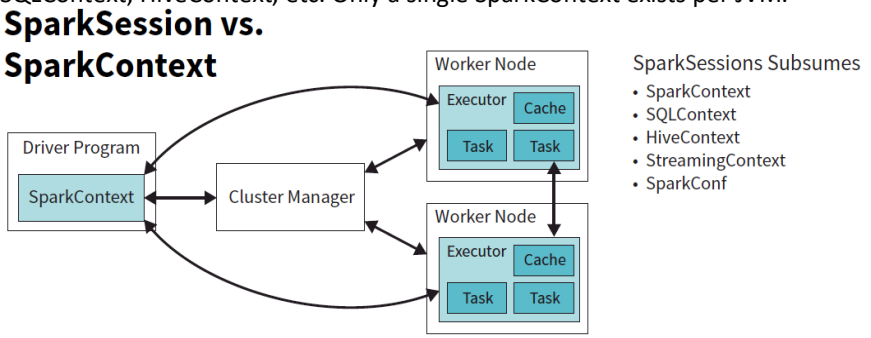
Spark

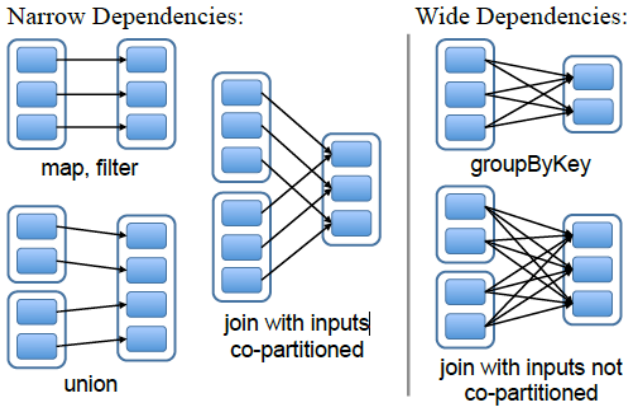
Ming-Hwa Wang, Ph.D.
COEN 242 Introduction to Big Data
Department of Computer Engineering
Santa Clara University

Apache Spark

- Apache Spark is a general-purpose and unified computing engine and a set of libraries for parallel data processing and distributed querying on computer clusters. Spark can be used for interactive queries with sub-second latency. Up to 100 times faster than Apache Hadoop.
- Spark philosophy:
  - Unified: a unified platform (the same computing engine with a consistent and composable set of structured APIs, e.g., DataFrames and Datasets) for writing big data applications to support a wide range of data analytics tasks efficiently via optimization
  - Computing engine: handles loading data from persistent storage systems (cloud storage, DFS, NoSQL, message queues) and performing computation on it, not permanent storage as the end itself. A cluster, or group of machines, pools the resources of many machines together allowing us to use all the cumulative resources as if they were one.
  - Libraries: standard libraries and third-party libraries including interactive or ad-hoc queries with SQL and structured data (Spark SQL), machine learning (MLlib and ML library), graph analytics (GraphX and GraphFrames), and stream processing (Spark Streaming and Structured Streaming).
- Spark is a tool for managing and coordinating the execution of tasks on data across a cluster of computers with standalone cluster manager, YARN, or Mesos.
- Spark is written in Scala, and runs on the Java Virtual Machine (JVM). Run Spark either on your laptop (local mode) or on a cluster (cluster mode).
- Spark's programming model is functional programming where the same inputs always result in the same outputs when the transformations on that data stay constant, and also allow external developer to extend the optimizer.
- Spark architecture: Spark cluster (on-premise or cloud)
  - A Spark master JVM acts as a cluster manager in a standalone deployment mode to which Spark workers register themselves as part of a quorum.
  - A Spark driver program distributes Spark tasks to each worker's executor, and receives computed results from each executor's tasks.
  - Spark workers launch executors
  - Spark executors (JVM containers) execute Spark tasks, store and cache data partitions in their memories
- SparkSession: the single unified entry point for Spark application for reading data, working with metadata, configuration, and cluster resource management. SparkSessions subsume StreamingContext, SparkConf, SparkContext, SQLContext, HiveContext, etc. Only a single SparkContext exists per JVM.



