L11: Spark Streaming and GraphX

ANLY 502: Massive Data Fundamentals
Simson Garfinkel & Marck Vaisman
April 24, 2017

GEORGETOWN UNIVERSITY

Agenda

Administrivia

- Last lecture today!
- Next week, project presentations - looking forward to them
- Q08 has been posted - will update the due date
- Q10 (scalable machine learning) and Q11 (today) will be posted tomorrow
- A5 grading will be completed by the end of the week

Data streams, Spark Streaming, Kafka

Demo/Lab

Break

GraphX on Spark

Demo/Lab

Time for Q&A about anything else of interest

Spark Streaming

Up until now, it's been batch processing on static datasets

We've used:

- Hadoop MapReduce
    - *Streaming with mrjob*
- Spark

Datasets have been static, as have been computations and analysis

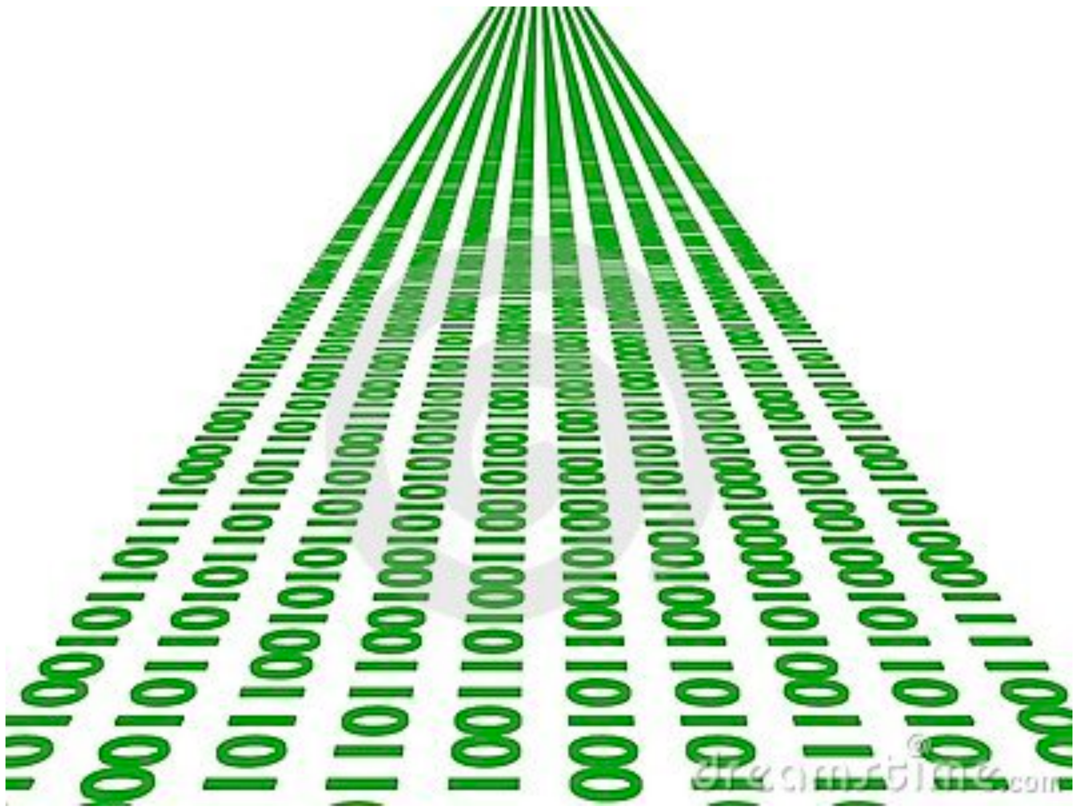
What is a data stream?

Streaming data is a continuous group of data records generated from sources like sensors, server traffic and online searches

- user activity on websites
- monitoring data
- server logs
- event data

Streaming data processing helps with:

- live dashboards
- real-time online recommendations
- fraud detection



Challenges with stream computations

Computations on certain metrics on streams can be challenging because of the need to iterate over the entire dataset.

- Quantiles: need to sort the data
- Mean = sum of values / count (all data)

When we add a new item to the stream, we increment the count

Before Spark Streaming

Case study: Conviva, Inc

Conviva helps top broadcasters, operators, and content owners experience excellence and create more profitable streaming businesses with their products

They needed to process real-time monitoring of online video metadata.

Two processing stacks:

- Custom built distributed stream processing system
  - *1000s of complex metrics on millions on video sessions*
  - *Required many dozens of nodes for processing*
- Hadoop backend for offline analysis
  - *Generating daily and monthly reports*
  - *Similar computation requirements as streaming system*

Twice the effort for live streaming data

Because of the custom stream processing and separate batch processing:

- Twice the effort to implement any new function
- Twice the number of bugs
- Twice the headache

Requirements for stream processing

Scalable to large clusters

Second-scale latencies

Simple programming model

Integrated with batch and interactive processing

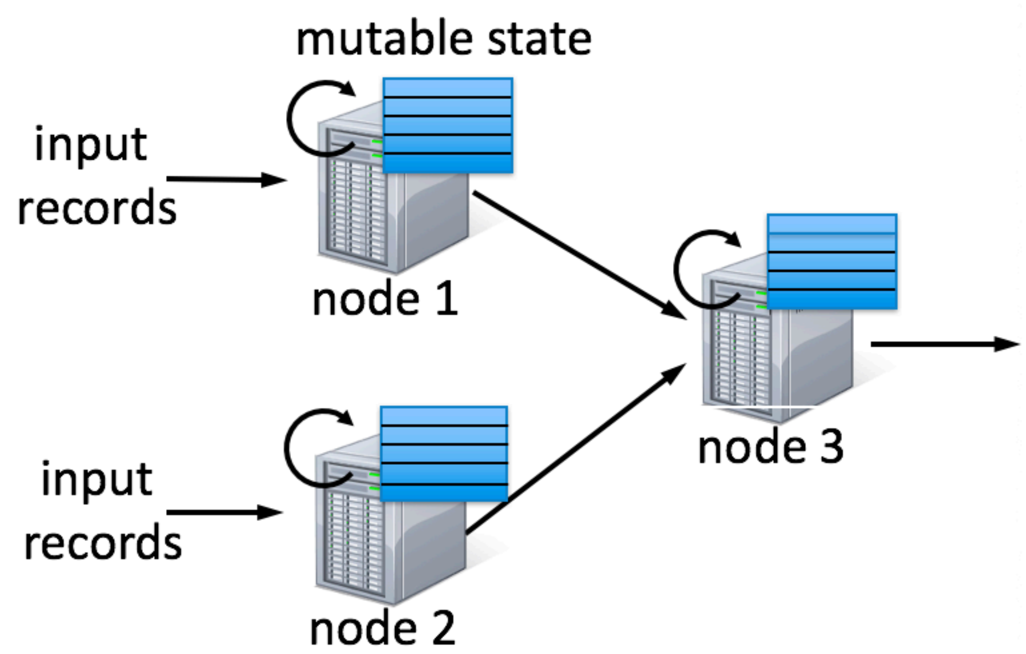
Efficient fault-tolerance in stateful computations

Stateful stream processing

Traditional streaming systems have an event-driven record-at-a-time processing model:

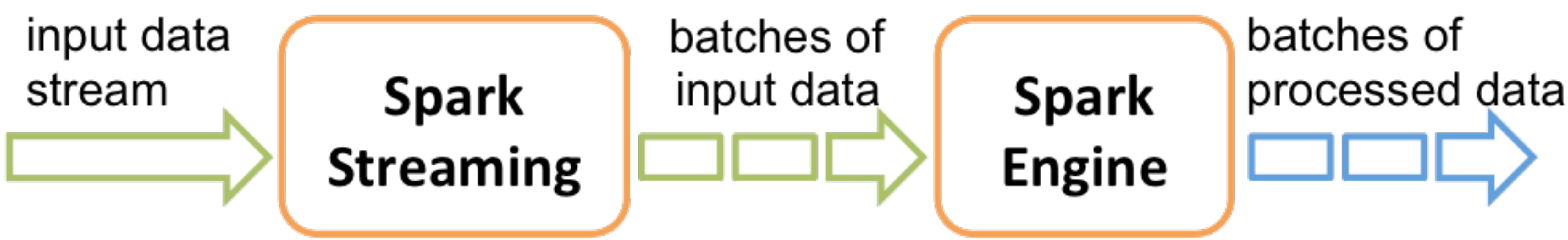
- each node has mutable state
- for each records, update state and send new records

State is lost if node dies



What is Spark Streaming

Extension of core Spark API that makes it easy to build fault-tolerant processing of real-time data streams



