Batch & Stream Processing with Apache Spark

Pramod Bhatotia
http://homepages.inf.ed.ac.uk/pbhatoti/

Credits for the lecture material:
Spark NSDI paper and presentation
Spark Streaming SOSP paper and presentation
Apache Flink Website

THE UNIVERSITY of EDINBURGH

Why Spark?

- MapReduce greatly simplified "big data" analysis on large, unreliable clusters

- But as soon as it got popular, users wanted more:
  - More **complex**, multi-stage applications (e.g. iterative machine learning & graph processing)
  - More **interactive** ad-hoc queries

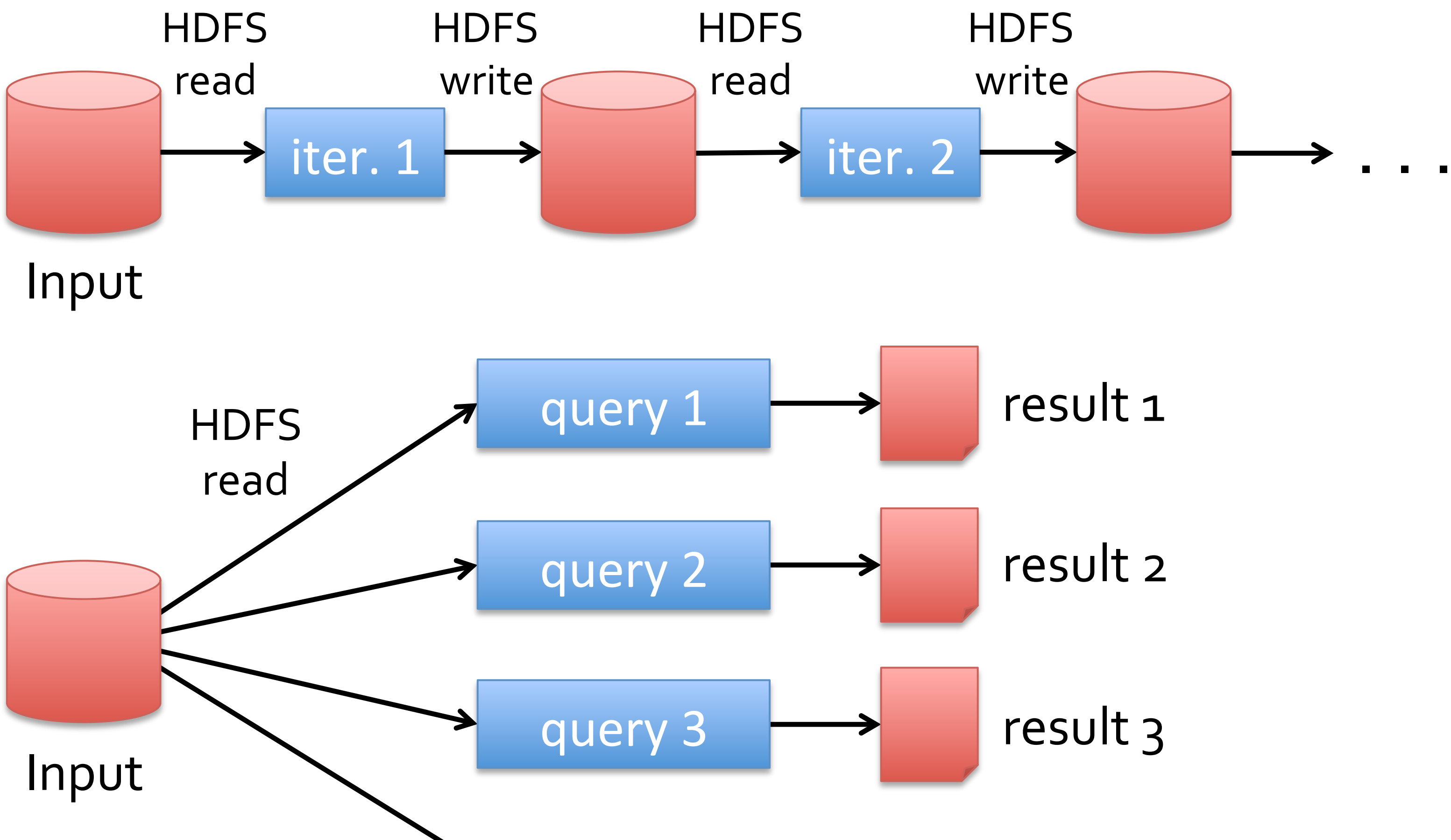
Why Spark?

- Complex apps and interactive queries both need one thing that MapReduce lacks:

  Efficient primitives for **data sharing**

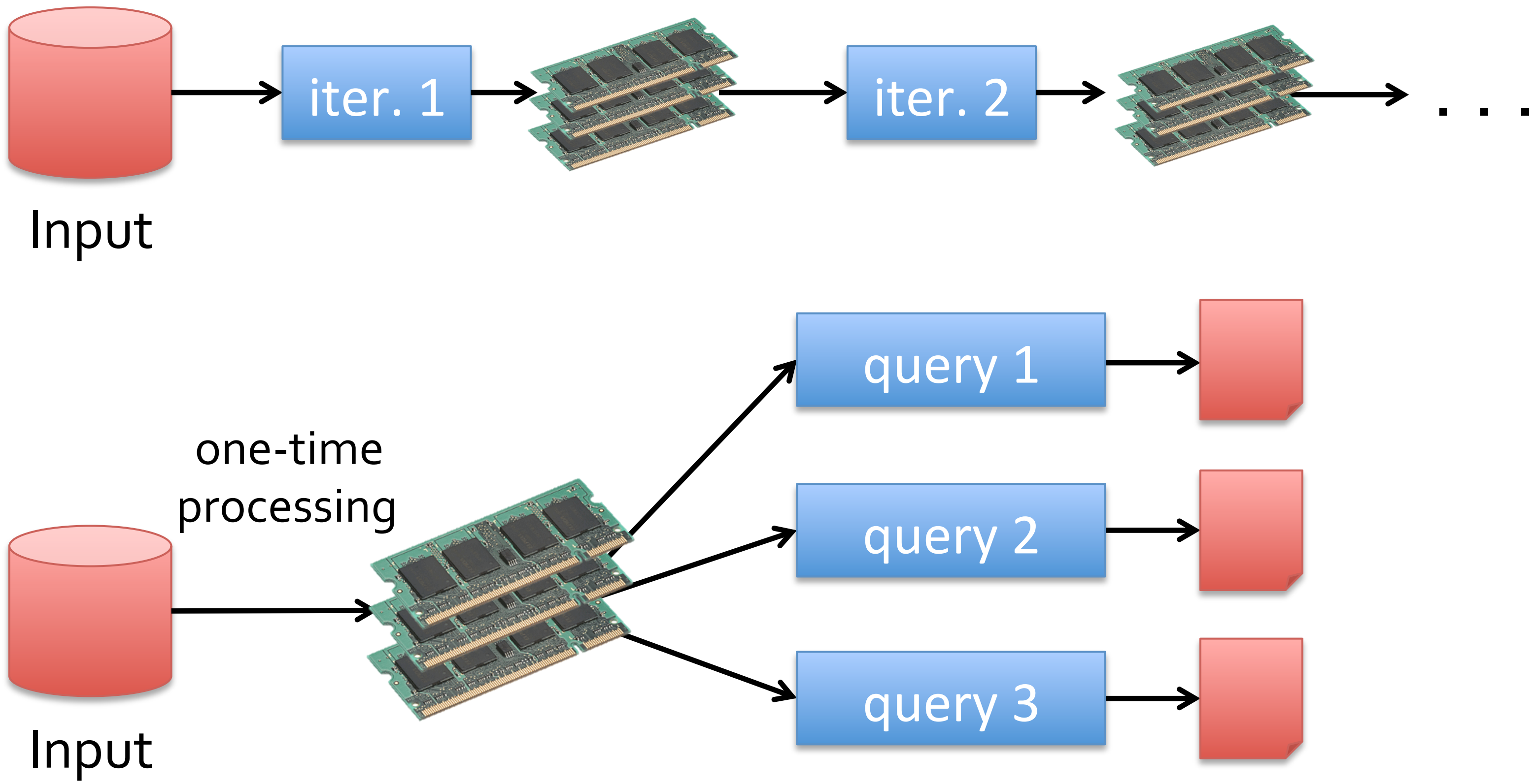
In MapReduce, the only way to share data across jobs is stable storage → slow!

