Stream Processing

Lecture 7

2018/2019

Table of Contents

Stream processing vs. Complex event processing

- CEP typically:
  - Goal: more oriented towards detecting patterns of events
  - Use high-level declarative language like SQL, or a graphical user interface
  - CEP engine performs the required matching, emitting event when patter is detected
  - Roots: publish-subscribe messaging systems; continuous queries

Stream processing vs. Complex event processing

- Stream processing typically:
  - Goal: more oriented towards producing aggregations and statistical metrics
  - Moving from low-level interfaces to declarative languages
  - Roots: moderns stream processing systems derive from Big data parallel processing frameworks

Distributed Stream Processing Systems

- Why distributed stream processing systems?
  - Scalability
    - Impossible to process all events in a single machine
  - Provide fault-tolerance
    - Need to tolerate server failures
  - Latency
    - Need to provide results fast
  - Data is distributed
    - Processing sensor data

Roadmap for the rest of the course

- Intro to big data frameworks
- Stream processing systems
  - Non-structured programming
  - Structured programming and SQL
  - Continuous streaming
- Stream processing ecosystem
- Storage for streamable data

Table of Contents

Google's MapReduce: summary

- "a programming **model** and an associated **implementation** for processing **large datasets**"
- "runs on a large cluster of **commodity machines** … a typical … computation processes many terabytes of data on **thousands** of machines"
- "a new abstraction that allows us to expresses **simple computations** we were trying to perform but **hides the messy details** of parallelization, fault-tolerance, data-distribution and load-balancing in a library"

Programming model

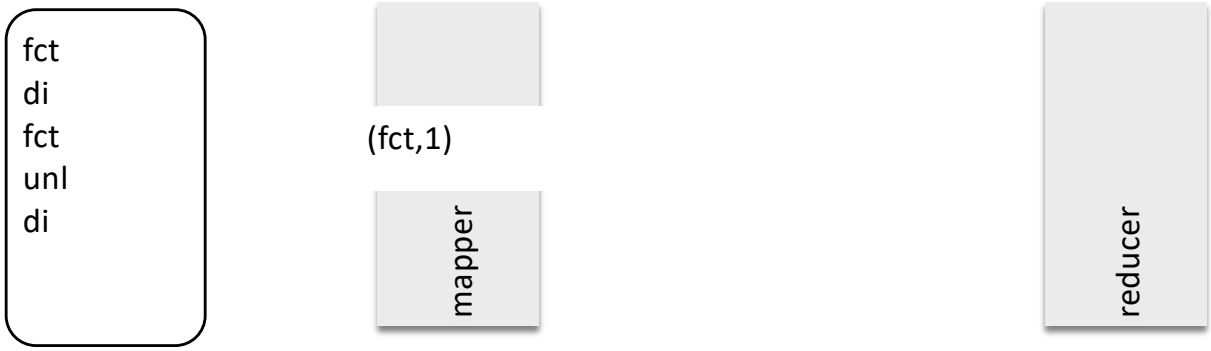
- Sequence of map and reduce stages
- Map: processes input (files); emit tuples
- Reduce: process tuples grouped by key; Emit tuples

Programming model... working

- Example: count the number of times each word appears in a document

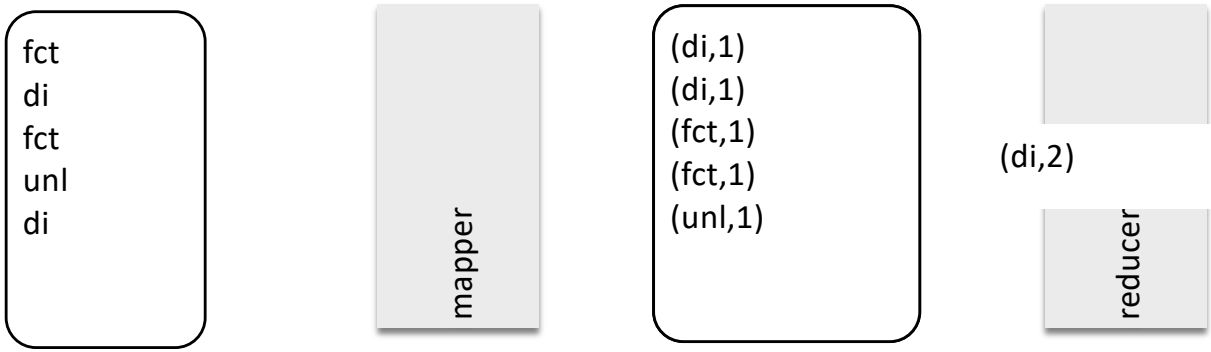
```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, “1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Programming model… working



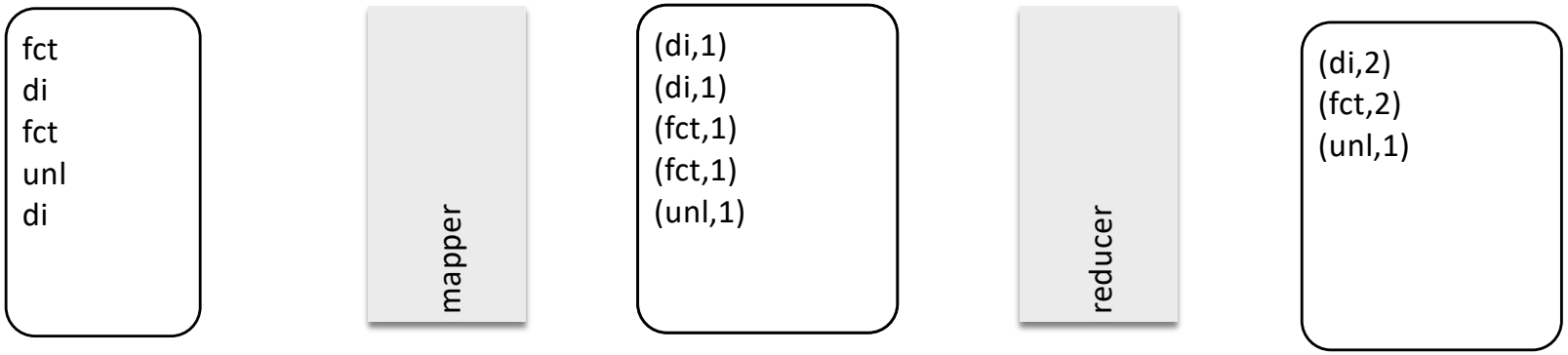

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, “1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Programming model… working




