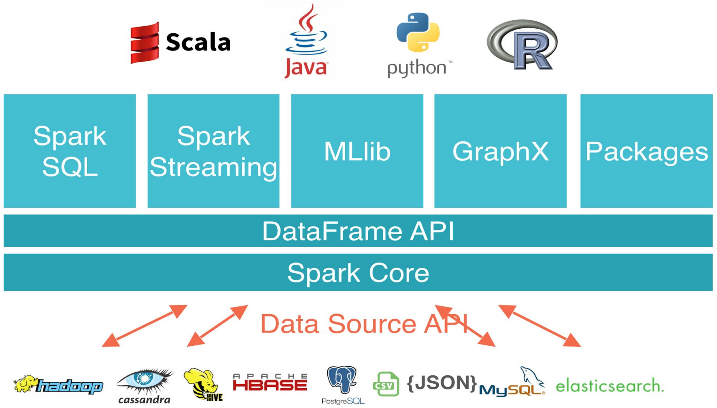


State Backends



Spark Streaming

- Data ingested from many sources
- Processed using complex algorithms expressed with high-level functions like map, reduce, join and window.
- Machine learning and graph processing algorithms on data streams

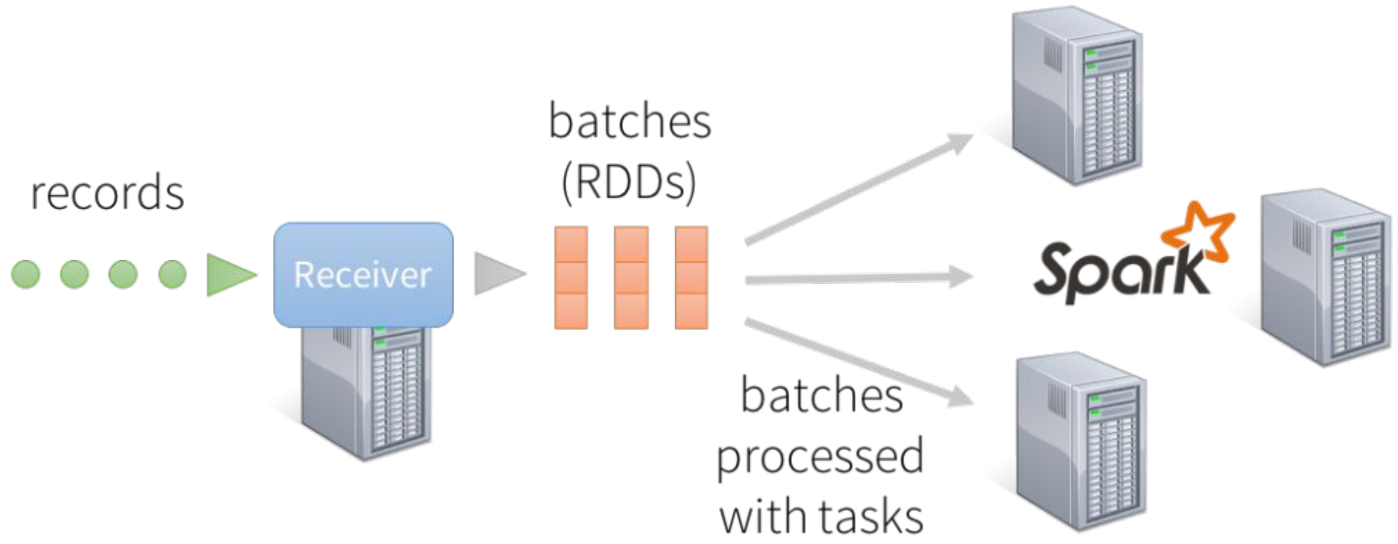


Resilient Distributed Datasets (RDD)

- Immutable distributed collection of objects
- Divided into logical partitions
- Can be persisted **in memory**
- How to create an RDD:
  - Parallelize existing collection
  - Reference dataset in an external storage system (e.g., HDFS)
- Transformations
  - **Lazy evaluation**
- Actions
  - Trigger actual computation