- Spark Applications consist of a single driver process (on the master node) and a set of executor processes (on worker nodes). The driver runs your main() function as a SparkSession and is responsible for a) maintaining information about the application; b) responding to a user's program or input; and 3) creating a DAG for the job, analyzing, distributing, and scheduling work across the executors based on data locality using delay scheduling. The executors are responsible for executing the work (fetching the bytecode before executing with a static set of variables and methods within the executor's context, static means that if the executor changes those variables or overwrites the methods without affecting other executors or the driver) and reporting the state of the computation back to the driver. Spark broadcast can quickly distribute large models to workers.
- Spark parallel operations:
  - Transformation: Spark will not act on transformations until we call an action. Transformation can be either narrow (only one output partition) dependencies/transformations or wide (multiple output partitions and thus need shuffle cross the nodes) dependencies/transformations.
    - map(), filter(), flatmap(), sample(), groupByKey(), reduceByKey(), union(), join(), cogroup(), crossProduct(), mapValues(), sort(), partitionBy(), etc.
    - Lazy evaluation: Spark builds up a plan of transformations, waits until the last minute to execute the code, then compiles this plan from RDD or DataFrame transformations to an efficient physical plan (as a directed acyclic graph DAG with DAGSchedulr) that will run as efficiently (e.g., avoid shuffling) as possible across the cluster. This provides immense benefits to the end user because Spark can optimize the entire data flow from end to end, e.g., "predicate pushdown".



- Actions return values to the driver after running a computation. If you invoke an action, the action will create a job. A job will be decomposed into single or multiple stages; stages are further divided into individual tasks which are executed by the executors. There are 3 kinds of actions: view data in the console, collect data to native object, and write to output data sources.
  - count(), collect(), reduce(), lookup(), save()
- When Spark is transferring data over the network using connection pools (instead of creating dedicated connection), it needs to serialize objects into a binary form. Kryo has a more compact binary representation than the standard Java serializer, and is also faster to compress or decompress.
- Spark's language APIs allow you to run Spark code from other languages including Scala (default language), Java, Python (PySpark), SQL (ANSI 2003), R (core SparkR and community driven package sparklyr). Python and R code will be translated into code that Spark can run on the executor JVMs, thus are slower.
- Starting Spark
  - Interactive mode via spark console (creating a SparkSession automatically)
  - Submit a standalone application by using "spark-submit".
- Spark UI at http://localhost:4040 maintains information on the state of Spark jobs and dependencies (as a DAG), environment, and cluster state, very useful for tuning and debugging.
  - A Spark job represents a set of transformations triggered by an individual action.
- Spark performs quick analytics interactively through notebooks (e.g., Jupyter, Spark-Notebook, Databricks notebooks, and Apache Zeppelin.)
- Tungsten: an umbrella project of Apache Spark's execution engine for performance.
  - Memory management and binary processing: eliminate the overhead of the JVM object model and garbage collection.
  - Cache-aware computation: (e.g., in-memory encoding, etc.)
  - Whole-stage ode generation via compilers
    - No virtual function dispatches
    - Intermediate data in memory vs. CPU registers
    - Loop unrolling and SIMD
- Spark core *immutable* abstractions: Datasets, DataFrames, SQL Tables, Resilient Distributed Datasets (RDD)
- Spark's APIs
  - Low level "unstructured" API
  - High level structured API