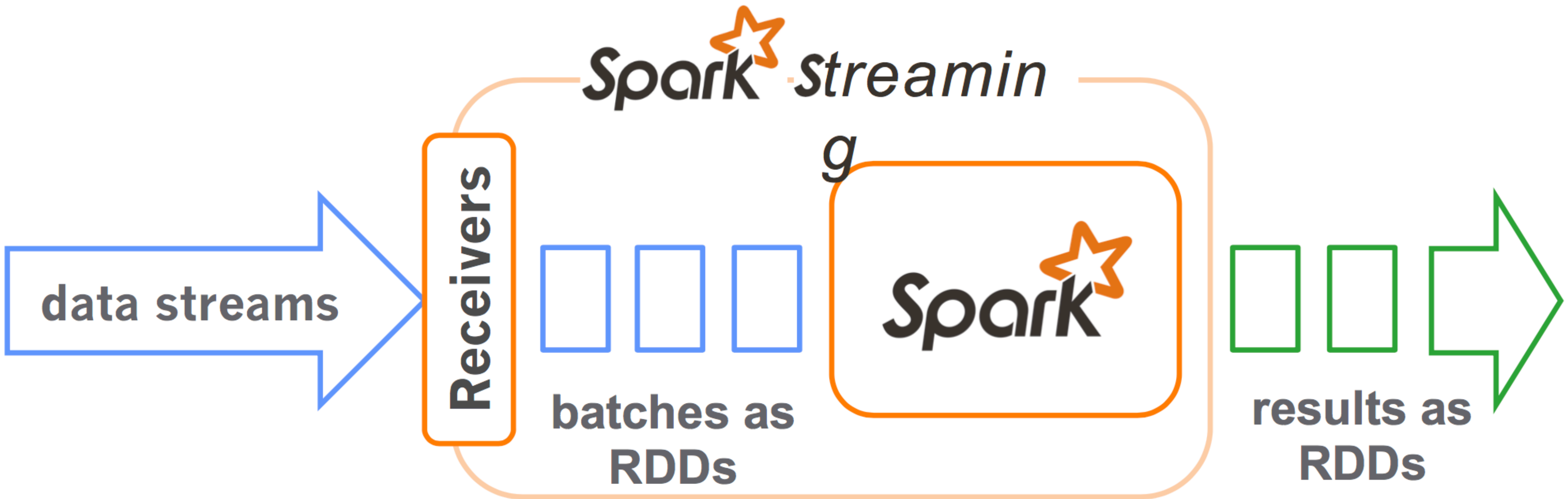
What is Spark Streaming?

- Receive data streams from input sources, process them in a cluster, push out to databases/ dashboards
- Scalable, fault-tolerant, second-scale latencies

DATABRICKS

How does Spark Streaming work?

- Chop up data streams into batches of few secs
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Processed results are pushed out in batches



Spark Streaming Programming Model

> *Discretized Stream (DStream)*
  - Represents a stream of data
  - Implemented as a sequence of RDDs

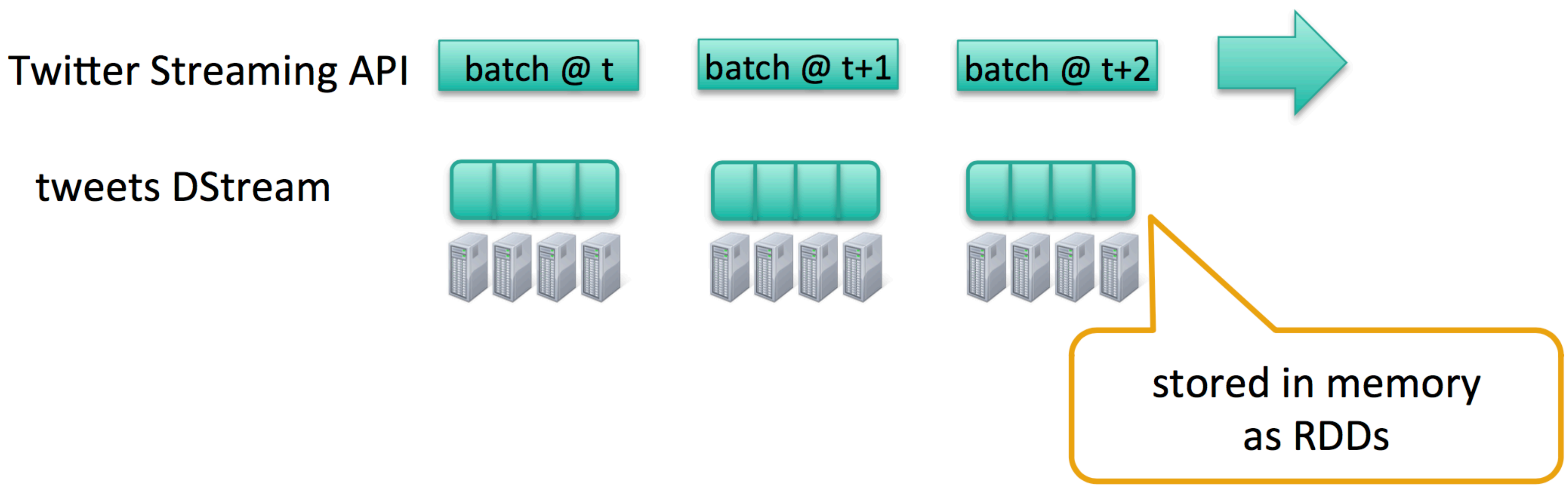
> DStreams API very similar to RDD API
  - Functional APIs in Scala, Java
  - Create input DStreams from different sources
  - Apply parallel operations

DATABRICKS

Example – Get hashtags from Twitter

```
val ssc = new StreamingContext(sparkContext, Seconds(1))
val tweets = TwitterUtils.createStream(ssc, auth)
```

Input DStream



DATABRICKS

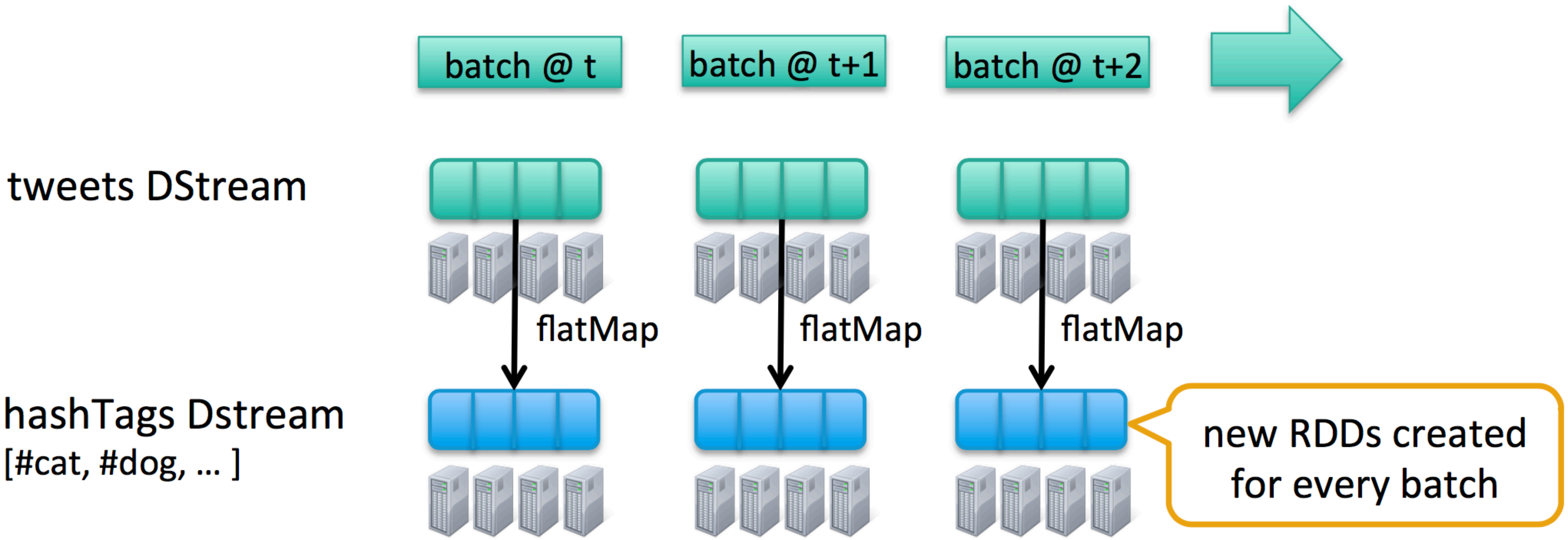
DStream Operations

Example – Get hashtags from Twitter

```
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
```