Example



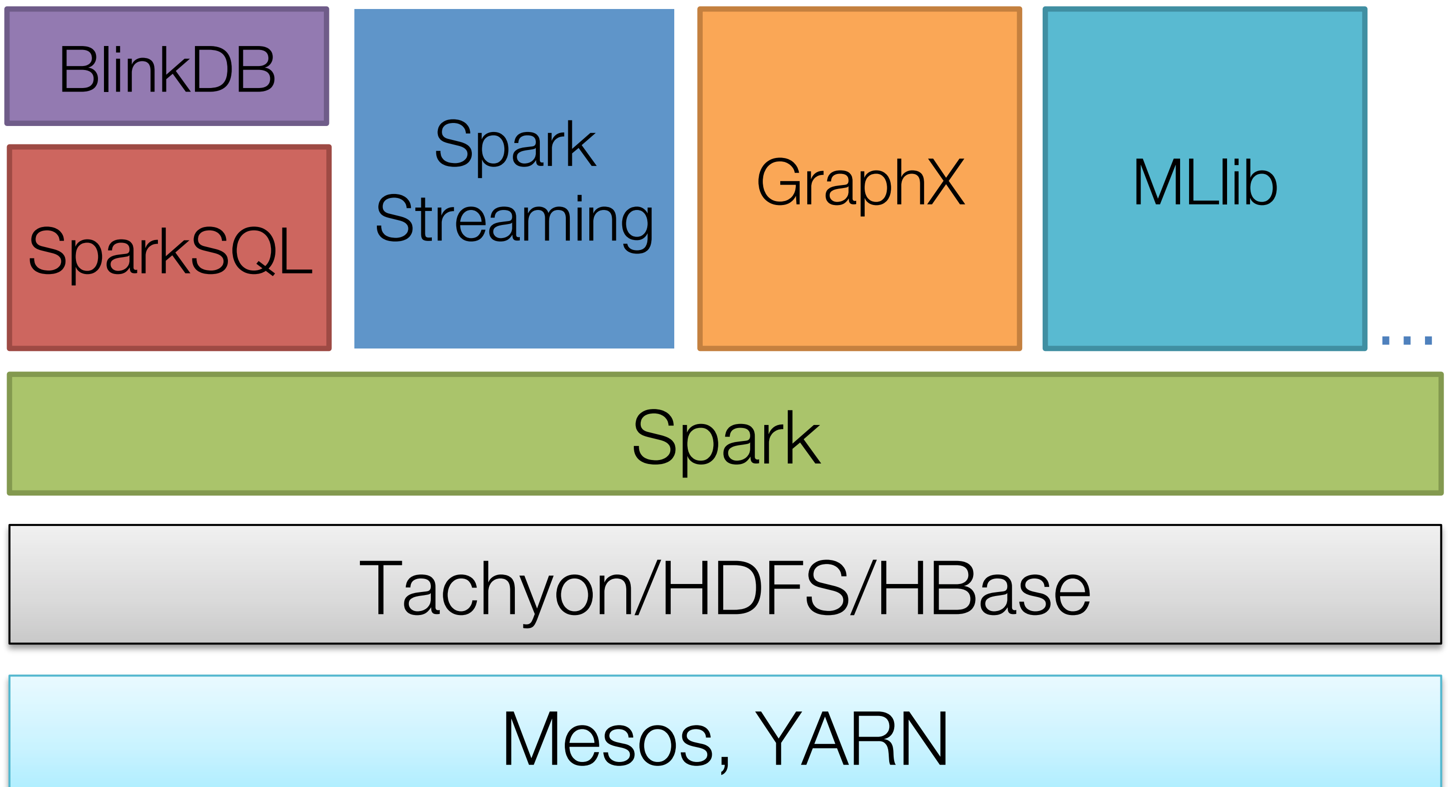
Slow due to replication and disk I/O,
but necessary for fault tolerance

Goal: In-memory data sharing



Spark project

An in-memory cluster computing framework



Pitfall of in-memory computing

10-100× faster than network/disk, but how to get FT?

How to design a distributed memory abstraction that is both **fault-tolerant** and **efficient**?