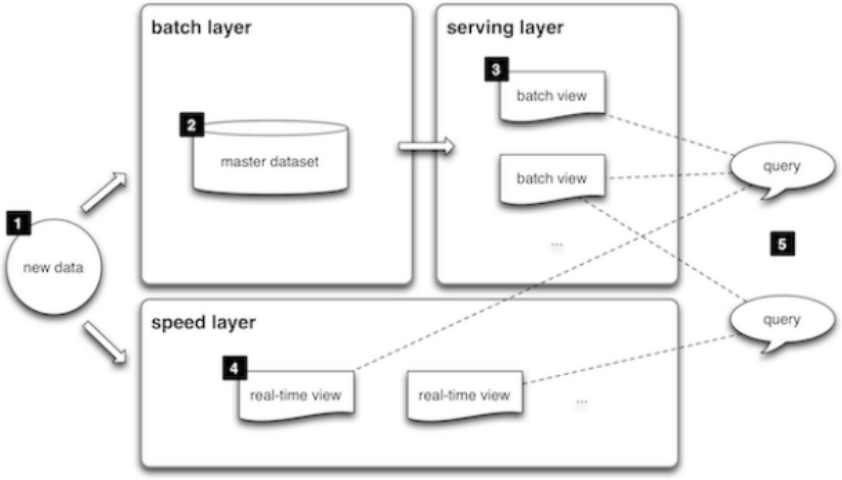
Low Level Unstructured API

- lower level primitives on Resilient Distributed Datasets (RDDs) in 2011, which is distributed collection of immutable Java Virtual Machine (JVM) objects, and RDDs are calculated against, cached, and stored in-memory.
- RDDs are the backbone of Apache Spark, and are schema-less data structures.
- RDD is split into chunks based on some key and distributed to executor nodes and operate in parallel. RDDs keep track (log) of all the transformations applied to each chunk to speed up the computations.
- Lineage-based fault recovery: RDDs guards against data loss and provide a fallback (data lineage) - if a partition of an RDD is lost it still has enough information to recreate that partition without requiring replication (which is expensive for data-intensive workloads). Each RDD tracks the graph of transformations (lineage graph) that was used to build it, and reruns these operations on base data to reconstruct any lost partitions.
- By default, Spark's RDDs are "ephemeral," in that they get recomputed each time they are used in an action. However, users can also persist selected RDDs in memory for rapid reuse. If the data does not fit in memory, Spark will automatically spill it to disk using LRU eviction policy.
  - Checkpointing (with a REPLICATE flag to persist) is used for long lineage graphs containing wide dependencies.
- Interface used to represent RDDs: partitions(), preferredLocations(), dependencies(), iterator(), partitioner(), etc.

Lambda Architecture

- a generic, scalable data-processing architecture designed to handle massive quantities of data by taking advantage of both batch- and stream-processing methods, balance latency, throughput, and fault-tolerance by using batch processing to provide comprehensive and accurate views of batch data, while simultaneously using real-time stream processing to provide views of online data

- lambda architecture describes a system consisting of three layers: batch processing, speed (or real-time) processing, and a serving layer for responding to queries



High Level Structured API

- Both DataFrames and DataSets are built upon RDDs and Spark SQL engine, form the core high-level and structured distributed data abstraction, and provide a uniform API across libraries and components in Spark.
- Spark DataFrames (2013): the most common structured API represents a table of data with rows and columns/schema.
  - A Spark DataFrame is an immutable distributed collection of data organized into named columns (resemble tables in relational database, impose a structure and schema) and can span multiple computers, but Python Pandas' DataFrame (or R's data.frames or data.tables) exist on one machine.
  - We usually do not manipulate chunks/partitions manually. We simply specify high-level transformations of data in the physical partitions and Spark determines how this work will actually execute on the cluster.
- Spark Datasets (2015): a distributed collection of data that can be constructed from JVM objects and then manipulated using functional transformations, using type-safe structured (i.e., object-oriented, strict compile-time type safety, and errors are caught at compile-time) APIs (available in Scala or Java but not available in Python and R because they are dynamical typed or not-type-safe languages).
  - The Dataset API allows users to work with semi-structured data (like JSON or key-value pairs) by assigning a Java class to the records inside a DataFrame, and manipulate it as a collection of typed objects, similar to a Java ArrayList or Scala Seq.
  - The Dataset class is parametrized with the type of object contained inside: Dataset<T> in Java and Dataset[T] in Scala. The types T supported are all classes following the JavaBean pattern in Java, and case classes in Scala.
  - It is easy to drop down to lower level, perform type-safe coding when necessary, and move higher up to SQL for more rapid analysis.
  - Spark optimizes DataSet through Catalyst optimizer, and generate efficient bytecode through Project Tungsten.
  - Whether you express your computations in Spark SQL, Scala, Java, Python, or R DataSet/DataFrame APIs, the underline code generated is identical.
- Unify Datasets and DataFrames (backward compatibility?)
  - Untyped API, e.g., DataFrame = Dataset[Row], and alias.
  - Typed API, e.g., DataSet[T] (in Scala) or DataSet<T> (in Java).

SparkSQL

- Formally, Hive on Spark (Shark), now SparkSQL
- Python query speeds were often twice as slow as the same Scala queries using RDDs
- SparkSQL: allows a user to register any DataFrame as a table or view (a temporary table) and query it using pure SQL. There is no performance difference between writing SQL queries or writing DataFrame code, they both "compile" to the same underlying plan that we specify in DataFrame code via Catalyst optimizer.
- SparkSQL provides common and uniform access to Hive, Avro, Parquet (a columnar format), ORC, JSON, JDBC/ODBC, etc.