```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, “1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Programming model… working

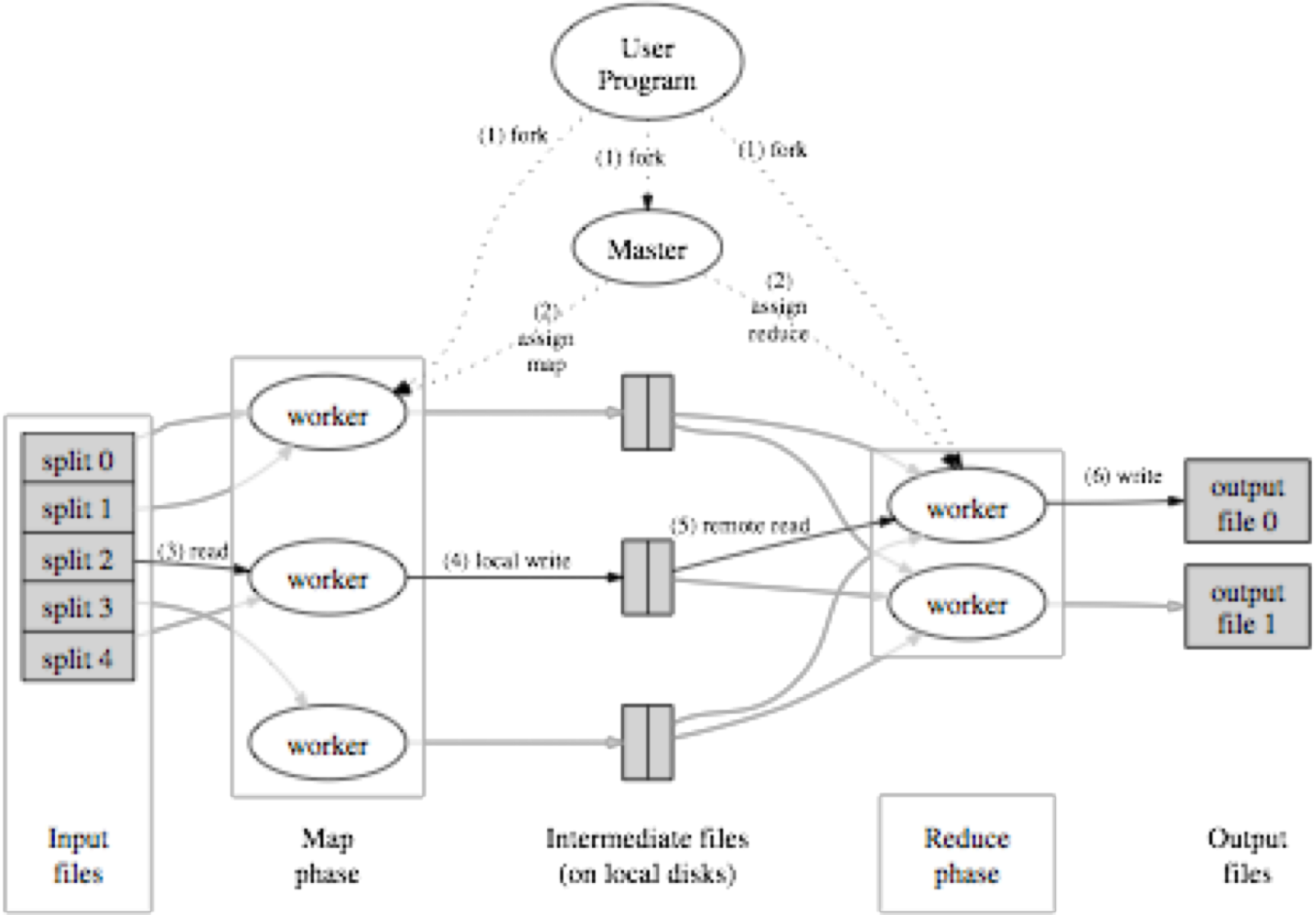



```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, “1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

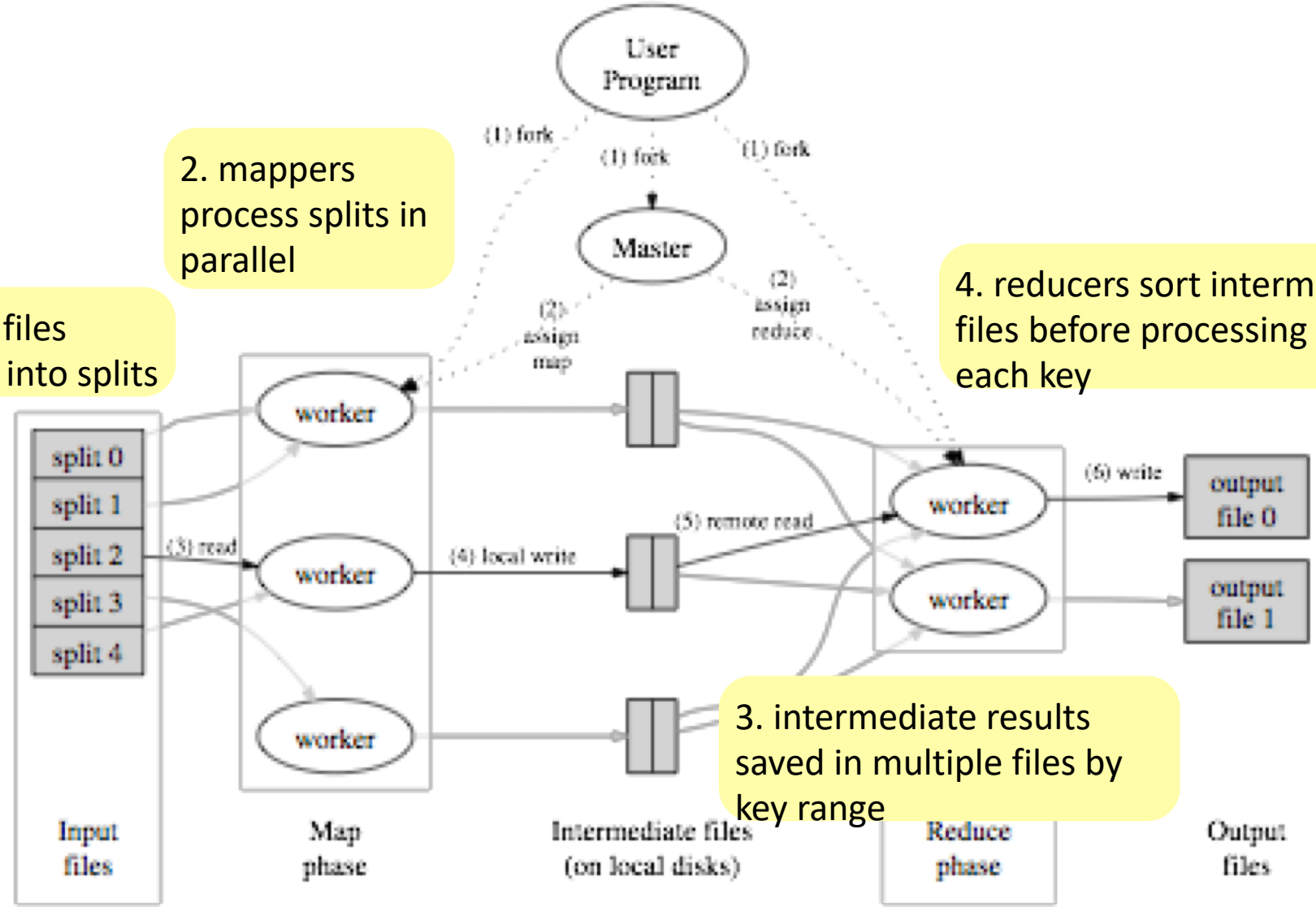
Programming model is not everything

- Programming model is simple, but…
- …how to run computations efficiently?

Map-reduce execution model



Map-reduce execution model



2. mappers process splits in parallel

1.input files divided into splits

4. reducers sort intermediate files before processing values for each key

3. intermediate results saved in multiple files by key range

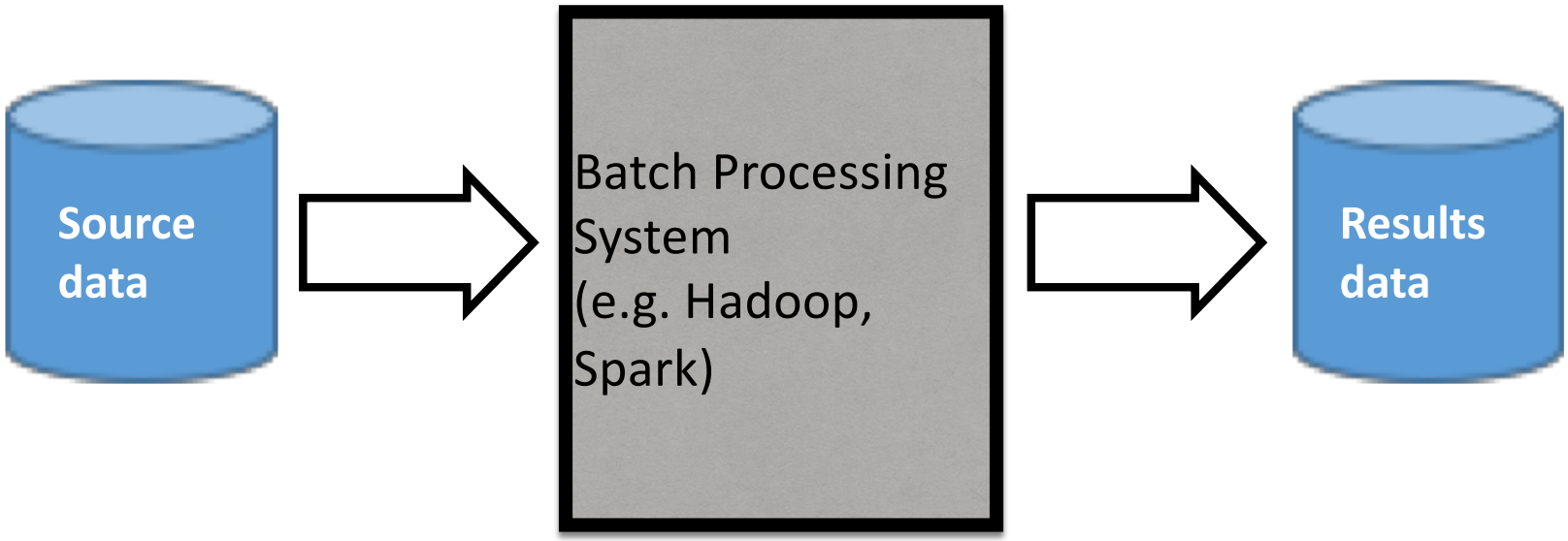
Limitations of map-reduce

- Scalable, but slow
  - Data stored on disk after each step
- Low-level programming
  - Simple programming model with no abstractions for helping writing programs
- Batch processing model not adequate for some applications
  - Need stream processing

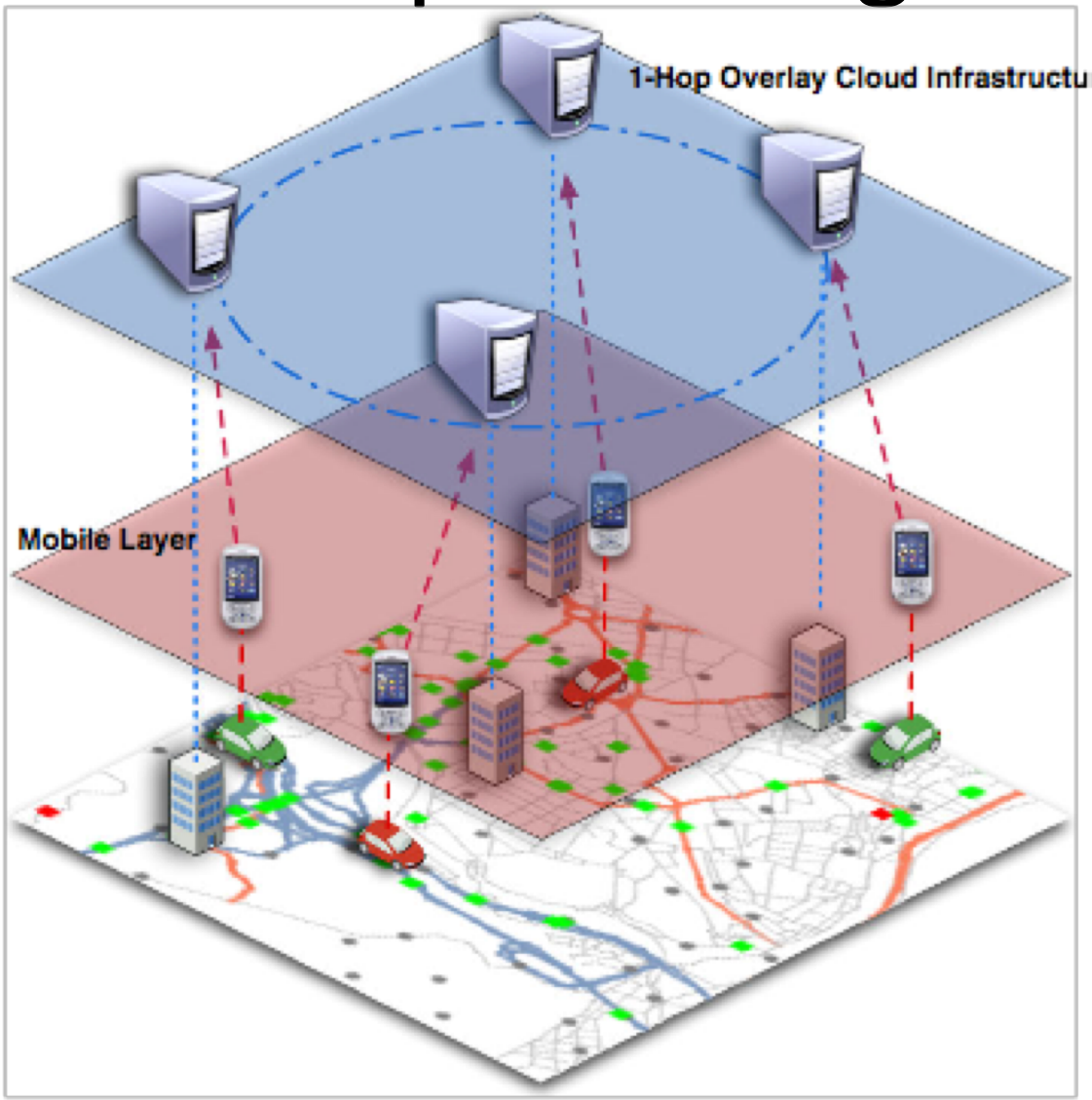
Table of Contents

Big Data / Batch processing

- All data known at the time of processing
- Goal: Execute computation over data and produce result
- Problem: what if new data arrives continuously, and new results should be computed continuously?



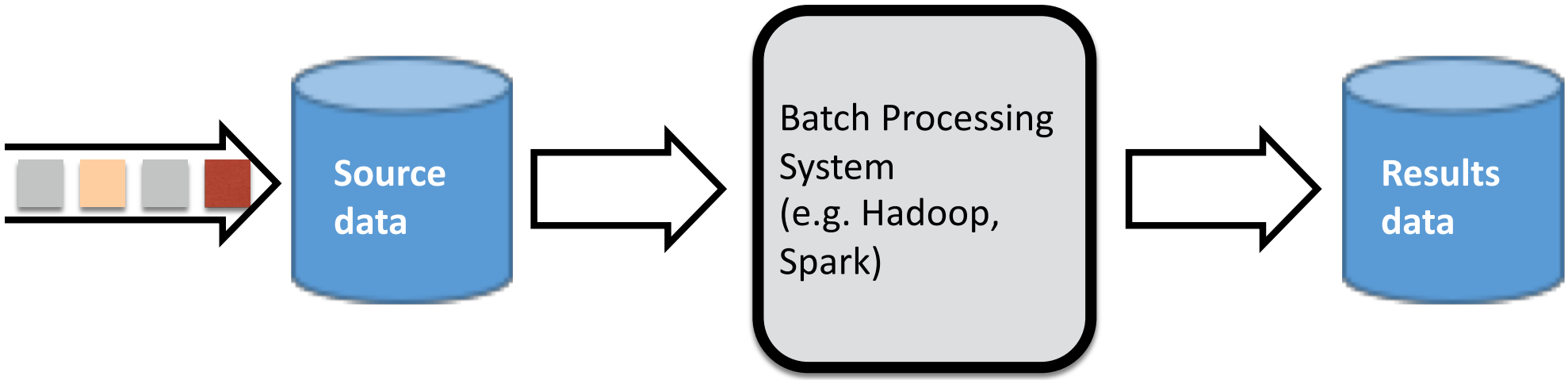
Examples of Big *Streaming* Data



Producing information on traffic based on information collected from users' mobile phones

Big Streaming Data

- Can we use (batch) big data processing tools?
- Save data as it arrives
- Execute computation periodically - e.g. every hour
- Problems?
- Long delay for results, computation not incremental, …

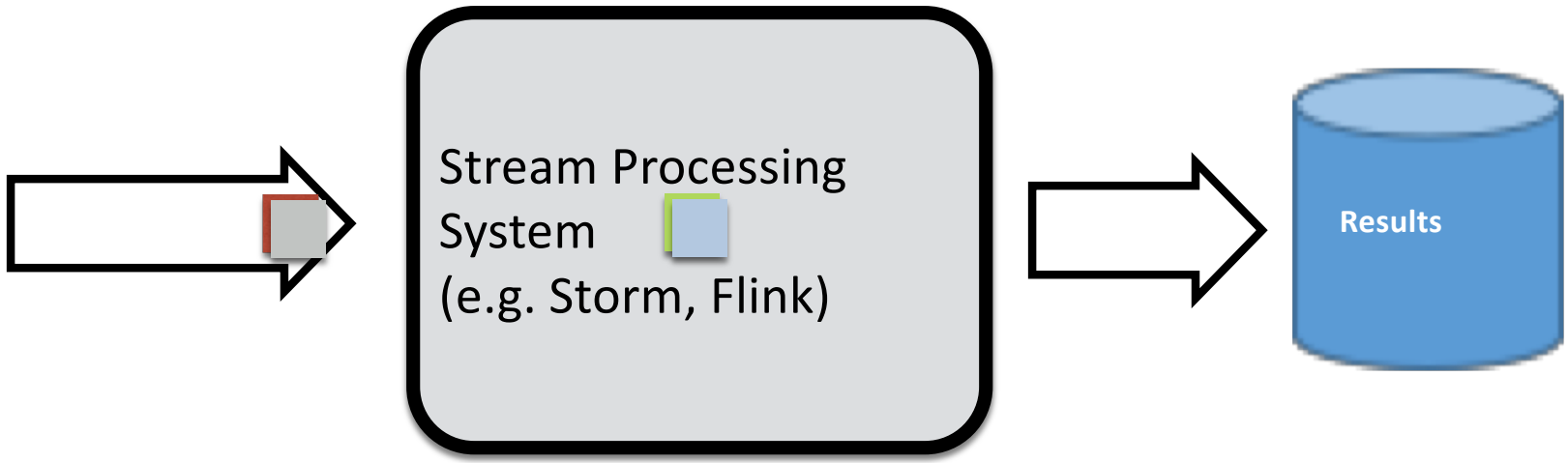


Big Streaming Data: requirements