Spark Streaming

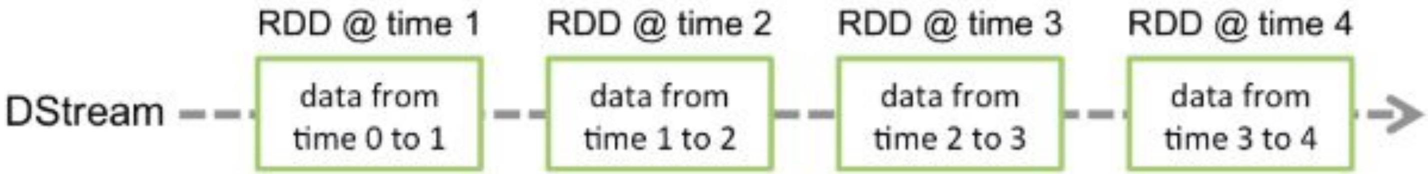
discretized stream processing



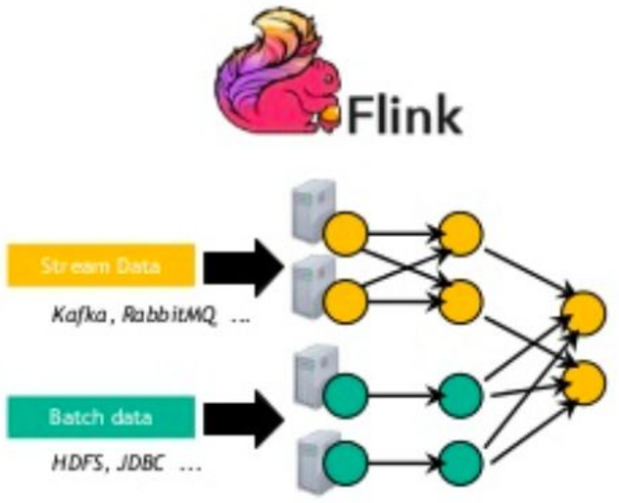
records processed in batches with short tasks
each batch is a RDD (partitioned dataset)

Discretized Streams

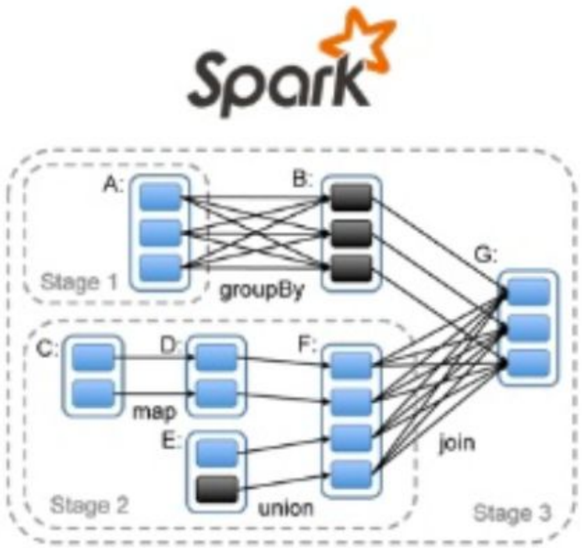
DStream = Sequence of RDDs



Computational Models



Flink computation is fully pipelined by default



Spark RDDs break down the computation into stages

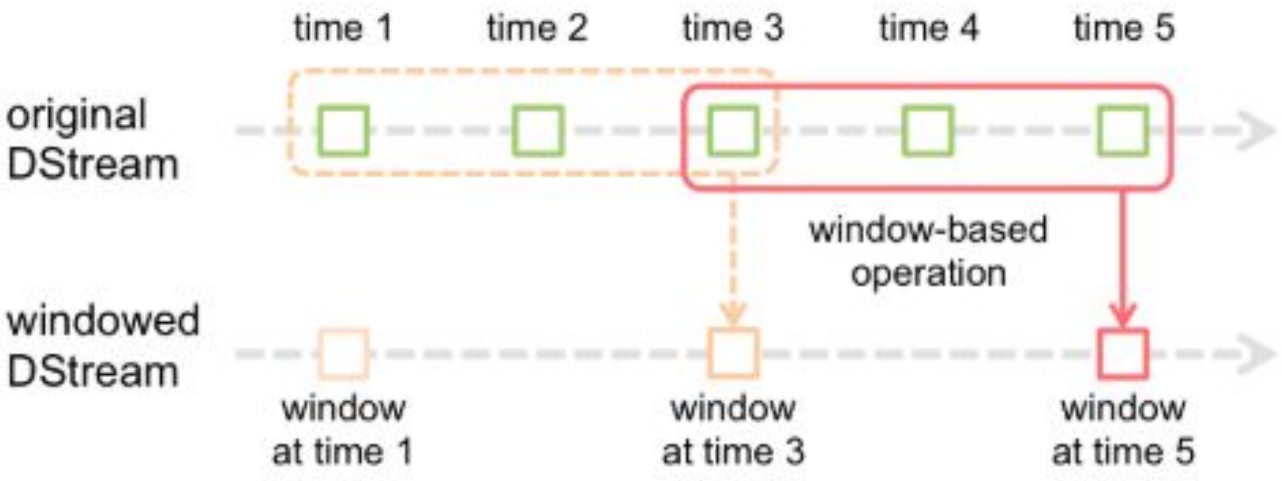
Word Count in TCP sockets

```
// Create a DStream that will connect to hostname:port, like localhost:9999
JavaReceiverInputDStream<String> lines = jssc.socketTextStream("localhost", 9999);
```

```
// Split each line into words
JavaDStream<String> words = lines.flatMap(x -> Arrays.asList(x.split(" ")).iterator());
```

```
// Count each word in each batch
JavaPairDStream<String, Integer> pairs = words.mapToPair(s -> new Tuple2<>(s, 1));
JavaPairDStream<String, Integer> wordCounts = pairs.reduceByKey((i1, i2) -> i1 + i2);

// Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.print();
```