MLlib and ML

- MLlib targets large-scale learning with data-parallelism or model-parallelism on RDDs. MLlib allows for preprocessing, munging, training of models, and making predictions at scale on data.
  - Data preparation: feature discretization and extraction, transformation, selection, hashing of categorical features, natural language processing methods, etc.
  - Machine learning algorithms: classification, regression, collaborative filtering, clustering, dimensionality reduction, etc.
  - Utilities: statistics, descriptive statistics, chi-square testing (for categorical features), C++-based linear algebra (dense or sparse), model evaluation methods, I/O formats, optimization primitives, etc.
    - Descriptive statistics: count(), max(), mean(), min(), normL1(), normL2(), numNonzeros(), variance(), etc.

- ML (spark.ml) operated on DataFrames with 3 main abstract classes:
  - Transformer: every new Transformer needs to implement a .transform() method. The spark.ml.feature contains Binarizer, Bucketizer, ChiSqSelector, CountVectorizer, DCT (discrete cosine transform), ElementwiseProduct, HashingTF (hashing trick transformer), IDF (inverse document frequency), IndexToString, MaxAbsScaler ([-1.0, 1.0]), MinMaxScaler ([0.0, 1.0]), NGram, Normalizer, OneHotEncoder, PCA (principal component analysis), PolynomialExpansion, QuantileDiscretizer, RegexTokenizer, RFormula, SQLTransformer, StandardScaler (0 mean and standard deviation of 1), StopWordRemover, StringIndexer, Tokenizer, VectorAssembler, VectorIndexer, VectorSlicer, Word2Vec, etc.
  - Estimator: make prediction or perform classification. Every new Estimator needs to implement a .fit() method.
    - Classification: LogisticRegression, DecisionTreeClassifier, GBTClassifier (gradient boosted trees classifier), RandomForestClassifier, NaïveBayes, MultilayerPerceptronClassifier, OneVsRest, etc.
    - Regression: AFTSurvivalRegression (accelerated failure time survival regression), DecisionTreeRegressor, GBTRegressor, GeneralizedLinearRegression (supports Gaussion, binomial, and poisson distribution with kernel/link functions), IsotonicRegression, LinearRegression, RandomForestRegressor, etc.
    - Clustering: BisectingKMeans (hierarchical), KMeans, GaussianMixture, LDA (latent Dirichlet allocation),
  - Analytic models using statistics or machine learning algorithms which are intensive iterative processes, and pipeline (better implemented in C++/GO/Java) can make this process efficient.
  - Pipeline: an end-to-end transformation-estimation process consists of discrete stages - a sequence of data pre-processing, feature extraction, model fitting, and validation stages – using stages parameter, the output of a preceding stage (using .getOutptCol()) is the input for the following stage.
    - The .fit() method returns the PipelineModel object, which can be used to predict using .transform() method.
    - You can save the Pipeline definition for later use using .save() method, and load it up by the .load method. Virtually all model returned by .fit() on an Estimator or Transformer can be saved and loaded back for reuse.
- Training machine learning models is a two-phase process. First, we initialize an untrained model, then we train it.
- The pyspark.ml.evaluation contains evaluator for evaluating the performance of the model.
- Parameter hyper-tuning (pyspark.ml.tuning): find the best parameters of the model
  - Grid search: an exhaustive algorithm, may take a lot of time to select the best model.
  - Train-validation splitting:

GraphX and GraphFrames

- Graph structures: vertices, edges, and properties:
  - Social networks: the vertices are the people while the edges are the connections between them. Restaurant recommendations: the vertices involve the location, cuisine type, and restaurants while the edges are the connections between them. Create a social network + restaurant recommendation graph based on the reviews of friends within a social circle
  - Analysis of flight data: airports are represented by vertices and flights between those airports are represented by edges. Properties: departure delays, plane type, and carrier, etc.
- GraphX and GraphFrames for graph processing: pageRank (regular and personalized), connected components, label propagation algorithm (LPA), SVD++, strongly connected components, shortest paths, breadth-first search, triangle count, etc. It has no Java or Python APIs and based on low-level RDDs.
- GraphX unifies ETL, exploratory analysis, and iterative graph computation within a single system
- GraphFrames
  - GraphFrames utilizes the power of Apache Spark DataFrames to support general graph processing similar to Spark's GraphX library but with high-level, expressive and declarative APIs in Scala, Java and Python. GraphFrames can seamlessly converted into GraphX.
  - Leverages the distribution and expression capabilities of the DataFrame API to both simplify your queries and leverage the performance optimizations of the Apache Spark SQL engine.
  - GraphFrames consist of DataFrames of vertices and edges.

```
// create a Vertices DataFrame
val vertices = spark.createDataFrame(
  List(("JFK", "New York", "NY"))).toDF("id", "city", "state")
// create a Edges DataFrame
val edges = spark.createDataFrame(List(("JFK", "SEA", 45,
1058923))).toDF("src", "dst", "delay", "tripID")
// create a GraphFrame and use its APIs
val airportGF = GraphFrame(vertices, edges)
// filter all vertices from the GraphFrame with delays greater an
// 30 mins
val delayDF = airportGF.edges.filter("delay > 30")
// Using PageRank algorithm, determine the Airport ranking of
// importance
val pageRanksGF =
airportGF.pageRank.resetProbability(0.15).maxIter(5).run()
```

```
display(pageRanksGF.vertices.orderBy(desc("pagerank")))
```