- Need to process data as it arrive (or at most with a very small delay)
- Need to be able to process data from multiple sources
- Need to tolerate faults

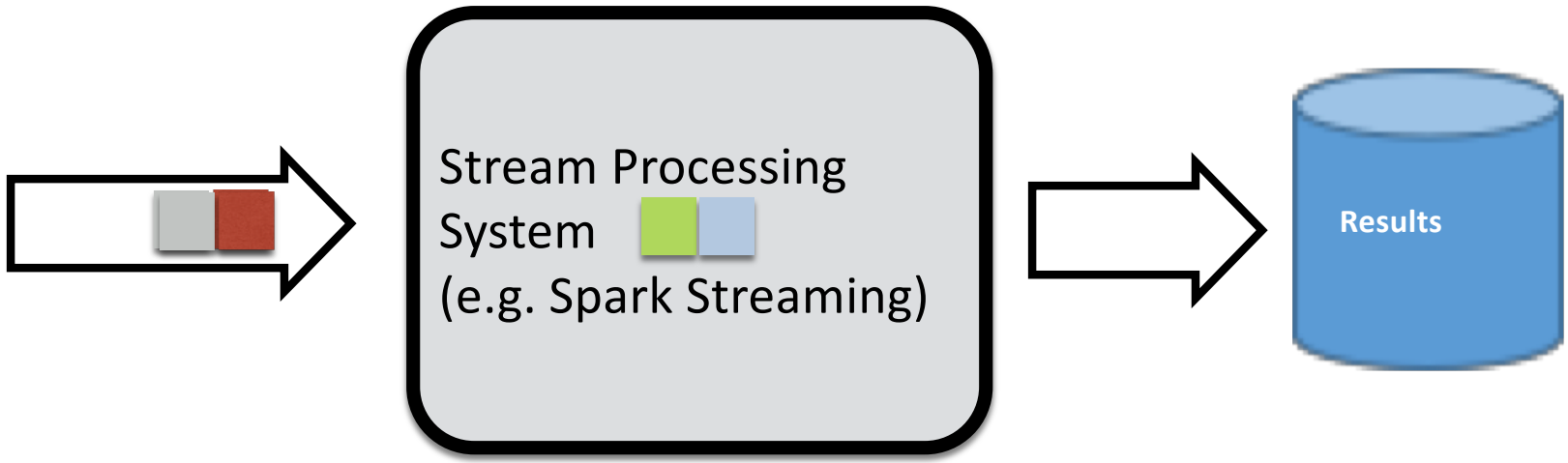
Two processing models (1)

- Continuous
  - Each tuple processed as it arrives
  - Processing system may keep state for executing window computation and incremental computation



Two processing models (2)

- Mini-batches
  - Tuples received for each X ms grouped in a mini-batch
  - Process mini-batches
  - Processing system may keep state for executing window computation and incremental computation



Stream processing: some issue

- Semantics
  - Reasoning about time
  - Joining multiple streams
- Performance
  - Latency
  - Fault tolerance
  - Sampling

Reasoning about time

- Stream processing often need to deal with time, but notion is tricky.
  - E.g.: compute X over the last five minutes. What does it mean?

Reasoning about time: event time

- Use the time of the event. Problems?

- Delay to start processing
  - Delays of event propagation
  - Have to deal with stragglers
    - Ignore straggler events
    - Issue correction of results
  - Have to deal with failures

Reasoning about time: process time

- Use the time the event reached the stream processing system. Problems?

- Combine events from different time periods
  - Delays of event propagation
  - Fault tolerance

Joining multiple streams

- Often needs to join events from multiple streams
  - E.g. in a website, join search query with click on search.
- Stream-stream join
  - Need to be able to join an event with an event in the past
- Stream-table join
  - Store data in a table; join stream with data in a table

Stream processing: some issue

- Semantics
  - Reasoning about time
  - Joining multiple streams
- Performance
  - Latency
  - Fault tolerance
  - Determinism