transformed DStream

transformation: modify data in one DStream to create another DStream

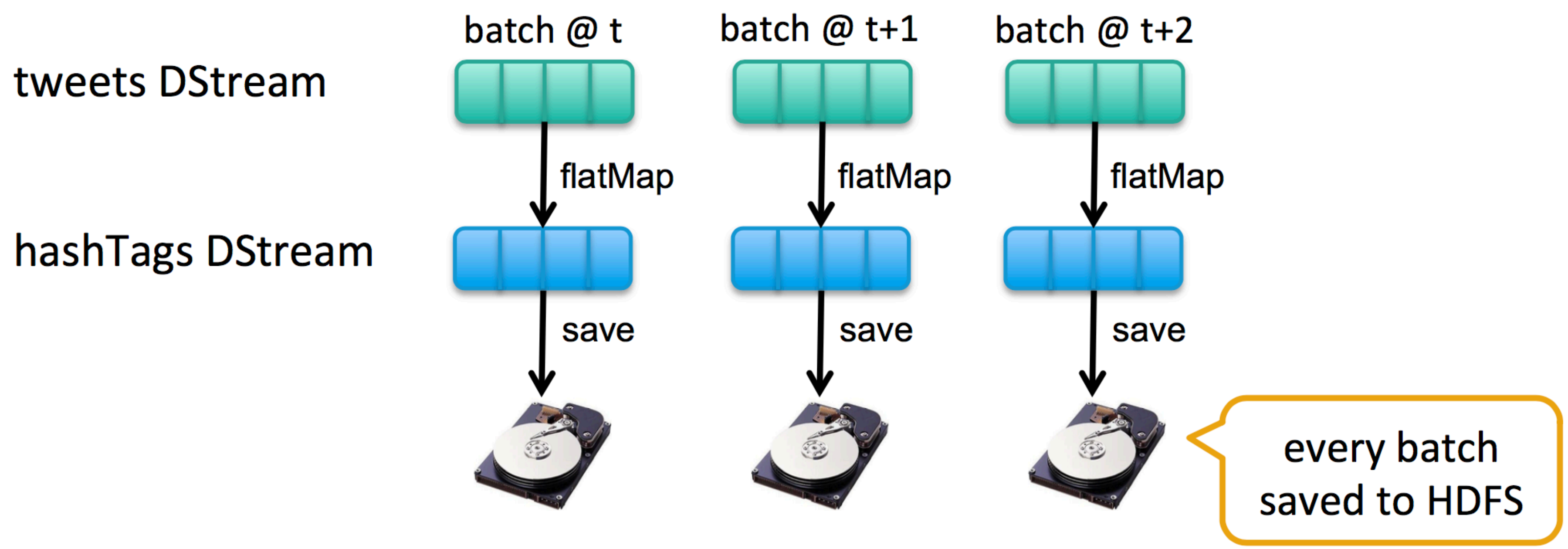


DATABRICKS

Example – Get hashtags from Twitter

```
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

output operation: to push data to external storage

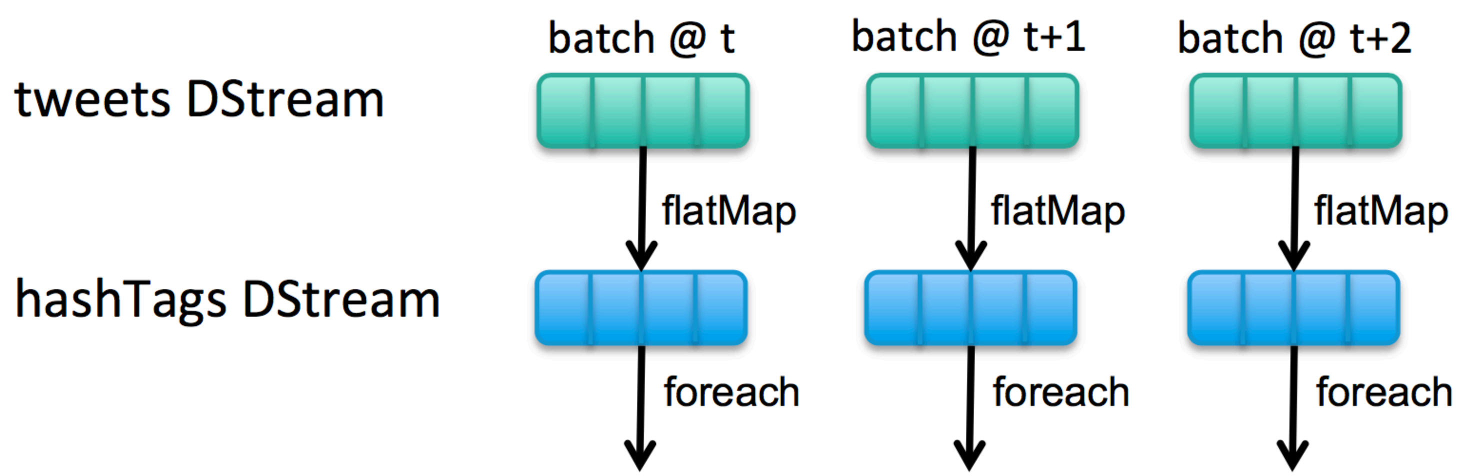


DATABRICKS

Example – Get hashtags from Twitter

```
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.foreachRDD(hashTagRDD => { ... })
```

foreach: do whatever you want with the processed data



Write to a database, update analytics UI, do whatever you want

DATABRICKS

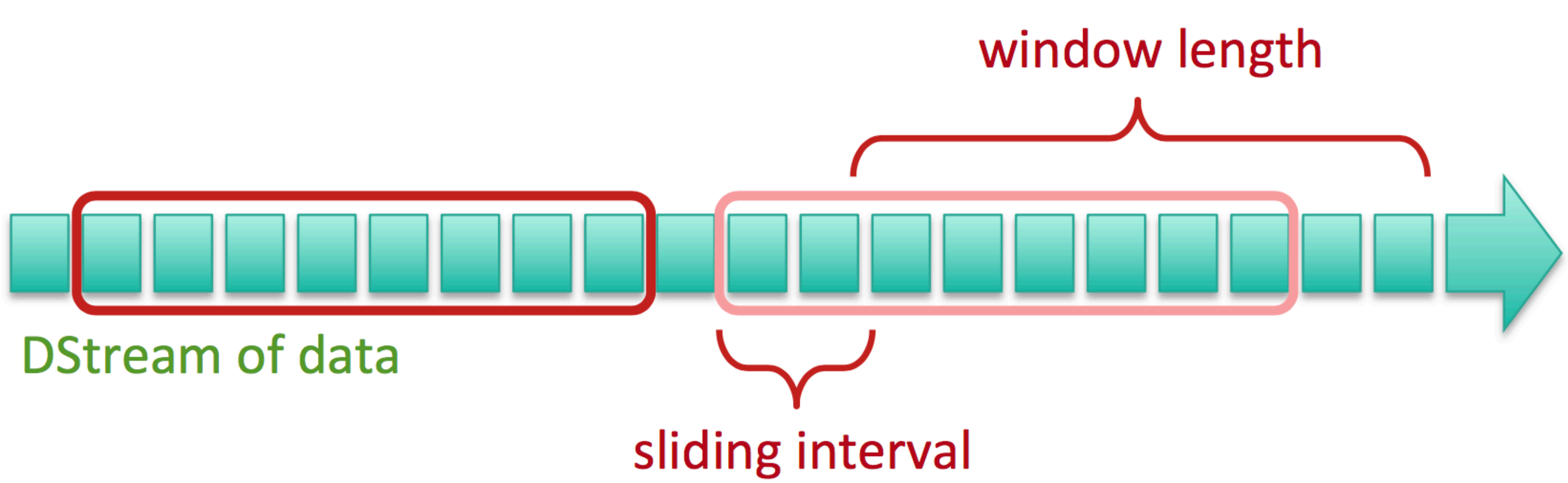
Window-based Transformations

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

sliding window operation

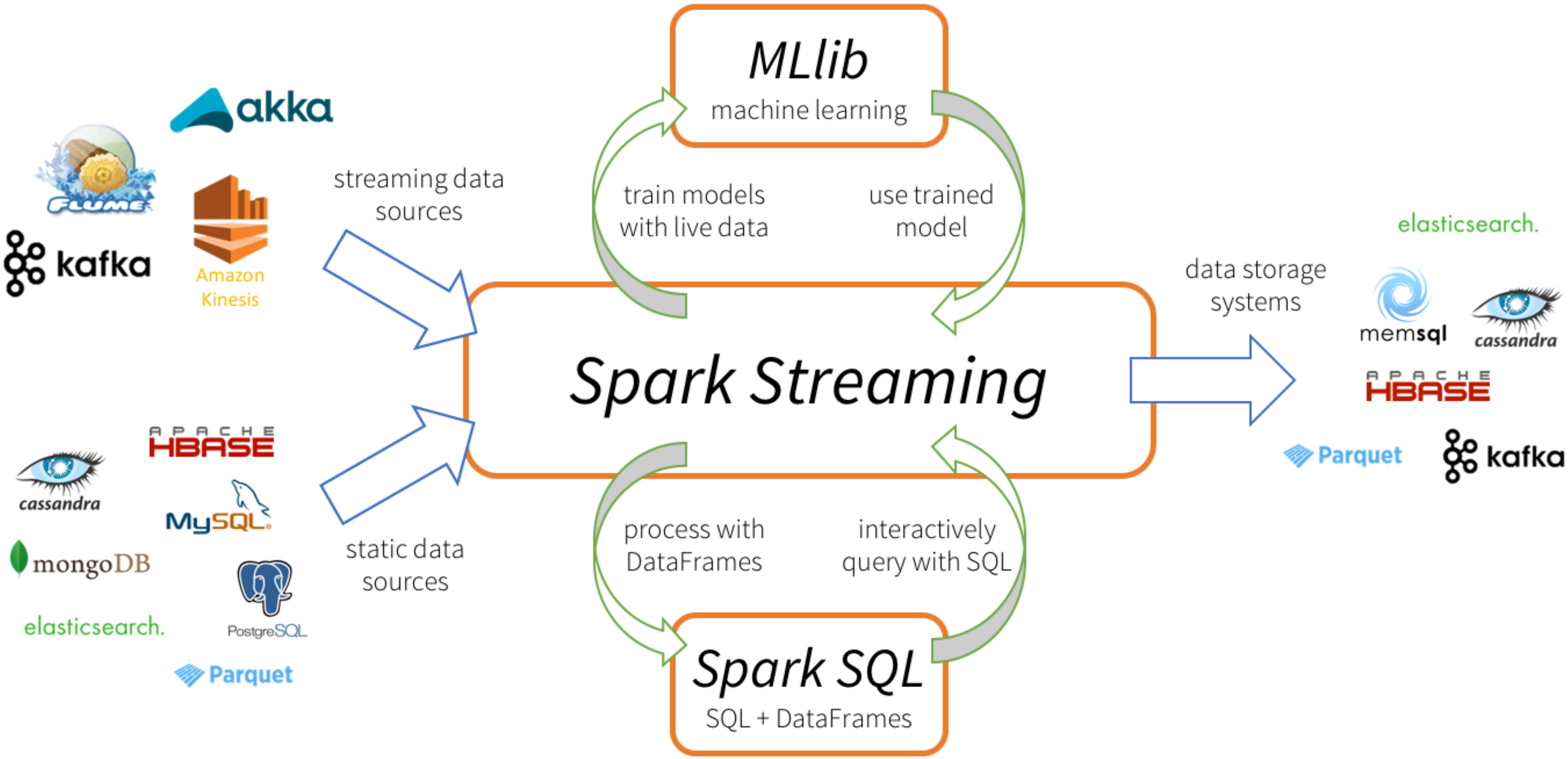
window length

sliding interval



DATABRICKS

Spark Streaming in the Spark Ecosystem



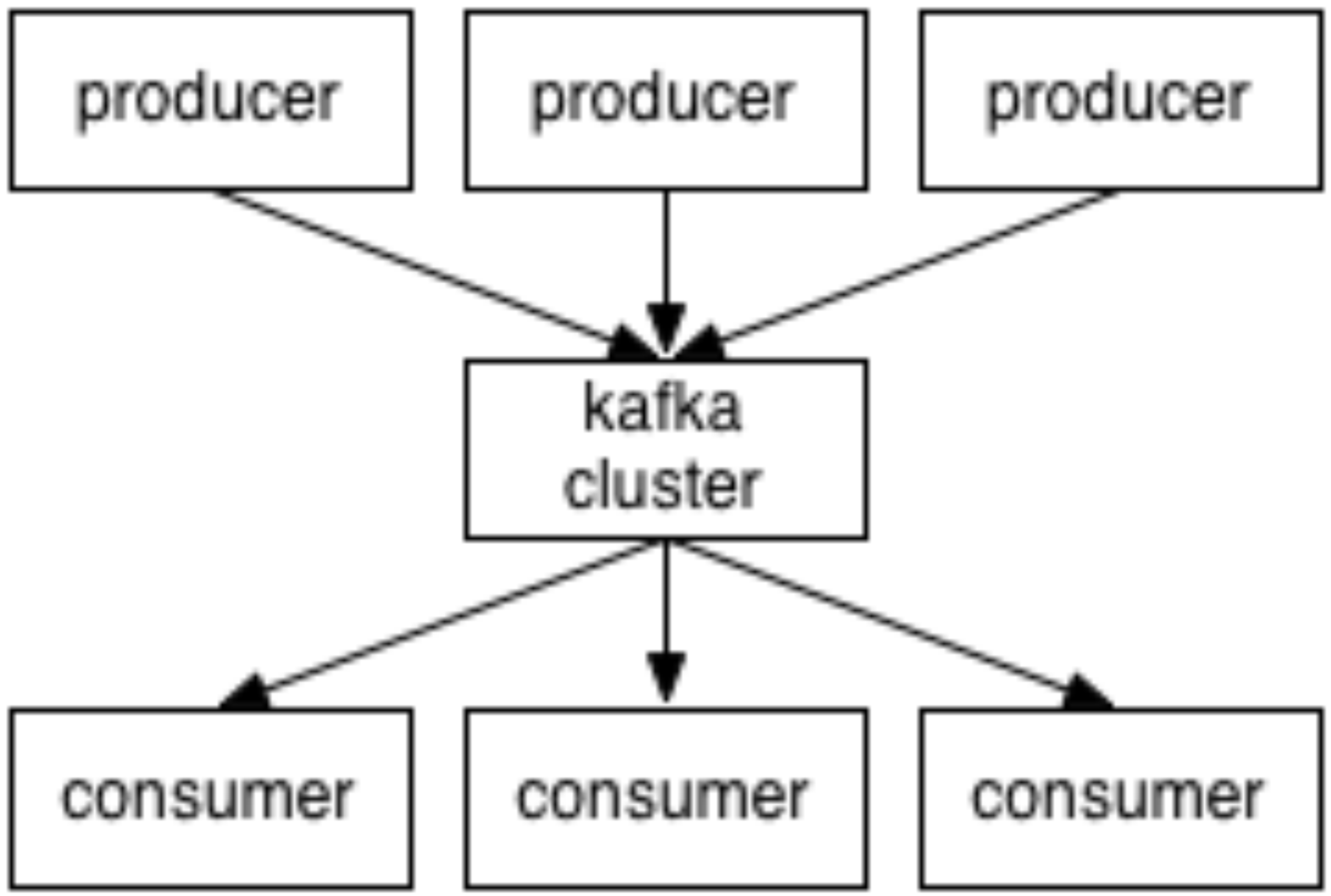
More Spark Streaming operations

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

Streaming data generation demo

We will see a demo of a Kafka producer and a Kafka consumer in real time.

- Kafka producer: a producer of data stream
- Kafka consumer: a consumer of data stream



Spark streaming jobs are continuous

A spark streaming job, once submitted to YARN should run forever until it is intentionally stopped.

You cannot run a spark streaming interactively (spark-shell, Jupyter, etc.). It has to be done through spark-submit.

Lab: Run the Hello World of Spark Streaming

Open one terminal window and connect to your cluster

Clone the class repository

Change to L11

WordCount Example

Explore the code in L11/streaming-netword-wordcount.scala and blog-replay.scala

Open another terminal window connection to cluster (two connections)

- one will run netcat
- one will run the spark streaming job

In one terminal type:

```
/usr/lib/spark/bin/run-example --master local streaming.NetworkWordCount localhost 9999
```

In second terminal type:

```
nc -lk 9999
```

Note: I could not get this to work quite as it should on EMR. The wordcount example does seem to work, but you cannot see the output until you quit the streaming job. The web log example spark streaming job seems to run but is not generating the output table as explained in the blog post https://aws.amazon.com/blogs/big-data/real-time-stream-processing-using-apache-spark-streaming-and-apache-kafka-on-aws/

GraphX

Graphs

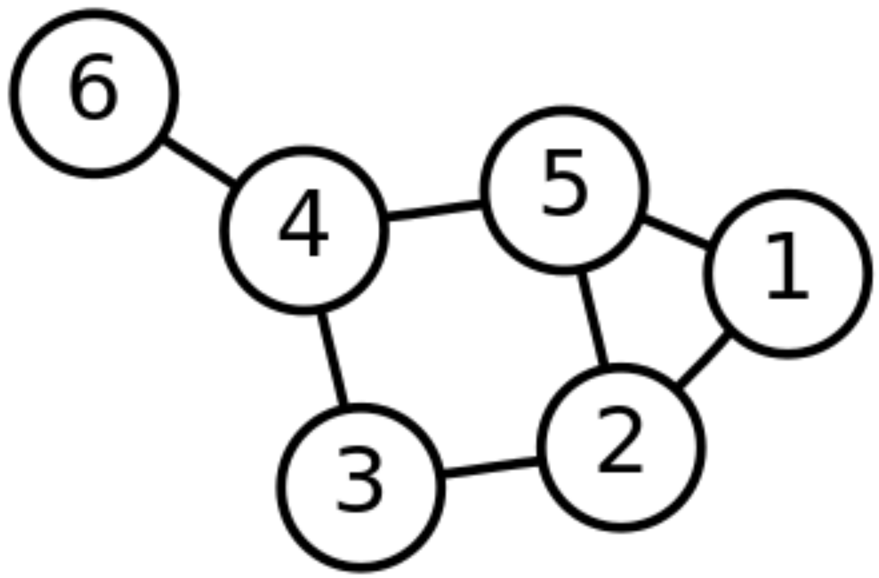
Collections of nodes/vertices (used interchangeably) and edges

Nodes represent actors/entities

- Nodes can have attributes

Edges represent relationships

- Directed/undirected
- Edges can have attributes



Many, many things are graphs

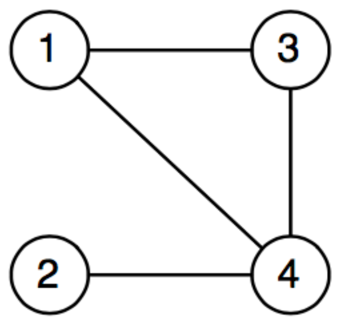
- Networks
- Relationships

Graph use cases

- Recommendations and personalization
- Fraud detection
- Topic modeling
- Community detection
- Shortest Distance

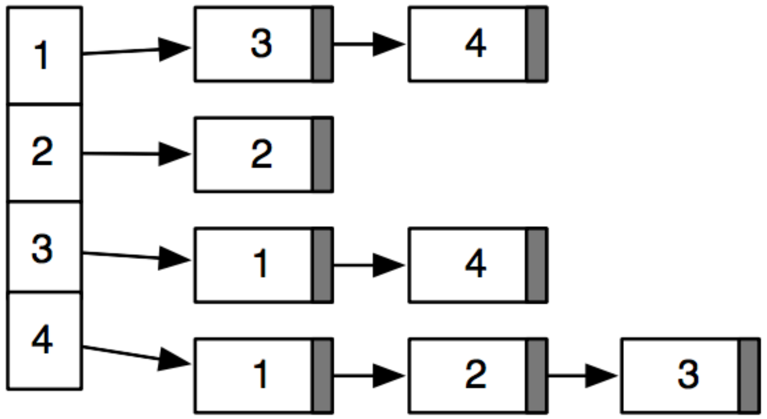
Representing a graph

Adjacency matrix. An $n \times n$ matrix of binary values in which location (i, j) is 1 if $(i, j) \in E$ and 0 otherwise. Note that for an undirected graph the matrix is symmetric and 0 along the diagonal.

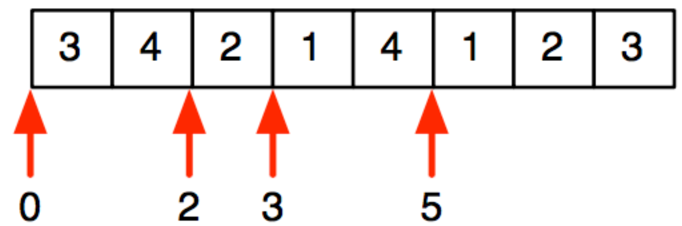


$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Adjacency list. An array A of length n where each entry $A[i]$ contains a pointer to a linked list of all the out-neighbors of vertex i. In an undirected graph with edge $\{u, v\}$ the edge will appear in the adjacency list for both u and v.



Adjacency array. Similar to an adjacency list, an adjacency array keeps the neighbors of all vertices, one after another, in an array `adj`; and separately, keeps an array of indices that tell us where in the `adj` array to look for the neighbors of each vertex.



Edge list. A list of pairs $(i, j) \in E$.

What is GraphX

GraphX is Apache Spark's API for graphs and graph-parallel computation. It extends the Spark RDD by introducing a new Graph abstraction: a **directed multigraph** with properties attached to each vertex and edge.

Operator Type	Operators	Description
Basic Operators	• numEdges • numVertices • inDegrees • outDegrees • degrees	
Property Operators	• mapVertices • mapEdges • mapTriplets	
Structural Operators	• reverse • subgraph • mask • groupEdges	
Join Operators	• joinVertices • outerJoinVertices	

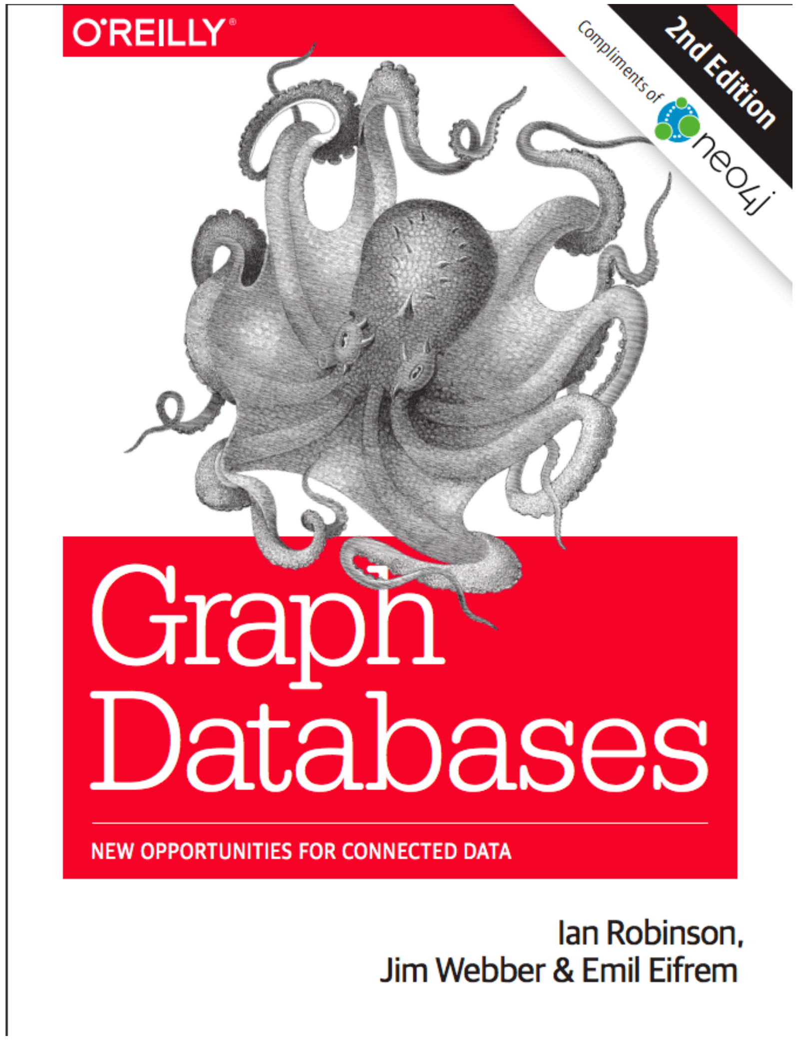
GraphFrames is a graph processing library for based on DataFrames.

Other Graph Analysis Tools

Neo4j

Titan

DataStax



Why GraphX?

GraphX combines advantages of both data-parallel and graph-parallel by efficiently expressing graph computations within the Spark data-parallel framework.

- Graph-parallel: An abstraction that compactly describes graph algorithms and a corresponding run- time engine that efficiently executes these algorithms on multicore and distributed systems.
- Data-parallel: scalable data processing

The goal of the GraphX project is to unify graph-parallel and data-parallel computation in one system with a single composable API. The GraphX API enables users to view data both as graphs and as collections (i.e., RDDs) without data movement or duplication. By incorporating recent advances in graph-parallel systems, GraphX is able to optimize the execution of graph operations.

GraphX: A Resilient Distributed Graph System on Spark

Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, Ion Stoica

AMPLab, EECS, UC Berkeley
{rxin, jegonzal, franklin, istoica}@cs.berkeley.edu

ABSTRACT

From social networks to targeted advertising, big graphs capture the structure in data and are central to recent advances in machine learning and data mining. Unfortunately, directly applying existing data-parallel tools to graph computation tasks can be cumbersome and inefficient. The need for intuitive, scalable tools for graph computation has lead to the development of new *graph-parallel* systems (*e.g.*, Pregel, PowerGraph) which are designed to efficiently execute graph algorithms. Unfortunately, these new graph-parallel systems do not address the challenges of graph construction and transformation which are often just as problematic as the subsequent computation. Furthermore, existing graph-parallel systems provide limited fault-tolerance and support for interactive data mining.