Key contribution

Resilient Distributed Datasets (RDDs)

In this lecture

1. What are RDDs?
2. How Spark uses RDDs to achieve efficiency and fault-tolerance?

RDDs

- Restricted form of distributed shared memory
  - Immutable, partitioned collections of records
  - Can only be built through *coarse-grained* deterministic transformations (map, filter, join, …)
- Efficient fault recovery using *lineage*
  - Log one operation to apply to many elements
  - Re-compute lost partitions on failure
  - No cost if nothing fails

Spark programming interface

- DryadLINQ-like API in the Scala language
- Usable interactively from Scala interpreter
- Interface for Java/Python/Scala

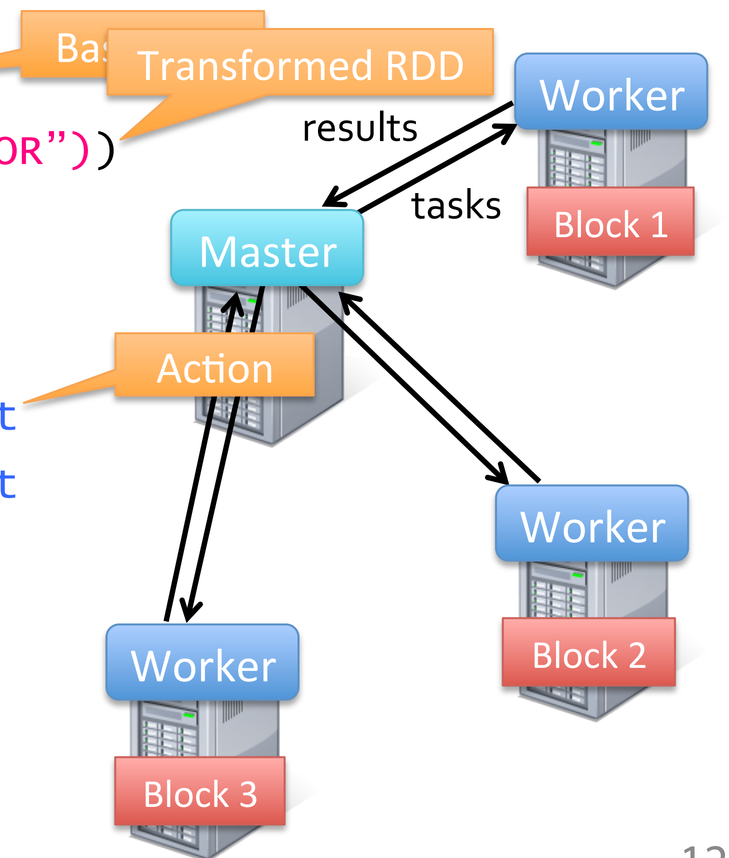
Provides:

- Resilient distributed datasets (RDDs)
- Operations on RDDs: *transformations* (build new RDDs), *actions* (compute and output results)
- Control of each RDD's *partitioning* (layout across nodes) and *persistence* (storage in RAM, on disk, etc)

Example: Log mining

Load error messages from a log into memory, then interactively search for various patterns

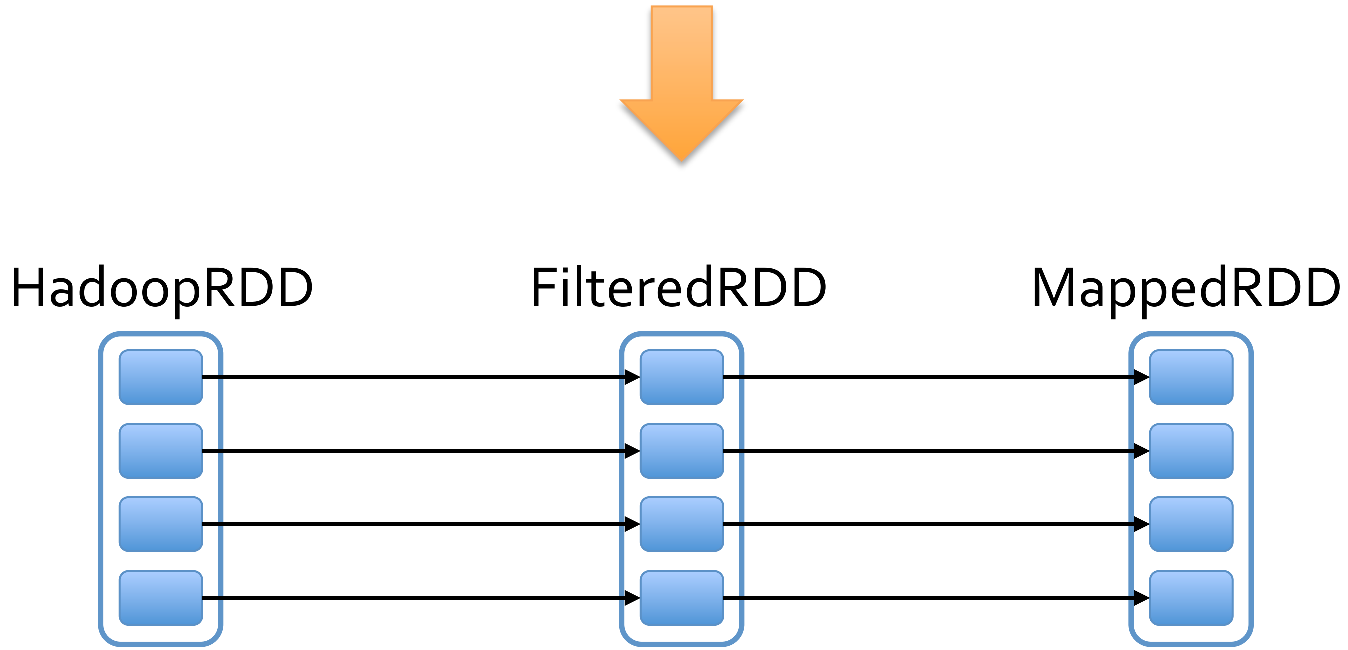
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
messages.persist()

messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count
```



Fault tolerance

RDDs track the graph of transformations that built them (their *lineage*) to rebuild lost data

E.g.:
```
messages = textFile(...).filter(_.contains("error"))
                        .map(_.split('\t')(2))
```

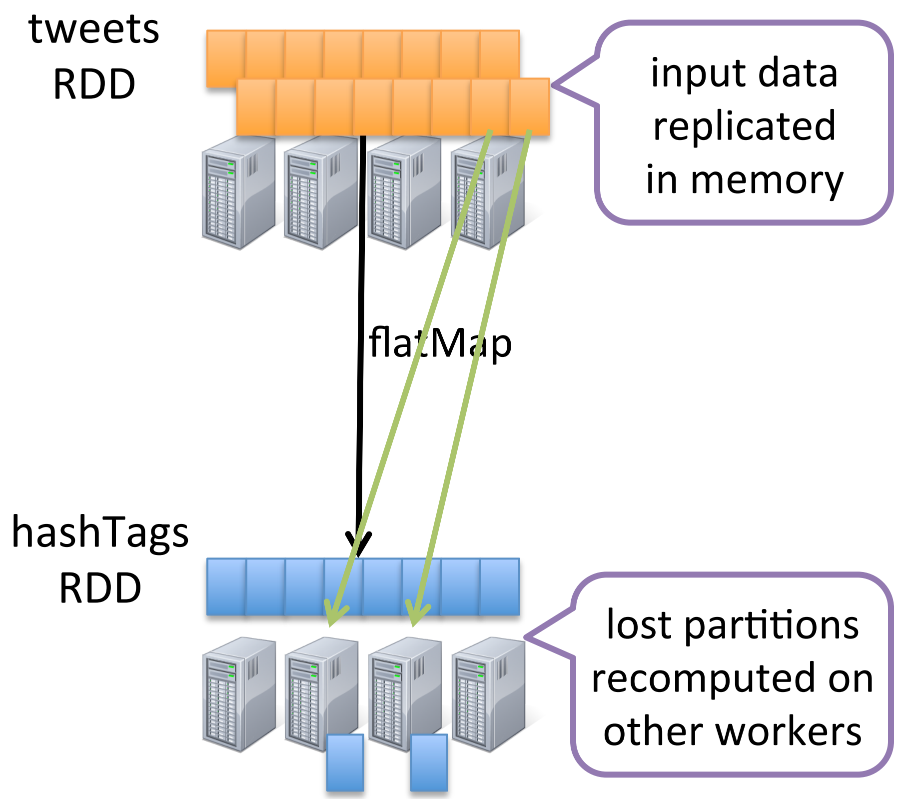


Fault-tolerance

Batches of input data are replicated in memory for fault-tolerance

Data lost due to worker failure, can be recomputed from replicated input data

All transformations are fault-tolerant, and *exactly-once* transformations



RDD operations

Transformation

- Map
- Filter
- GroupBy
- Union
- Intersect
- Join
- Aggregate
- …

Actions

- Reduce
- Collect
- Count
- First
- …

Summary

- RDDs offer a simple and efficient programming model for a broad range of applications

- Leverage the coarse-grained nature of many parallel algorithms for low-overhead recovery