  - Sampling

Latency in stream processing

- Some applications impose real time or bounded latency constraints on processing
- Results need to be produced at a rate compatible with the ingress rate
- Effects of fault-tolerance should be transient (and perceived as jitter, rather than accumulate).
- Partitioning can speed up computations, via parallelism, but can lead to some stragglers.
  - Not easy to anticipate. May be too late upon detection
  - Sensitive to input / improper partitioning

Fault tolerance in stream processing

- Batch processing
  - In worst case, can tolerate faults by recomputing everything
- Stream processing
  - Not usually feasible to replay the stream(s) from the very beginning
  - Implies some form of periodic checkpointing (or replication)

Determinism in stream processing

- Redundant processing is useful in some scenarios…
  - Can provide fault tolerance;
  - Mitigate the impact of stragglers in latency.
- Processing the same stream twice should yield the same stream of results.
- Algorithm should not depend on factors external to the store data

Sampling in stream processing

- Execute processing over a part of the data. Why is this acceptable?
- For high ingress data rates, sampling may be employed to meet desired processing latency
- Sampling is not straightforward and impacts on the accuracy and interpretation of the processing results

Systems for stream processing

- Continuous processing
  - Apache Storm
    - Open sourced by Twitter
    - API: proprietary, SQL-like
  - Apache Flink
    - API: proprietary, table-based, SQL-like

- Mini-batch processing
  - Spark streaming
    - API: proprietary, table-based, SQL-like

Table of Contents

Spark Streaming

- Spark Streaming is an extension of the Spark batch processing systyem to enable scalable, high-throughput, fault-tolerant stream processing of live data streams.