We introduce GraphX, which combines the advantages of both data-parallel and graph-parallel systems by efficiently expressing graph computation within the Spark data-parallel framework. We leverage new ideas in distributed graph representation to efficiently distribute graphs as tabular data-structures. Similarly, we leverage advances in data-flow systems to exploit in-memory computation and fault-tolerance. We provide powerful new operations to simplify graph construction and transformation. Using these primitives we implement the PowerGraph and Pregel abstractions in less than 20 lines of code. Finally, by exploiting the Scala foundation of Spark, we enable users to interactively load, transform, and compute on massive graphs.

1. INTRODUCTION

From social networks to advertising and the web, big graphs can be found in a wide range of important applications. By modeling the relationships between users, products, and ideas, graphs allow us to identify communities, target advertising, and decipher the meaning of documents. In response to the growing size and importance of graph data, a range of new large-scale distributed *graph-parallel* frameworks (*e.g.*, Pregel [9], PowerGraph [6], and others [5, 3, 11]) have emerged. Each framework introduces a new programming abstraction that allows users to compactly describe graph algorithms (*e.g.*, PageRank, Belief Propagation, ...) and a corresponding run-time engine that efficiently executes these algorithms on multicore and distributed systems. By abstracting away the challenges of large-scale distributed system design, these frameworks simplify the design, implementation, and application of new sophisticated graph algorithms to large-scale real-world graph problems.

While existing graph-parallel frameworks share many common properties, each presents a slightly different view of graph computation tailored to either the originating domain or a specific family of graph algorithms and applications. Unfortunately, because each framework relies on a separate runtime, it is difficult to compose these abstractions. Furthermore, while these frameworks address the challenges of graph computation, they do not address the challenges of data ETL (preprocessing and construction) or the process of interpreting and applying the results of computation. Finally, few frameworks have built-in support for interactive graph computation.

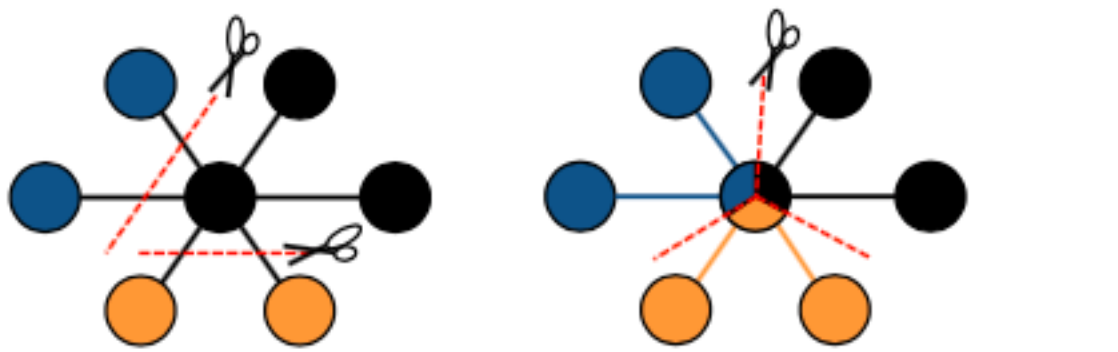
Alternatively *data-parallel* systems like MapReduce and Spark [12] are designed for scalable data processing and are well suited to the task of graph construction (ETL). By exploiting data-parallelism, these systems are highly scalable and support a range of fault-tolerance strategies. More recent systems like Spark even enable interactive data processing. However, naively expressing graph computation and graph algorithms in these data-parallel abstractions can be challenging and typically leads to complex joins and excessive data movement that does not exploit the graph structure.

To address these challenges we introduce GraphX, a graph computation system which runs in the Spark data-parallel framework. GraphX extends Spark's Resilient Distributed Dataset (RDD) abstraction to introduce the Resilient Distributed Graph (RDG), which associates records with vertices and edges in a graph and provides a collection of expressive computational primitives. Using these primitives, we implement PowerGraph [6] and Pregel [9], two of the most widely used graph-processing frameworks. In addition, we provide new operations to view, filter, and transform graphs, that substantially simplify the process of graph ETL and analysis. The GraphX RDG leverages advances in distributed graph representation and exploits the graph structure to minimize communication and storage overhead. Our primary contributions are:

1. a new graph abstraction called the Resilient Distributed Graph (RDG) that supports a wide range of graph operations on top of a fault-tolerant, interactive platform.
2. a tabular representation of the efficient vertex-cut partitioning described by [6] and data-parallel partitioning heuristics.
3. implementations of the PowerGraph and Pregel graph-parallel frameworks using RDGs in less than 20 lines of code each.
4. preliminary performance comparisons between a popular data-parallel and graph-parallel frameworks running PageRank on a large real-world graph.

Proceedings of the First International Workshop on Graph Data Management Experience and Systems (GRADES 2013), June 23, 2013, New York, New York, USA.