- Resources: https://spark.apache.org/

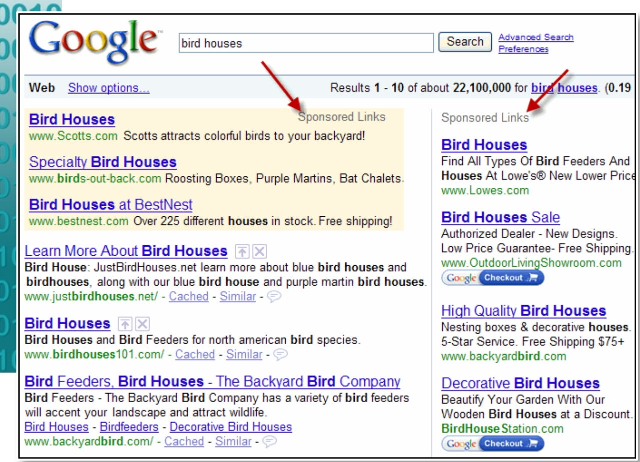
Why stream processing?

Many big-data applications need to process large data streams in real-time

Website monitoring

Fraud detection

Ad monetization



Spark streaming

A stream processing framework

Scales to hundreds of nodes

Achieves second-scale latencies

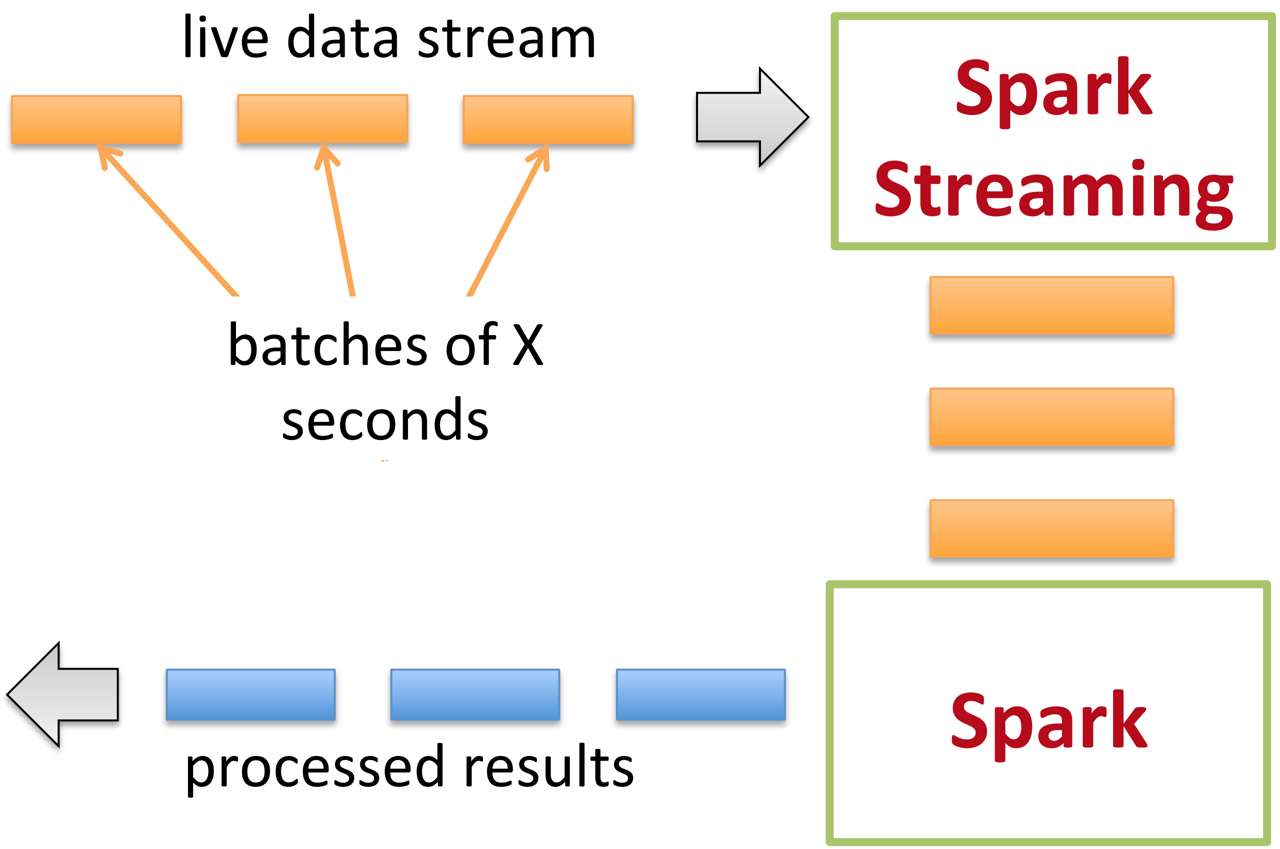
Efficiently recover from failures

Integrates with batch and interactive processing

Consistent “exactly-once” semantics

Spark streaming

- Run a streaming computation as a series of very small, deterministic batch jobs



Input source

Spark streaming provides input from

Kafka, HDFS, Flume, Akka Actors, Raw TCP sockets, etc.

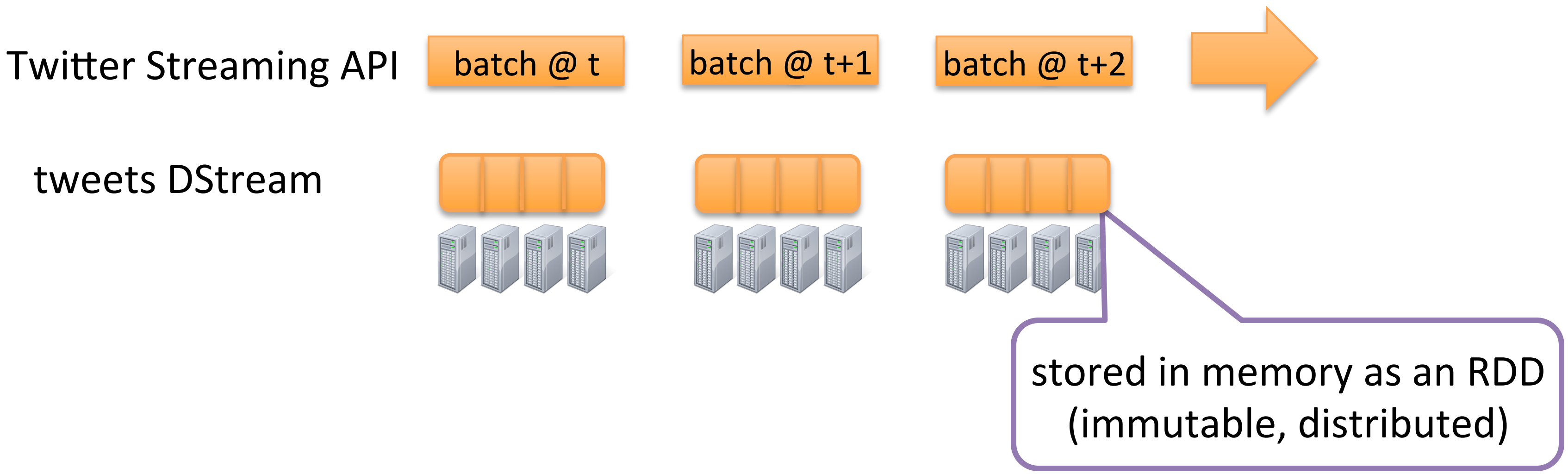
Very easy to write a *receiver* for your own data source

Also, generate your own RDDs from Spark, etc. and push them in as a "stream"

Example: Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
```