Matei Zaharia, et. al. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In Proc. SOSP'13.
http://people.csail.mit.edu/matei/papers/2013/sosp_spark_streaming.pdf

http://spark.apache.org/streaming/

Apache Spark

- Apache Spark provides in-memory, fault-tolerant distributed processing
- Spark programs can comprise multiple chained data transformation steps.
  - each step produces a RDD (Resilient Distributed Dataset)
    - RDD is the core abstraction

Apache Spark: Data Model

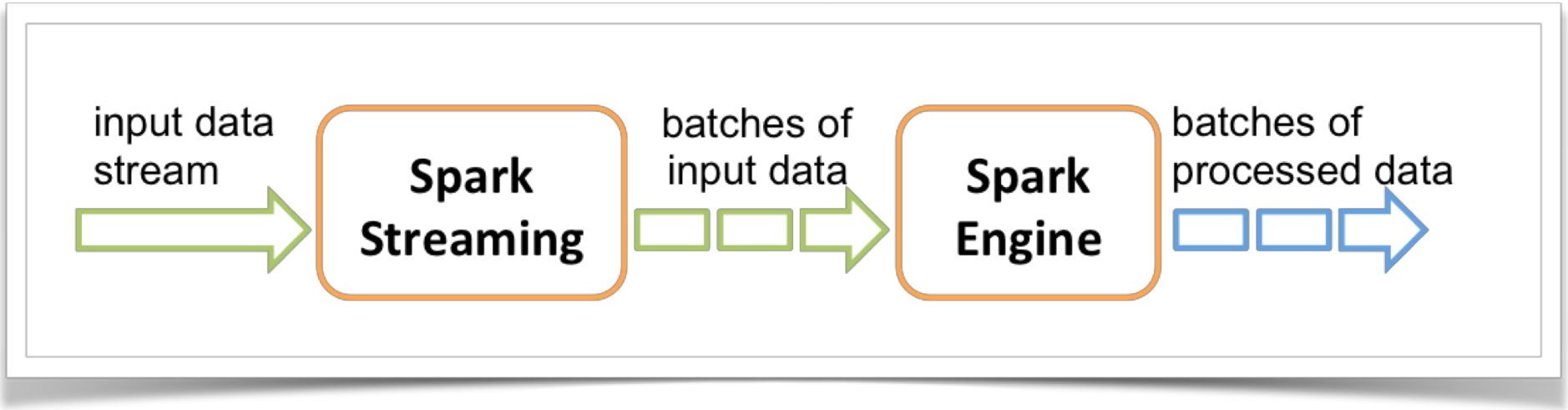
- RDDs are immutable data
  - logically a RDD is a set of data tuples;
  - physically distributed (partitioned) across many nodes;
  - upon a failure (or cascade of failures), RDDs can be recreated automatically and efficiently from the dependencies.

Apache Spark: Data Model

- Spark programs describe the flow of transformations that creates an RDD from another, usually in several steps.
- Spark programs, therefore, encode the dependencies among the various RDDs
  - this is known as the lineage graph

Core idea

- Discretize a continuous stream of data into a sequence of small, discrete batches, each represented as a RDD
- Each of the resulting RDDs is processed in Spark to produce an output stream of RDDs



SparkStreaming : input data stream

- Spark already provides a number of standard data input streams.

Programming Model

- Incoming stream data is split and batched according to fixed time intervals, resulting in a new RDD for each interval period - a discrete stream of RDDs.
  - (the size of the discrete RDDs varies as it depends on the ingress rate)
- RDDs are processed in a similar way as in Spark – by applying transformations.
- Output operators/actions provide several ways to consume the processed stream

Example: skeleton