Sliding Windows



Window length and sliding interval must be multiples of the batch interval of the source DStream

Fault-tolerance

- A streaming application must operate 24/7
  - Resilient to failures
- Two types of data that are checkpointed:
  - Metadata checkpointing - Saving of the information defining the streaming computation to fault-tolerant storage
    - Configuration - The configuration that was used to create the streaming application.
    - DStream operations - The set of DStream operations that define the streaming application.
    - Incomplete batches - Batches whose jobs are queued but have not completed yet.
  - Data checkpointing - Saving of the generated RDDs to reliable storage. This is necessary in some stateful transformations that combine data across multiple batches.

Delivering Guarantees in Spark Streaming

Processing phases of Spark Streaming:

1. Receive data (depends on data source)
2. Do transformation (**exactly once**)
3. Push outputs. (at least once - depends on data source)

Other streaming frameworks

STORM

APACHE kafka®

samza

Use cases (1)

Event-driven applications: stateful, ingest events from 1/many streams and react by triggering computations, state updates, or external actions.

- Fraud detection
- Anomaly detection
- Rule-base alerting
- Business process monitoring

Data Analytics: extract information and insight from raw data in real-time fashion

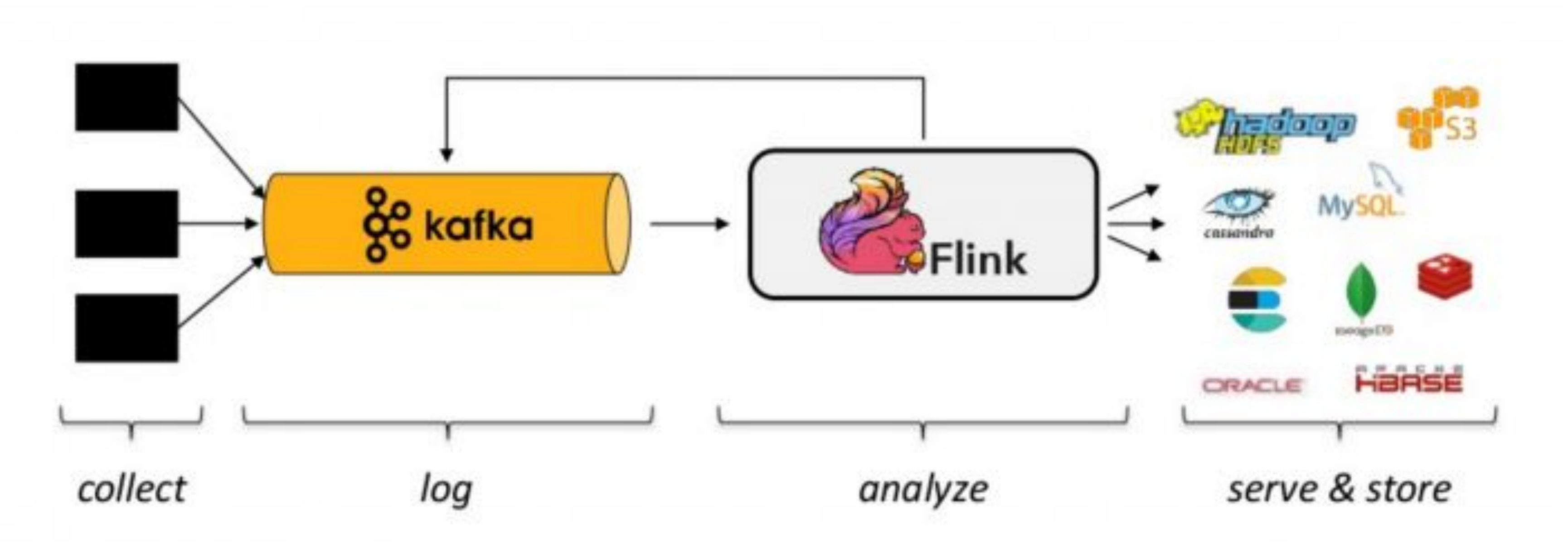
- Quality monitoring of networks
- Large-scale graph analysis