DStream: a sequence of RDDs representing a stream of data

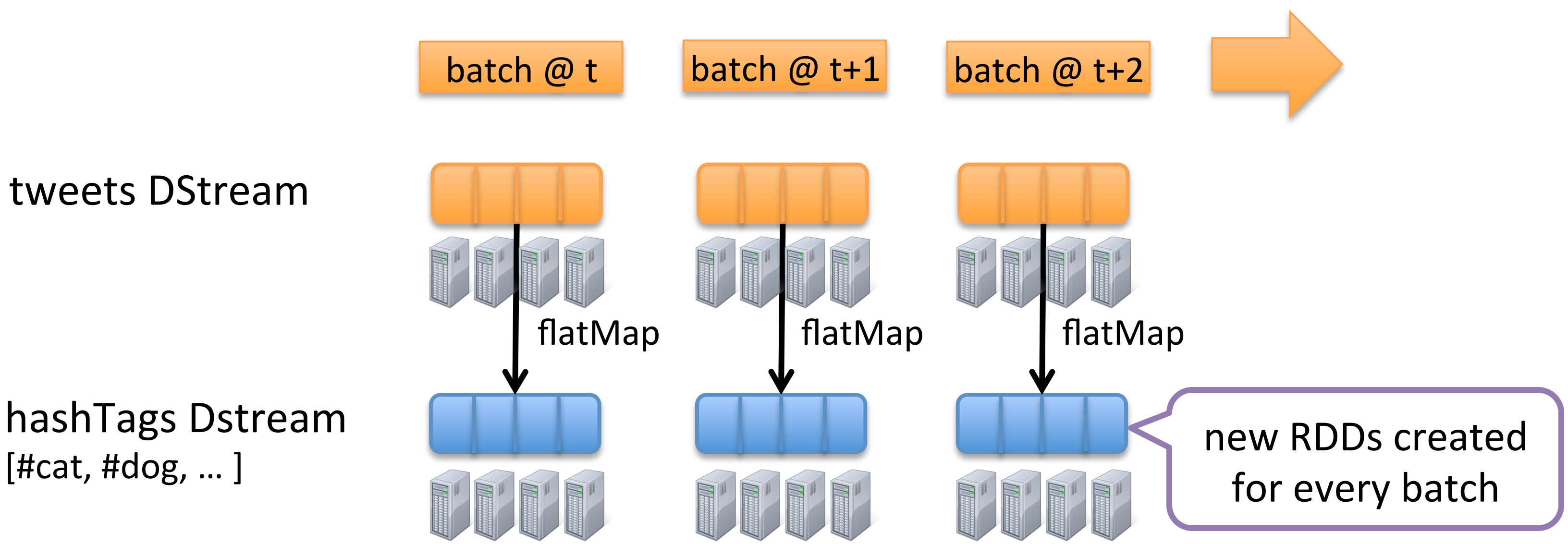


Example: Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap(status => getTags(status))
```

new DStream

transformation: modify data in one DStream to create another DStream



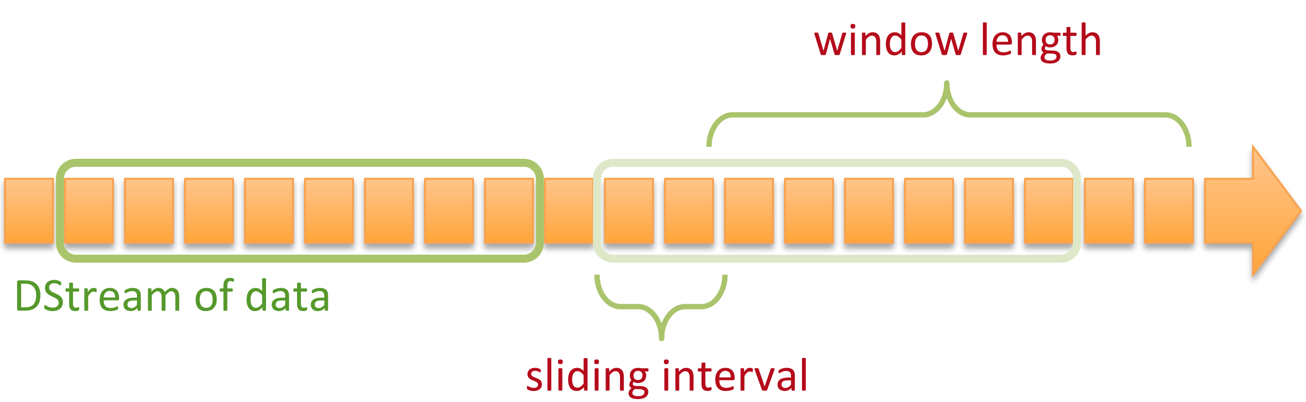
Sliding windows

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap(status => getTags(status))
val tagCounts = hashTags.window(Minutes(1), Seconds(5)).countByValue()
```

sliding window operation

window length

sliding interval



A primer on Apache Flink

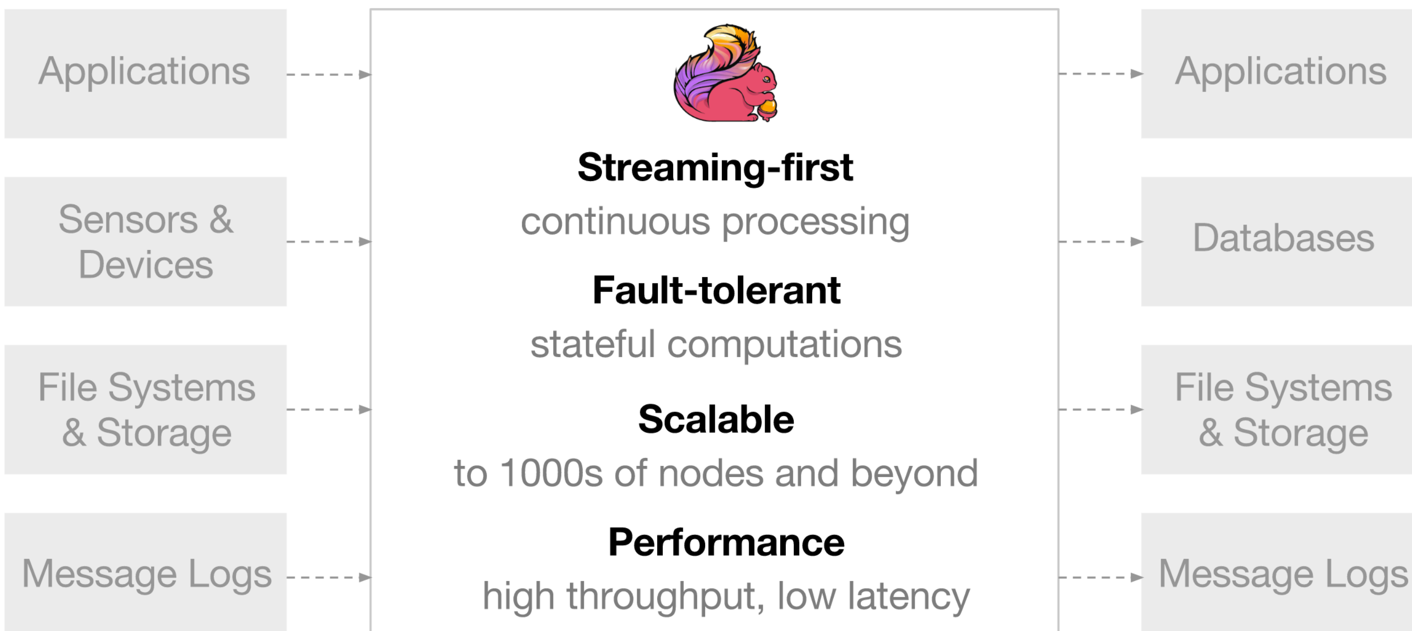
Flink

Pramod Bhatotia
http://homepages.inf.ed.ac.uk/pbhatoti/

THE UNIVERSITY of EDINBURGH

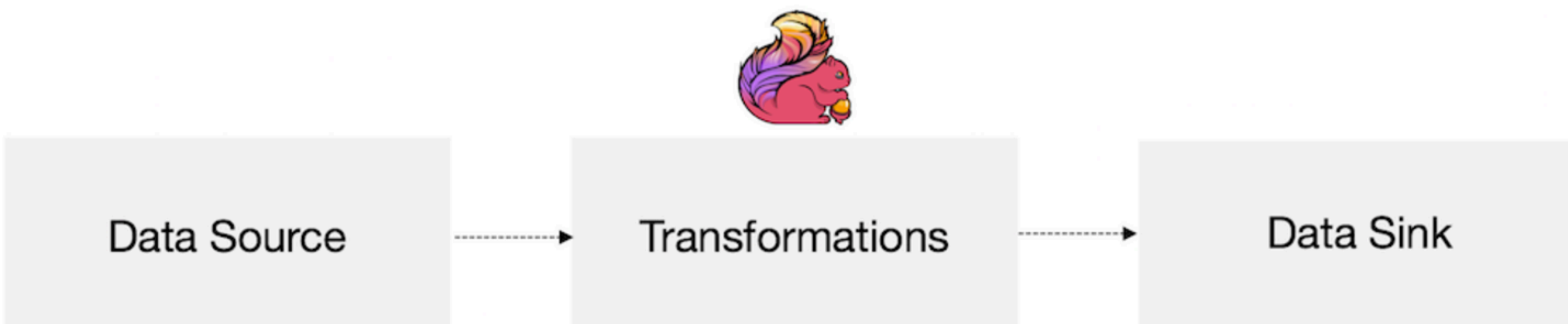
Why Flink?

- Apache Spark follows a batched streaming model
  - Geared towards for high throughput
  - Streaming application requires low-latency too!