```
import pyspark
from pyspark.streaming import StreamingContext

sc = pyspark.SparkContext('local[*]')
try :
    ssc = StreamingContext(sc, 1)

    lines = ssc.socketTextStream("localhost", 7777)

    counts = ...

    counts.pprint()
    ssc.start()
    ssc.awaitTermination()
except err:
    print( err)
    sc.stop()
    ssc.stop()
```

Example

- In the example, we will consider a set of log files containing information for web accesses.

date IP_source return_value operation URL time

2016-12-06T08:58:35.318+0000 37.139.9.11 404 GET /codemove/TTCENCUFMH3C 0.026

Example: filter function

- **filter**(*func*)
- Return a new DStream by selecting only the records of the source DStream on which *func* returns true.

```
lines = ssc.socketTextStream("localhost", 7777)

counts = lines.filter(lambda line : len(line) > 0 )
```

Example: map function

- **map**(*func*)
- Returns a new DStream by passing each element of the source DStream through a function *func*.

```
lines = ssc.socketTextStream("localhost", 7777)

counts = lines.filter(lambda line : len(line) > 0 ) \
              .map(lambda line: line.split(' ')) \
              .map(lambda vals: (vals[1],1))
```

Example: reduceByKey function

- **reduceByKey**(*func*, [*numTasks*])
- When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function.

```
lines = ssc.socketTextStream("localhost", 7777)

counts = lines.filter(lambda line : len(line) > 0 ) \
            .map(lambda line: line.split(' ')) \
            .map(lambda vals: (vals[1],1)) \
            .reduceByKey(lambda a, b: a+b)
```

Example: transform function

- **transform**(*func*)
- Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream. This can be used to do arbitrary RDD operations on the DStream.

```
lines = ssc.socketTextStream("localhost", 7777)

counts = lines.filter(lambda line : len(line) > 0 ) \
              .map(lambda line: line.split(' ')) \
              .map(lambda vals: (vals[1],1)) \
              .reduceByKey(lambda a, b: a+b) \
              .transform(lambda rdd: \
              rdd.sortBy(lambda x: x[1], ascending=False))
```

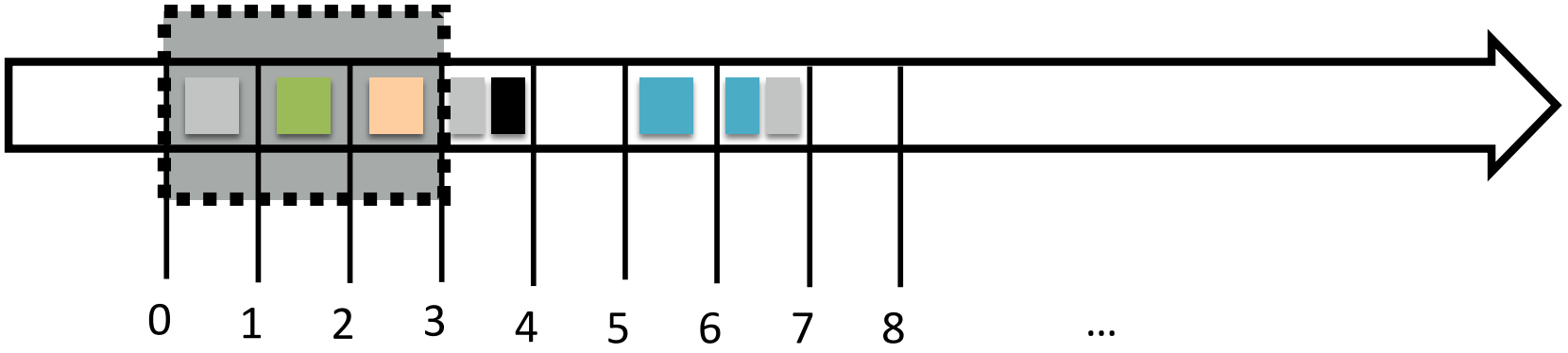
Example: join function

- **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.