GraphX partitioning



(a) Edge-Cut (b) Vertex-Cut

Figure 2: **Edge-Cut vs Vertex-Cut:** An edge-cut **(a)** splits the graph along edges while a vertex-cut **(b)** splits the graph along vertices. In this illustration we partition the graph across three machines (corresponding to color).

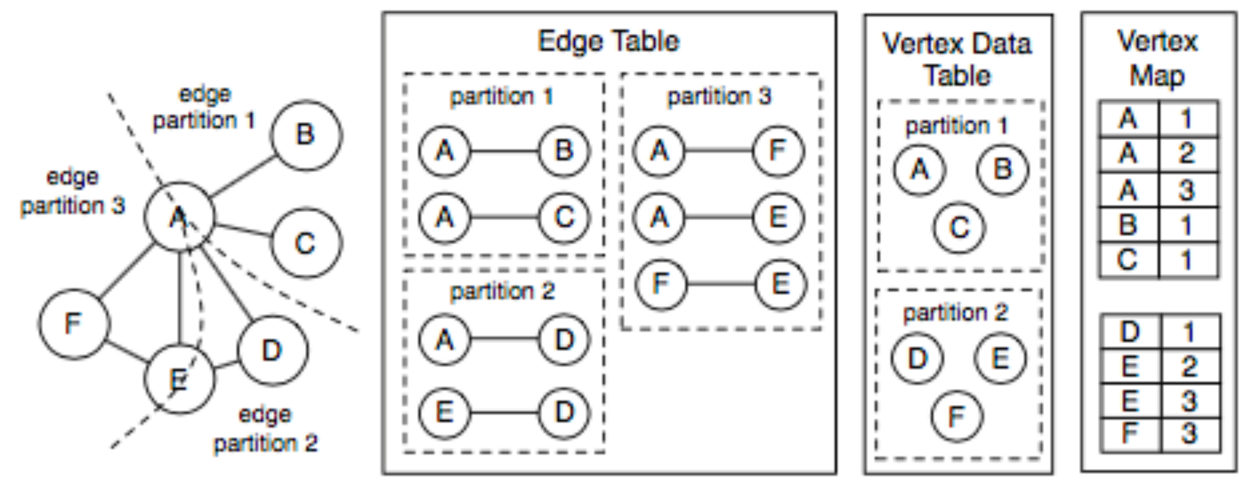
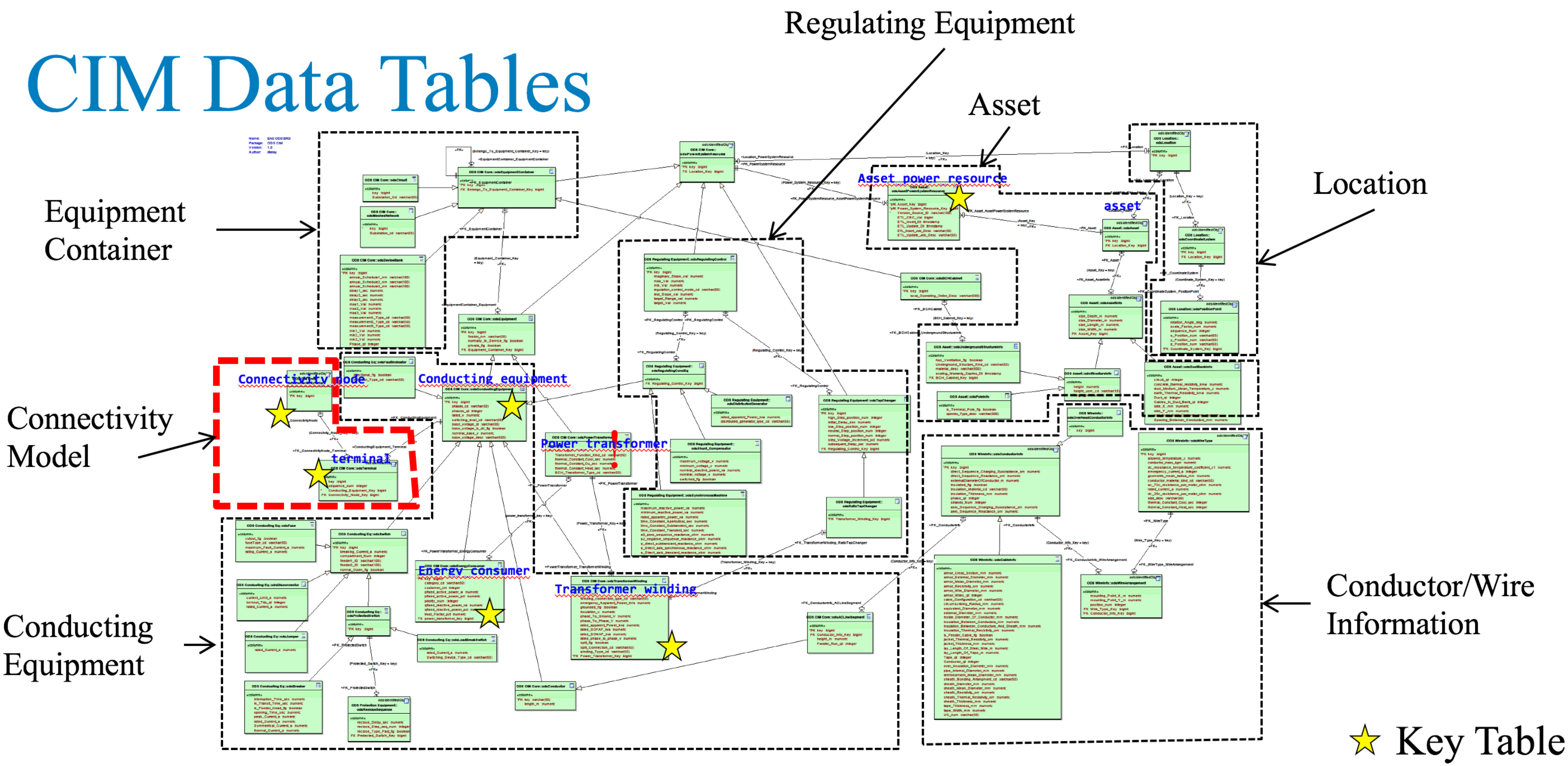


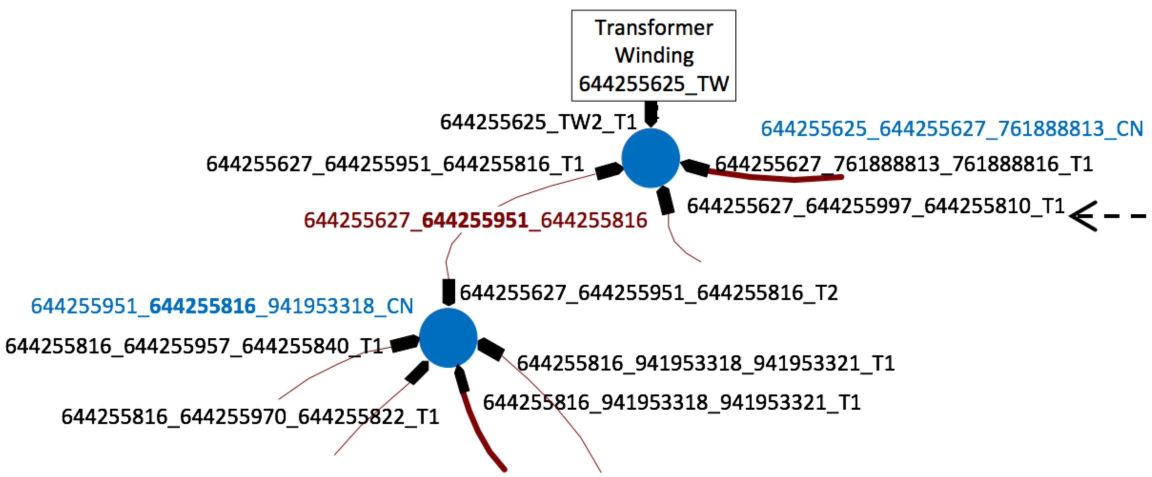
Figure 3: **GraphX Tabular Representation of a Vertex-Cut:** Here we partition the graph on the left across three virtual partitions using a vertex-cut. The edge table contains the edge data as well as the vertex ids for each edge and is partitioned by the virtual `pid` field associated with each record. The vertex table contains the vertex id and vertex data and is partitioned (keyed) by the vertex `id`. Finally, the vertex map contains tuples of `(vid,pid)` and encodes the mapping from vertex `id` to the edge table partitions which contain adjacent edges. The vertex map table is also partitioned and keyed by the vertex id.

Example: making hierarchical data fit a graph model

CIM: Common Information Model (xml)



Topology example (underground transformer)



Zoomed-In Connectivity Node

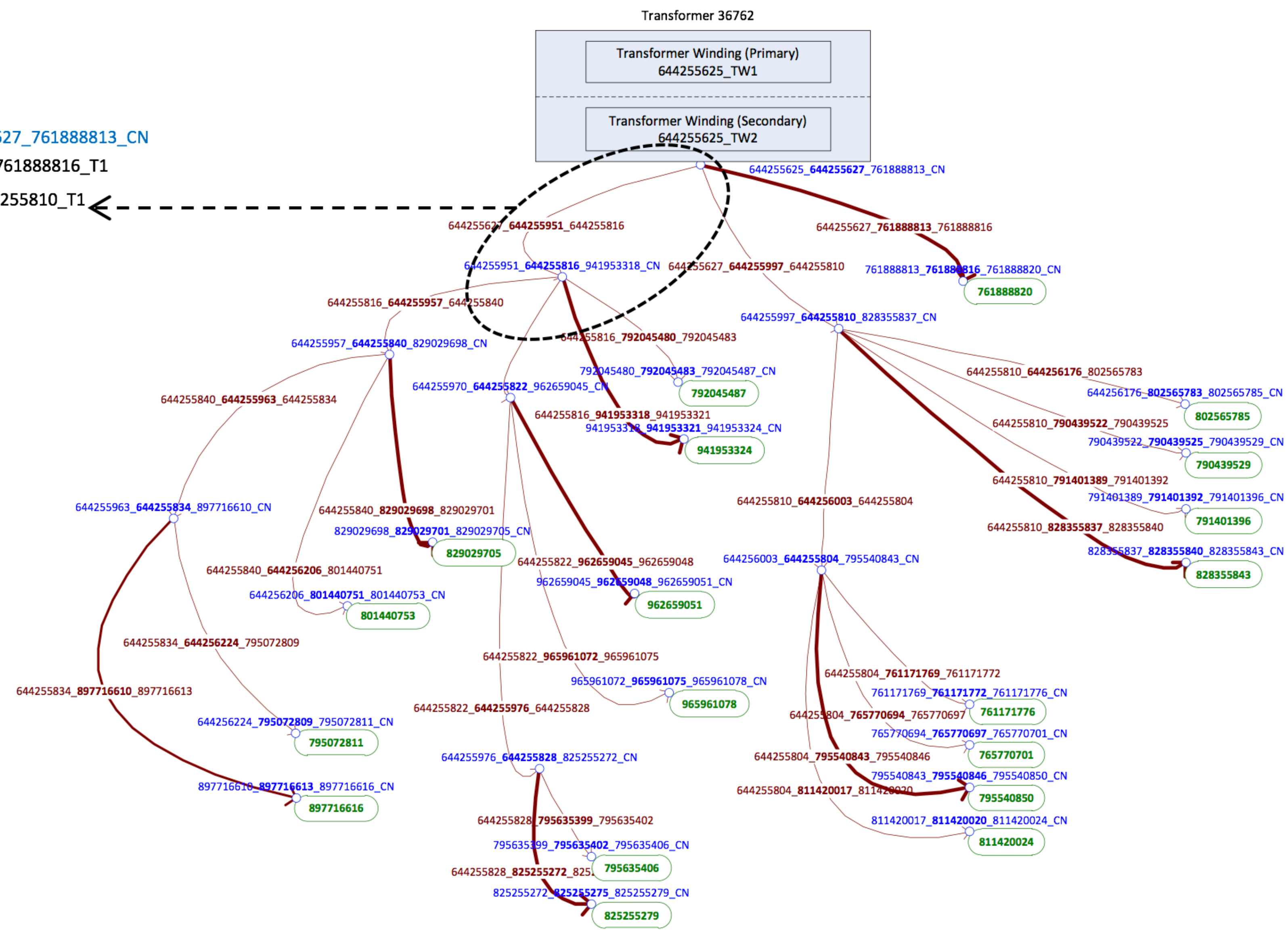
Transformer

Connectivity Node

A/C Line Segment

SLID (from CIM)

Terminal



Raw extraction

conducting_equipment_key	connectivity_node_key	conducting_equipment_mrid	conducting_equipment_type	equipment_container_key	terminal_mrid	connectivity_node_mrid	level
180098280	-1	644255625_TW2	NA	-1	644255625_TW2_T1	NA	0
180098280	192467809	644255625_TW2	NA	-1	644255625_TW2_T1	644255625_644255627_761888813_CN	0
215062305	192467809	644255627_644255951_644255816	U/G Secondary	167882914	644255627_644255951_644255816_T1	644255625_644255627_761888813_CN	1
215062306	192467809	644255627_644255997_644255810	U/G Secondary	167882914	644255627_644255997_644255810_T1	644255625_644255627_761888813_CN	1
215062307	192467809	644255627_761888813_761888816	U/G Secondary	167882914	644255627_761888813_761888816_T1	644255625_644255627_761888813_CN	1
215062305	192467814	644255627_644255951_644255816	U/G Secondary	167882914	644255627_644255951_644255816_T2	644255951_644255816_941953318_CN	1
215062348	192467814	644255816_644255957_644255840	U/G Secondary	167882914	644255816_644255957_644255840_T1	644255951_644255816_941953318_CN	2
215062349	192467814	644255816_644255970_644255822	U/G Secondary	167882914	644255816_644255970_644255822_T1	644255951_644255816_941953318_CN	2
215062350	192467814	644255816_792045480_792045483	U/G Secondary	167882914	644255816_792045480_792045483_T1	644255951_644255816_941953318_CN	2
215062351	192467814	644255816_941953318_941953321	U/G Secondary	167882914	644255816_941953318_941953321_T1	644255951_644255816_941953318_CN	2
215062348	192467815	644255816_644255957_644255840	U/G Secondary	167882914	644255816_644255957_644255840_T2	644255957_644255840_829029698_CN	2
215062359	192467815	644255840_644255963_644255834	U/G Secondary	167882914	644255840_644255963_644255834_T1	644255957_644255840_829029698_CN	3
215062360	192467815	644255840_644256206_801440751	U/G Secondary	167882914	644255840_644256206_801440751_T1	644255957_644255840_829029698_CN	3
215062361	192467815	644255840_829029698_829029701	U/G Secondary	167882914	644255840_829029698_829029701_T1	644255957_644255840_829029698_CN	3
215062357	192467816	644255834_644256224_795072809	U/G Secondary	167882914	644255834_644256224_795072809_T1	644255963_644255834_897716610_CN	4
215062358	192467816	644255834_897716610_897716613	U/G Secondary	167882914	644255834_897716610_897716613_T1	644255963_644255834_897716610_CN	4
215062359	192467816	644255840_644255963_644255834	U/G Secondary	167882914	644255840_644255963_644255834_T2	644255963_644255834_897716610_CN	3
215062349	192467817	644255816_644255970_644255822	U/G Secondary	167882914	644255816_644255970_644255822_T2	644255970_644255822_962659045_CN	2
215062352	192467817	644255822_644255976_644255828	U/G Secondary	167882914	644255822_644255976_644255828_T1	644255970_644255822_962659045_CN	3
215062353	192467817	644255822_962659045_962659048	U/G Secondary	167882914	644255822_962659045_962659048_T1	644255970_644255822_962659045_CN	3
215062354	192467817	644255822_965961072_965961075	U/G Secondary	167882914	644255822_965961072_965961075_T1	644255970_644255822_962659045_CN	3