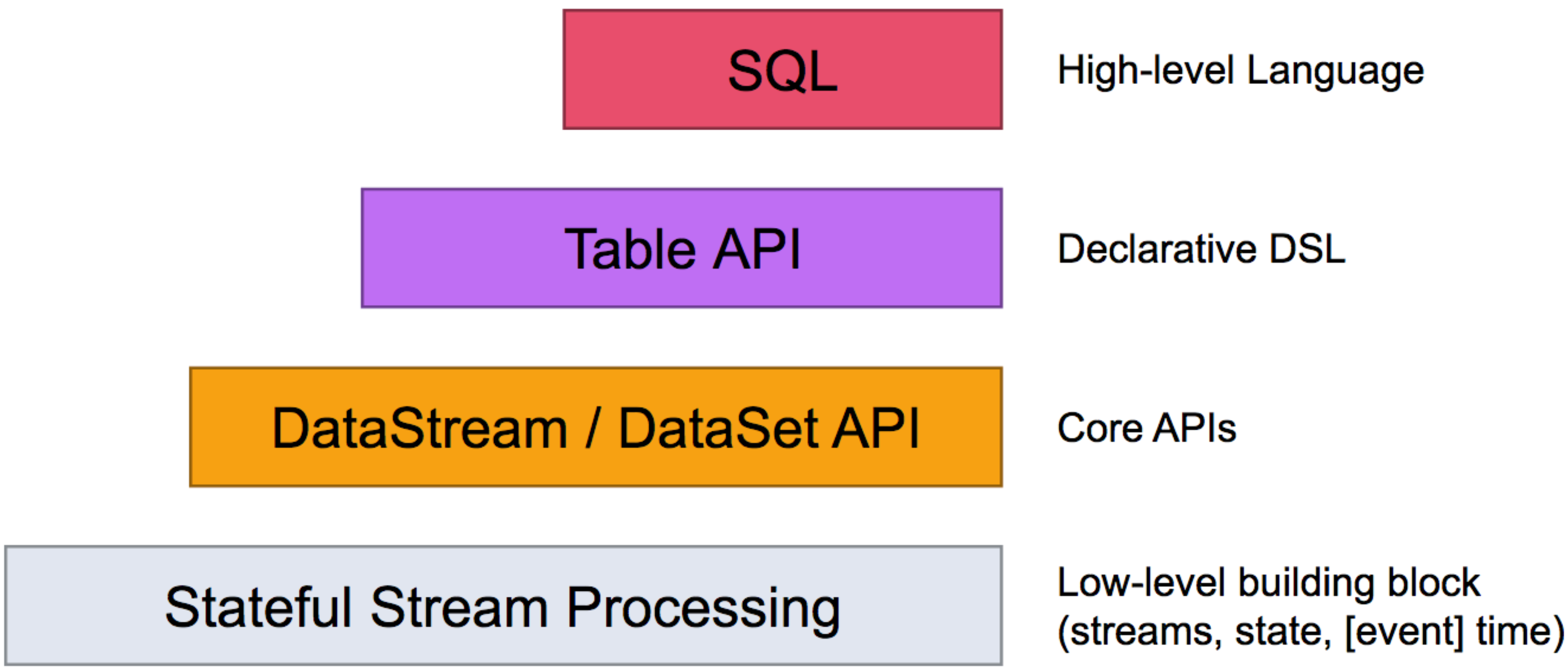


Flink Programs

Continuous execution model



Flink APIs


SQL
High-level Language
Table API
Declarative DSL
DataStream / DataSet API
Core APIs
Stateful Stream Processing
Low-level building block
(streams, state, [event] time)