```
otherCounts = otherLines.filter(lambda line : \
                 len(line) > 0) \
              .map(lambda line: line.split(' ')) \
              .map(lambda vals: (vals[1],1))

counts = lines.filter(lambda line : len(line) > 0 ) \
              .map(lambda line: line.split(' ')) \
              .map(lambda vals: (vals[1],1)) \
              .join( otherCounts)
```

Windowing

- In some cases, we want to process data over a larger interval — for example, process data of last 3 seconds, but produce results every second.
- Windowing groups the input stream RDDs that fall inside a larger time interval into one compound RDD. It advances automatically with passing time.
- SparkStream supports this by using a sliding window.
  - s.window("3s") would then output RDDs comprising the records in intervals: [0,3), [1,4), [2,5), …



Example: window

- **window**(*windowLength*, *slideInterval*)
- Define window of length *windowLength* at every *slideInterval* period

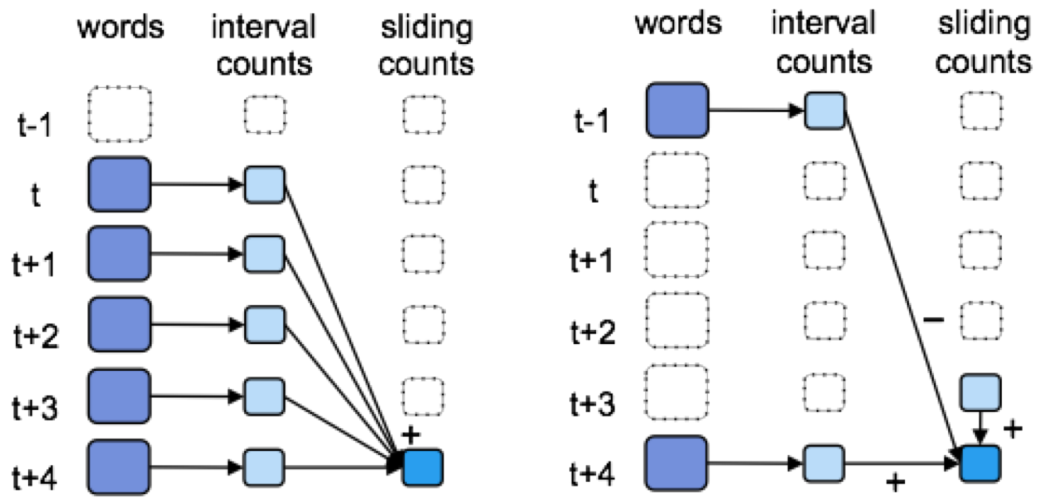
```
lines = ssc.socketTextStream("localhost", 7777)

lines = lines.window(3,1)