215062352	192467818	644255822_644255976_644255828	U/G Secondary	167882914	644255822_644255976_644255828_T2	644255976_644255828_825255272_CN	3
215062355	192467818	644255828_795635399_795635402	U/G Secondary	167882914	644255828_795635399_795635402_T1	644255976_644255828_825255272_CN	4
215062356	192467818	644255828_825255272_825255275	U/G Secondary	167882914	644255828_825255272_825255275_T1	644255976_644255828_825255272_CN	4

- This example has a tree that is four levels deep beginning from secondary transformer. This required a total of four complete cycles of **equipment -> terminal -> node -> terminal -> equipment**. This process was done in R.
- The level of depth of a tree (from feeder to primary transformer, or from secondary transformer to SLID) is not know before the tree is traversed.

Parent-Child relationship

Parent | Child

transformer_mrid	parent_conducting_equipment_mrid	parent_class	parent_conducting_equipment_type_cd	parent_normal_open_flg	child_conducting_equipment_mrid	child_class
644255625	644255625_TW2	transformer_winding	NA	NA	644255627_644255951_644255816	ac_line_segment
644255625	644255625_TW2	transformer_winding	NA	NA	644255627_644255997_644255810	ac_line_segment
644255625	644255625_TW2	transformer_winding	NA	NA	644255627_761888813_761888816	ac_line_segment

SQL-friendly, standard parent/child relationship, but there is no semantic interpretation: an ac line is also node. In addition, it is still difficult to identify special properties, such as Point of Coupling.

transformer_mrid	parent_conducting_equipment_mrid	parent_class	parent_conducting_equipment_type_cd	parent_normal_open_flg	child_conducting_equipment_mrid	child_class	child_conducting_equipment_type_cd	child_normal_open_flg	vertex_flg	[illegible]	[illegible]	parent_vertex_child_edge	parent_edge_child_vertex	[illegible]
[illegible]	[illegible]	transformer_winding	NA	NA	[illegible]	ac_line_segment	[illegible]	NA	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE
[illegible]	[illegible]	transformer_winding	NA	NA	[illegible]	ac_line_segment	[illegible]	NA	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE
[illegible]	[illegible]	transformer_winding	NA	NA	[illegible]	ac_line_segment	[illegible]	NA	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE
[illegible]	[illegible]	ac_line_segment	[illegible]	NA	[illegible]	ac_line_segment	[illegible]	NA	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
[illegible]	[illegible]	ac_line_segment	[illegible]	NA	[illegible]	ac_line_segment	[illegible]	NA	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
[illegible]	[illegible]	ac_line_segment	[illegible]	NA	[illegible]	ac_line_segment	[illegible]	NA	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE

Edge list

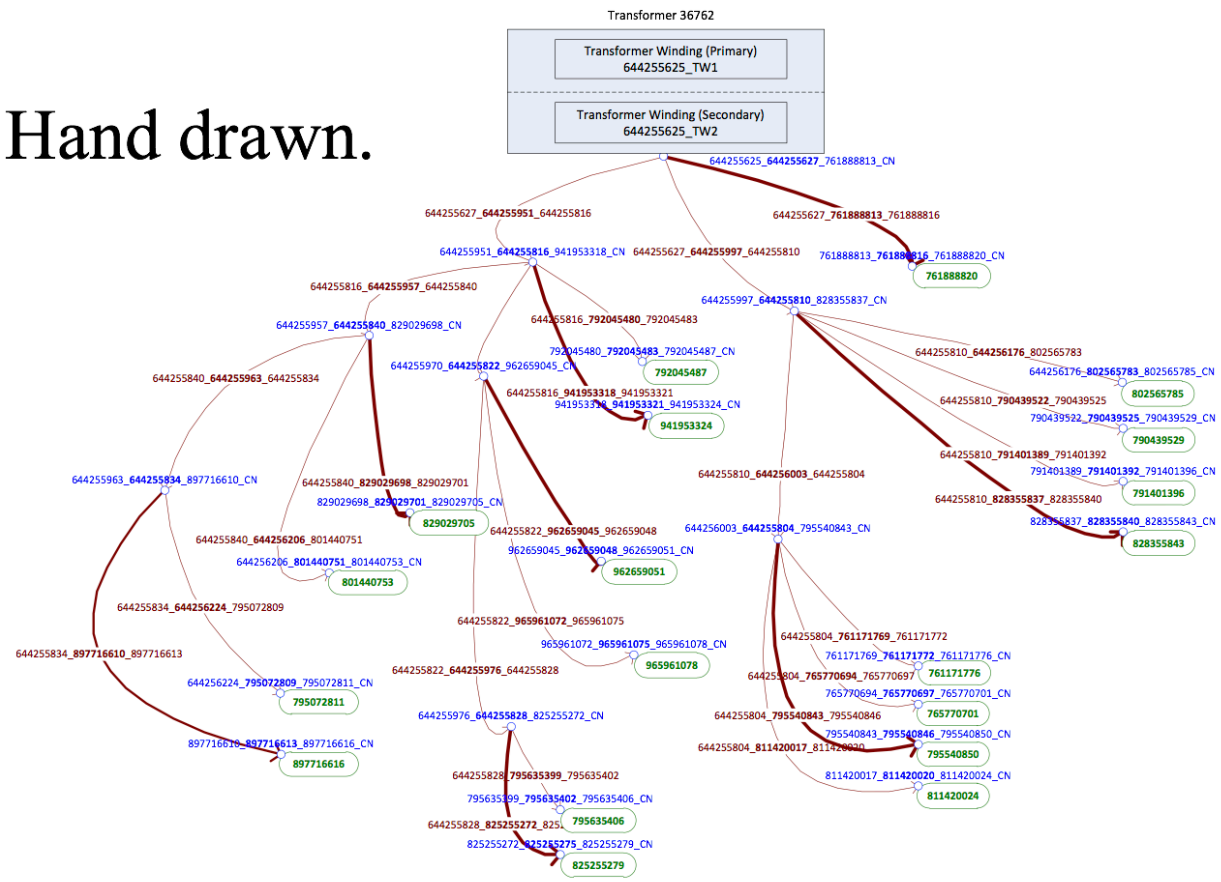
Edge | Node | Node

transformer_mrid	edge_mrid	order	from	from_class	from_type_cd	to	to_class	to_type_cd
644255625	644255627_644255951_644255816	1	644255625_TW2	transformer_winding	NA	ccn_2	common_connectivity_node	NA
644255625	644255627_644255997_644255810	2	644255625_TW2	transformer_winding	NA	ccn_1	common_connectivity_node	NA
644255625	644255627_761888813_761888816	3	644255625_TW2	transformer_winding	NA	761888820	energy_consumer	Service Location
644255625	644255810_644256003_644255804	4	ccn_1	common_connectivity_node	NA	ccn_3	common_connectivity_node	NA
644255625	644255810_644256176_802565783	5	ccn_1	common_connectivity_node	NA	802565785	energy_consumer	Service Location
644255625	644255810_790439522_790439525	6	ccn_1	common_connectivity_node	NA	790439529	energy_consumer	Service Location
644255625	644255810_791401389_791401392	7	ccn_1	common_connectivity_node	NA	791401396	energy_consumer	Service Location

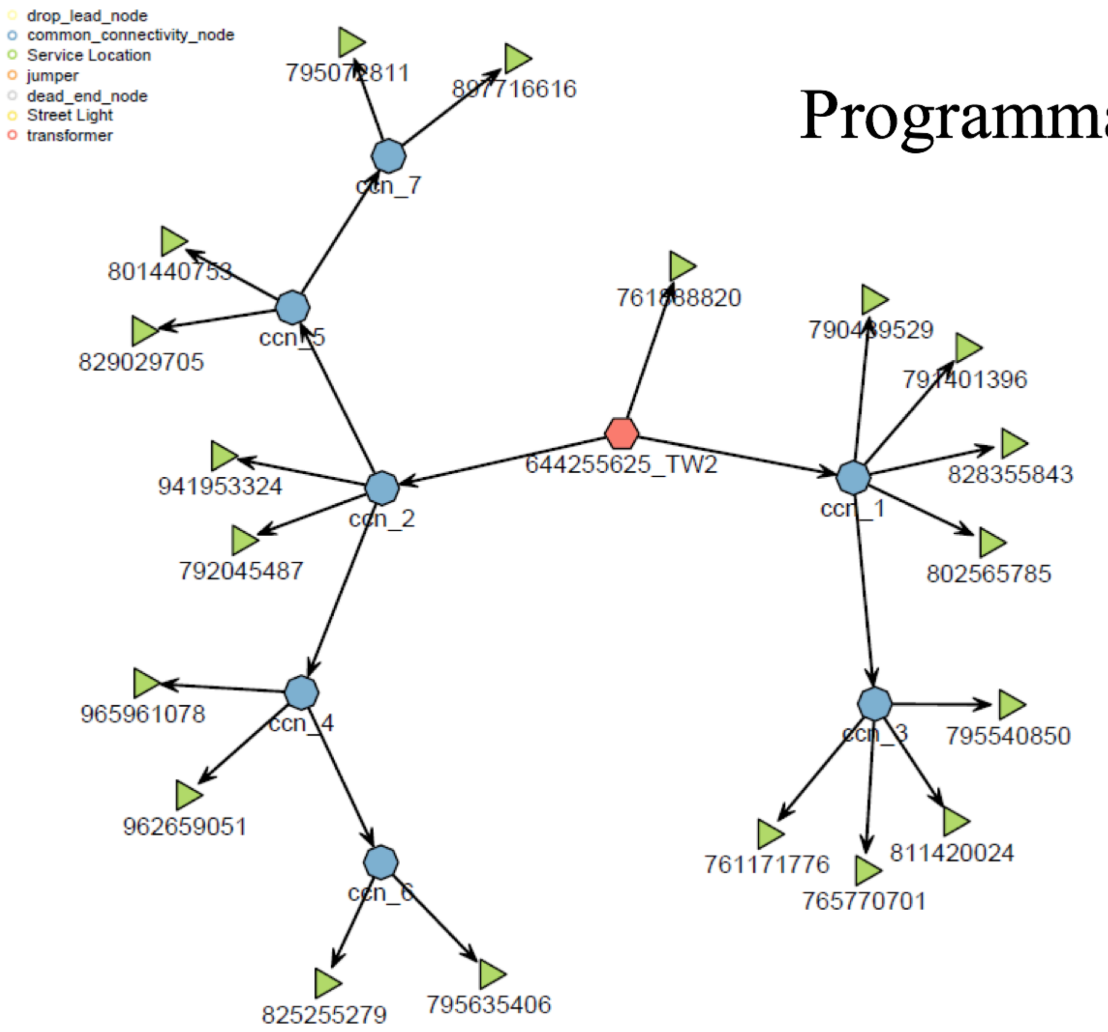
Edges are wires and lines whereas nodes are the other physical or virtual entities, such as transformers, SLIDs, cuts, street lights, connectivity node, and points of coupling.

Programmatic tracing and visualization

Hand drawn.



Programmatic.



Lab: GraphX Demo

Start a spark-shell or jupyter notebook with Scala kernel

Open the L11/page-rank.scala file. You can run through this line by line. This program runs the PageRank algorithm on YouTube online social network data.

Open the L11/connected-components.scala file. This program finds strongly connected vertices in the LiveJournal social network data.

Open the L11/triangle-count.scala. This program counts the number of triangles (groups of 3 vertices) in the Facebook social circle dataset.

All these programs read in an edge list.