Example program

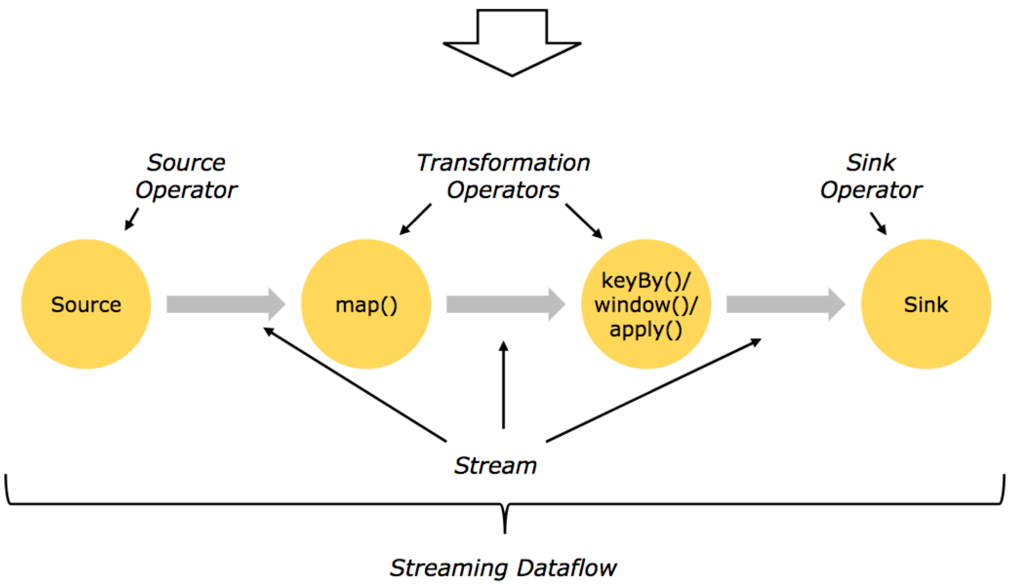
```
DataStream<String> lines = env.addSource(
                               new FlinkKafkaConsumer<>(…));

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

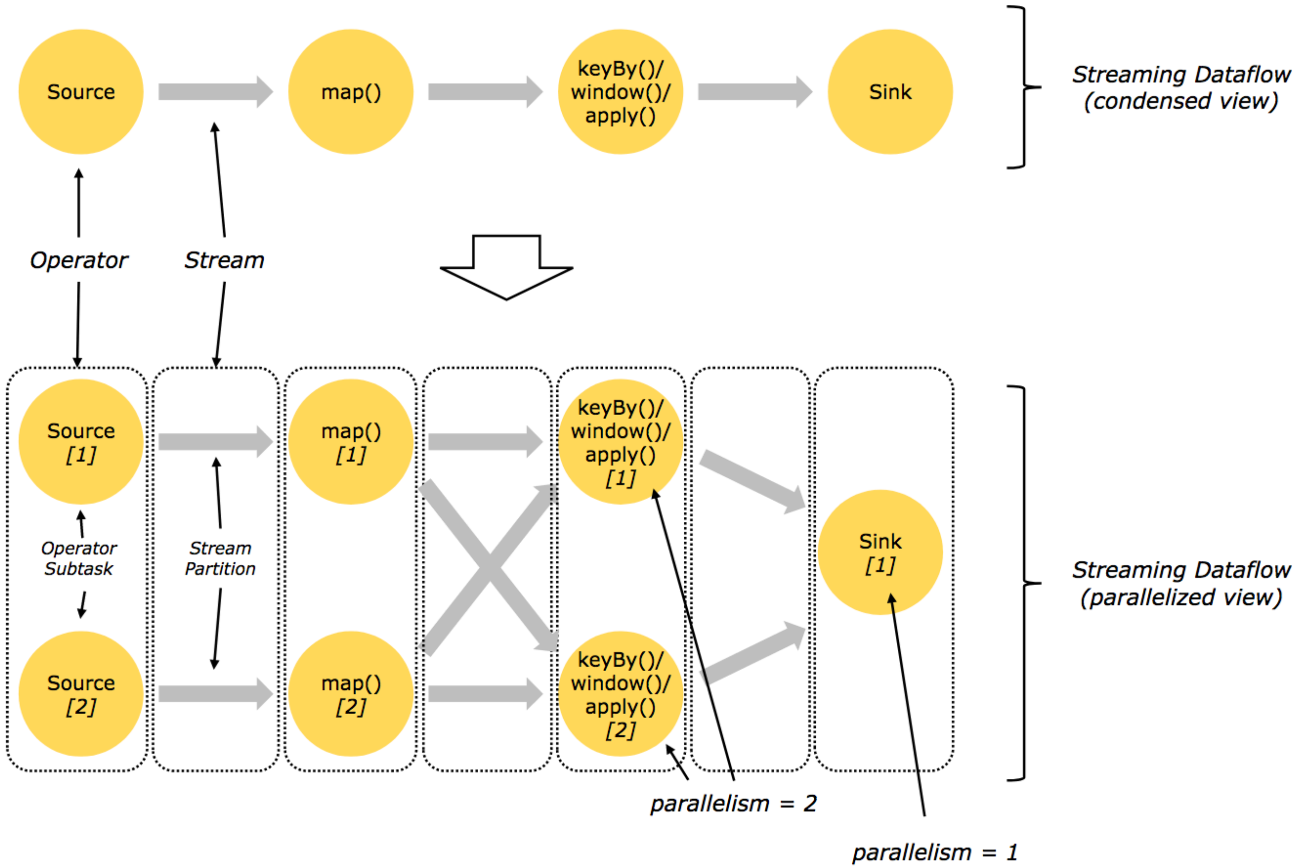
Transformation

Transformation

Sink



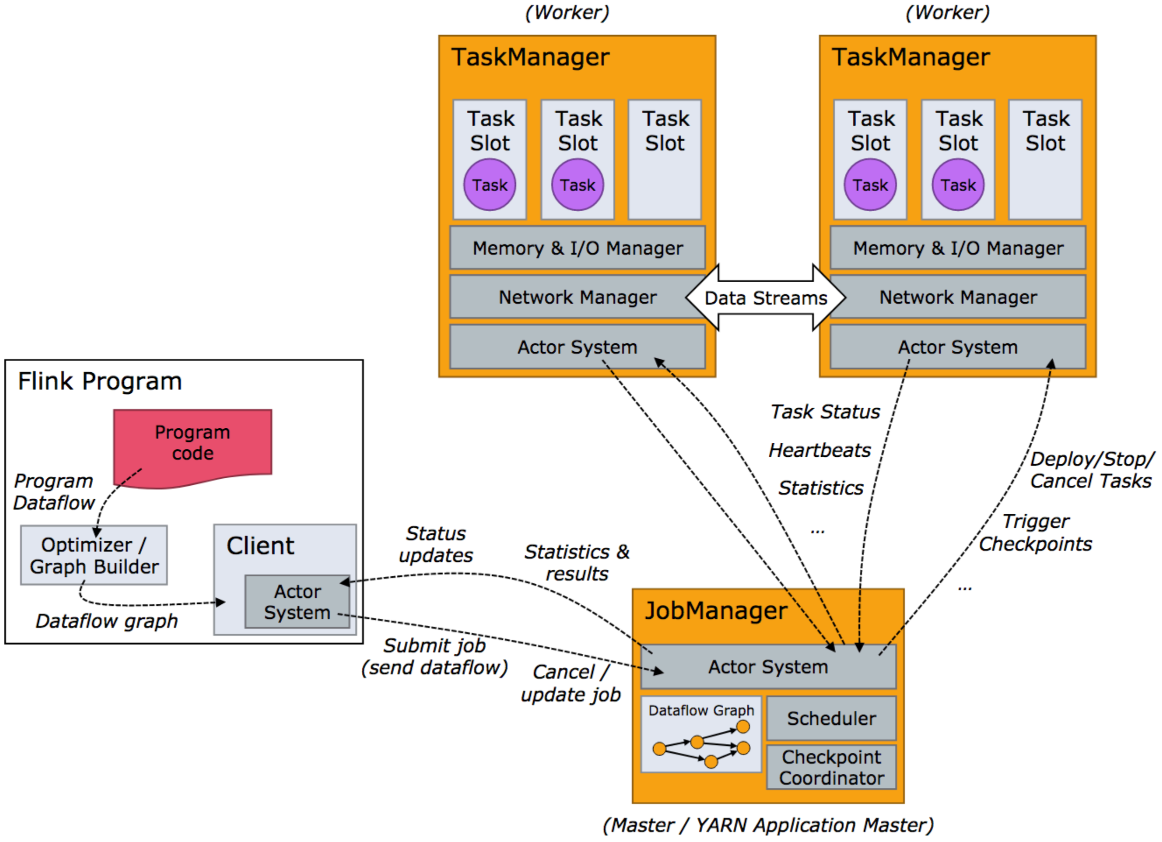
Data-parallel execution



Other features in Flink

- Window
  - Tumbling or sliding windows
  - Time-based or data-driven
- Time
  - Event time (Watermarks)
  - Ingestion time
- Stateful operators
  - Key-value store
- Fault-tolerance
  - Checkpointing
  - Stream replay

Distributed run-time



Summary

- Apache Spark and Flink
  - Unified data engines for batch and stream processing
  - Expose a data-parallel programming model
  - Designed to be scalable, fault-tolerant, strong semantics

- Resources:
  - Spark: https://spark.apache.org/
  - Flink: https://flink.apache.org/

Thanks!

http://homepages.inf.ed.ac.uk/pbhatoti/