Use cases (2)

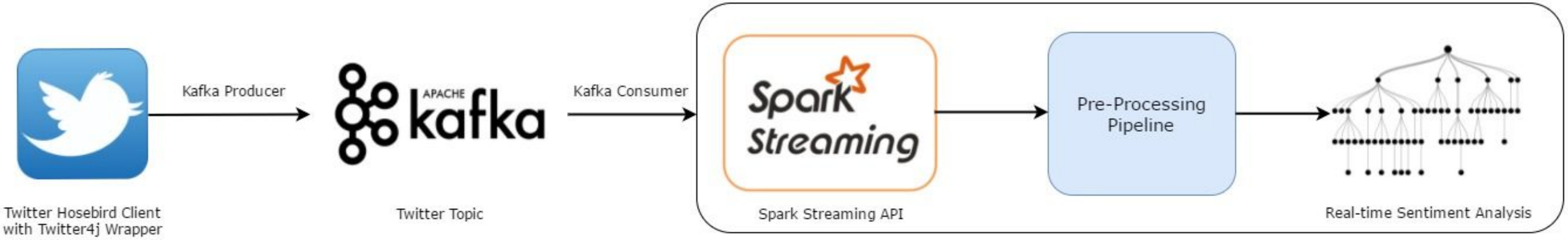
Data pipeline applications:

- Extract-transform-load (ETL) is a common approach to convert and move data between storage systems.
- Data pipelines transform and enrich data and can move it from one storage system to another.
- Continuous streaming mode instead of being periodically triggered.
- Real-time search index building in e-commerce
- Continuous ETL

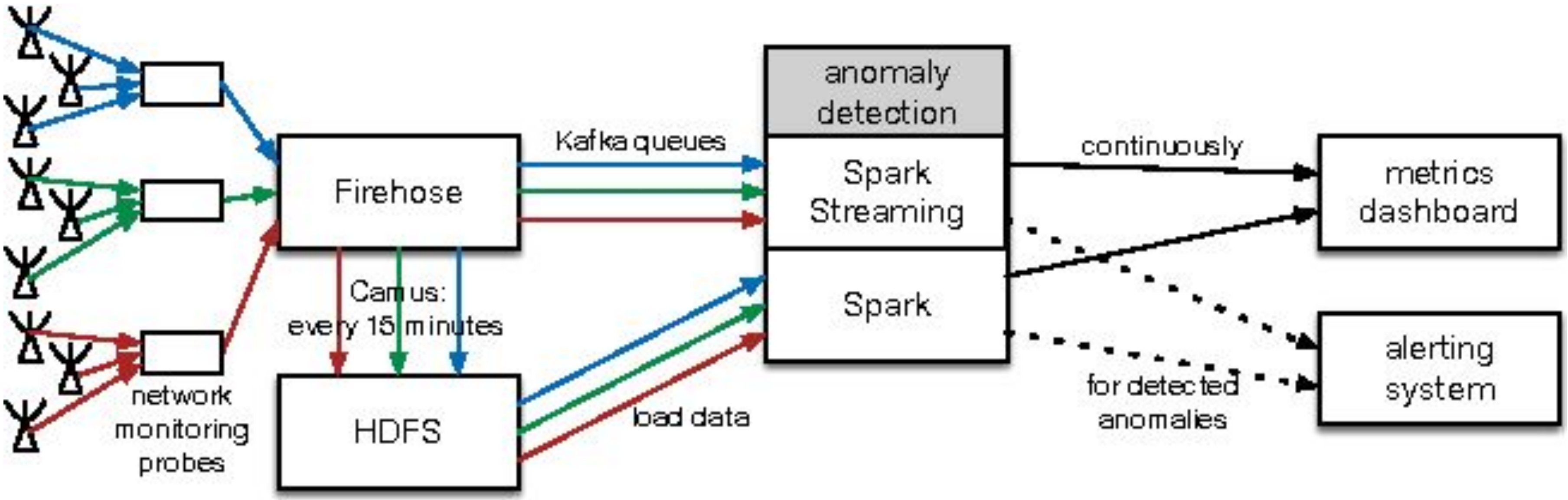
Typical Application Architecture



Twitter Sentiment Analysis



Anomaly Detection




That's all Folks!