count = lines. ...
```

Incremental Aggregation (1)

- Incremental aggregation over a sliding window is possible using the reduceByWindow operator and an associative merge function:
  - x.reduceByWindow("5s", (a, b) -> a + b)
  - first aggregates data for each 1 second interval; then merge results into a single RDD result.
  - as the window slides, and one interval is dropped and another is added, it is not necessary to repeat the reduction of intervals already processed in the previous window, but still requires the aggregation of 5 intermediate results.

Incremental Aggregation (2)

– Optimized incremental aggregation is available for invertible merge functions:
  - **x.reduceByWindow("5s", (a, b) -> a + b, (a, b) -> a - b)**
  - as the window slides, the dropped interval is removed from the total and only the new interval that enters the window is added.



(a) Associative only (b) Associative & invertible

State tracking

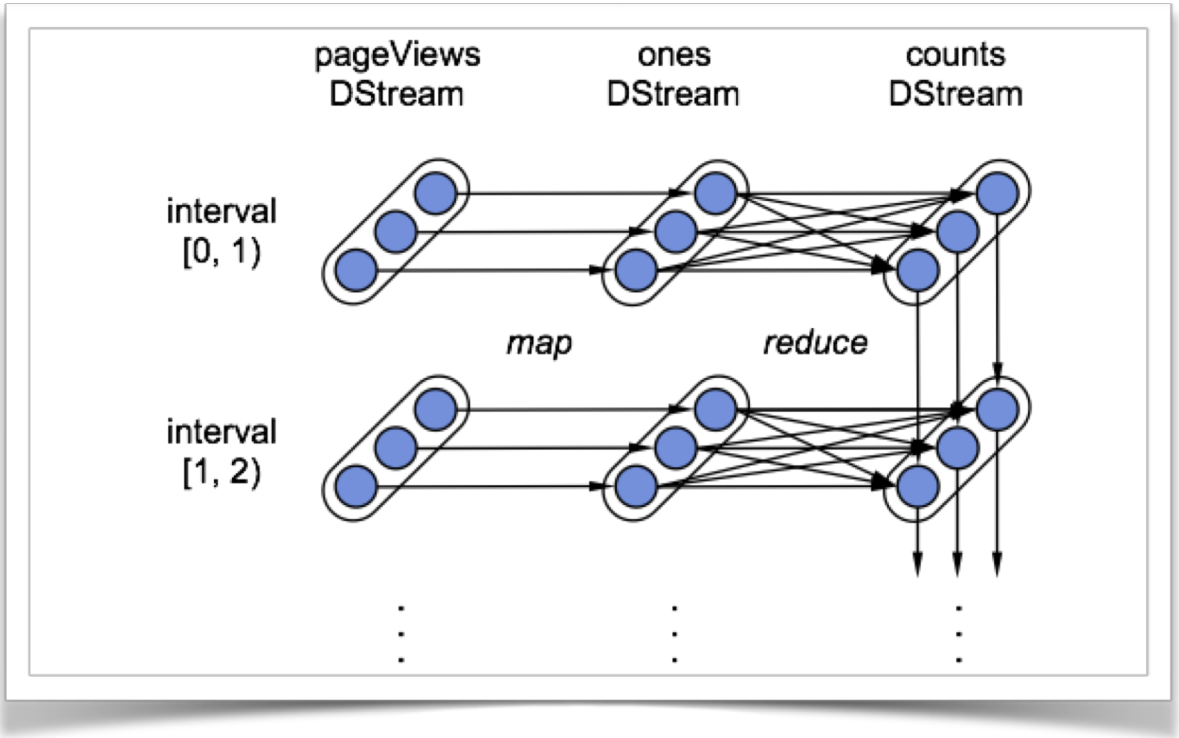
- Computations can also carry arbitrary state across the whole discretized stream [and not just over a fixed size window]
- ***updateStateByKey****(func)*
- returns a new "state" Stream RDD, where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key.
  - This can be used to maintain arbitrary state data for each key, for the whole stream.
  - Not all keys need to maintain state at all times.

SparkStreaming : fault-tolerance

- SparkStreaming can recover from faults, by rebuilding missing RDDs across several nodes, in parallel.
  - Uses lineage information to allow minimal re-computation of the RDDs lost in a fault.
  - Independent sections of the lineage graph can be processed and recovered in parallel

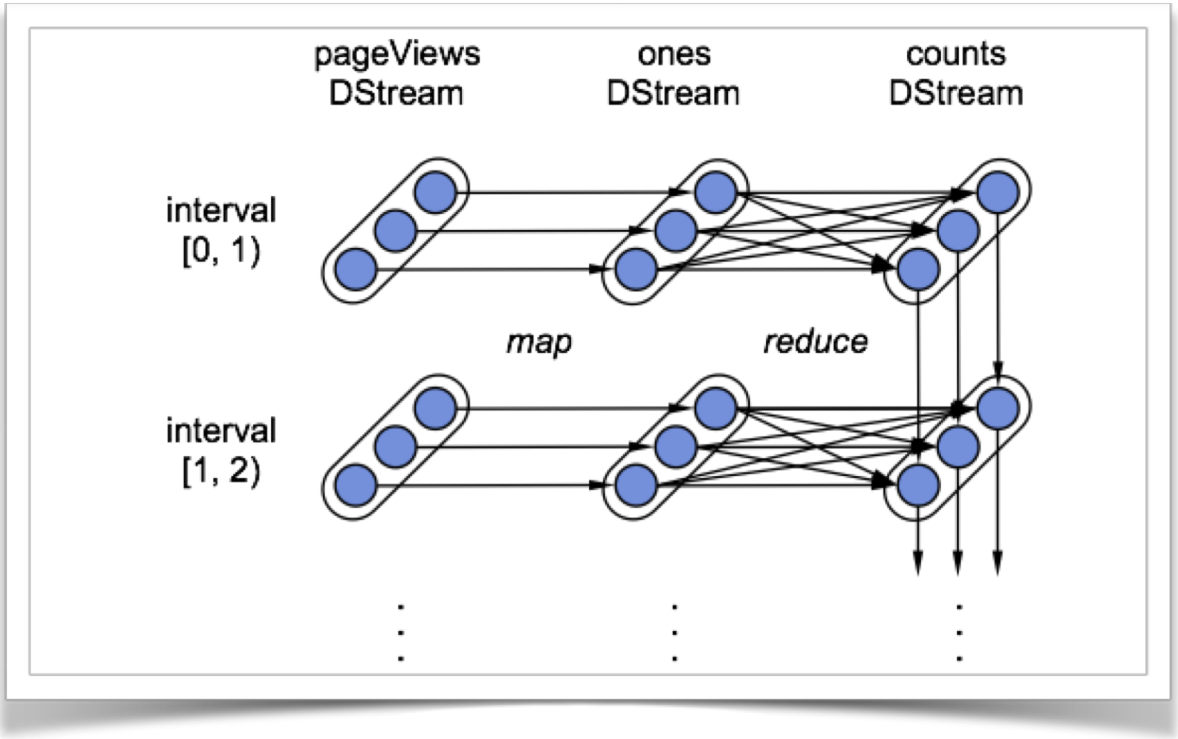
SparkStreaming : lineage graph

– The lineage graph in SparkStreaming is logically static but dynamic in physical terms.



Fault-tolerance

– Can recover from faults, by rebuilding missing RDDs
  - Maintains lineage information to minimize re-computation
  - Maintains checkpoints



SparkStreaming : checkpointing

– The lineage graph in SparkStreaming is logically static but dynamic in physical terms
  - ***updateStateByKey,*** creates a lineage graph that goes back to the first RDD of the input stream
  - Fault recovery would require persisting the entire stream.
  - Instead, SparkStreaming performs **periodic checkpointing**, by saving internal state to the filesystem
  - Upon a failure, RDDs are re-created from the last valid checkpoint.

SparkStreaming : limitations

- SparkStreaming operates over discrete portions of the stream that are created **based on time**, with a **fixed** minimum latency
  - Fixed duration time intervals do not generally translate into constant sized RDDs… it depends on the ingress rate
- SparkStreaming does not **naturally** support processing streams at a granularity that is not time oriented…
  - For instance, it is difficult to process each stream item independently and update the resulting stream as each item arrives

Transformations (for reference)

Full list at:
http://spark.apache.org/docs/latest/streaming-programming-guide.html#transformations-on-dstreams

Transformation	Meaning
map(*func*)	Return a new DStream by passing each element of the source DStream through a function *func*.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items.
filter(*func*)	Return a new DStream by selecting only the records of the source DStream on which *func* returns true.
union(*otherStream*)	Return a new DStream that contains the union of the elements in the source DStream and *otherDStream*.
count()	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.
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.
reduceByKey(*func*, [*numTasks*])	When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function.
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.

Bibliography

- Martin Kleppmann. Designing data-intensive applications. Chapter 11.

- Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: fault-tolerant streaming computation at scale. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13). ACM, New York, NY, USA, 423-438. DOI: https://doi.org/10.1145/2517349.2522737