- 3 kinds of queries: SQL-type queries on vertices and edges, graph-type queries, and motif queries (providing a structure pattern)
- Finding structural motifs, airport ranking using PageRank, and shortest paths between cities, breadth first search (BFS), strongly connected components, saving and loading graphs, etc.

- GraphFrames vs. GraphX

	GraphFrames	*GraphX*
Built on	DataFrames	RDDs
languages	Scala, Java, Python	Scala
Use cases	Queries and algorithms	Algorithms
Vertex IDs	Any type (in Catalyst)	Long
Vertex/edge attributes	Any number of DataFrame columns	Any type (VD, ED)
Return types	GraphFrame or DataFrame	Graph(VD, ED), or RDD(Long, VD)

Deep Learning and TensorFrames (Deprecated)

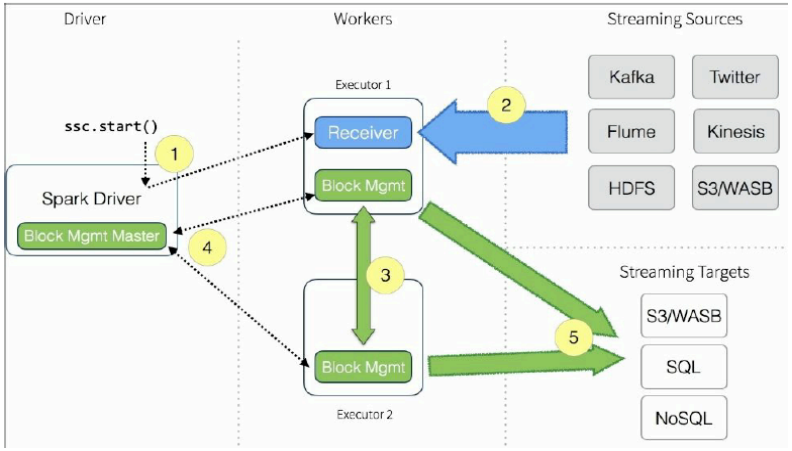
- Traditional algorithmic approach: programming known steps or quantities, that is, you already know the steps to solve a specific problem
- Human brain has about 100 billion neurons in our brain, each connected to approximately 10,000 other neurons, resulting in a mind-boggling 10^{15} synaptic connections.
- Neural networks learn by example and are not actually programmed to perform a specific task
  - Learning processes – modifications of the connections between the interconnected elements
  - Multilayer perceptron (MLP) has three layers: input, hidden, and output
- Machine learning need feature engineering
  - Feature engineering is about determining which of these features (independent variables) are important in defining the model. Coming up with features is difficult, time-consuming, requires expert knowledge.
  - Feature selection based on domain knowledge to select useful subset of the features
  - Feature extraction: transform the data from a high dimensional space to a smaller space of fewer dimensions, e.g., principal component analysis (PCA).
- Deep Learning is part of a family of machine learning methods. Deep learning replace or minimize the need for manual feature engineering - automating feature engineering or teaching machines to learn how to learn (unsupervised Feature Learning).
  - Deep learning success by
    - Advances and availability of distributed computing (e.g., Apache Spark) and hardware (especially GPUs – graphic processing units)
    - Advances in deep learning research: Deep learning libraries: TensorFlow, Theano, Torch, Caffe, Microsoft Cognitive Toolkit (CNTK), mxnet, and DL4J.
  - Applications facial recognition, handwritten digit identification, game playing, speech recognition, language translation, and object classification.
- TensorFlow (the flow of tensors) is a Google open source software library for numerical computation using data flow graphs by Google's BrainTeam. Built on C++ with a Python interface.
- TensorFrames
  - With TensorFrames, one can manipulate Spark DataFrames with TensorFlow programs.
  - TensorFrames provides a bridge between Spark DataFrames and TensorFlow. This allows you to take your DataFrames and apply them as input into your TensorFlow computation graph. TensorFrames also allows you to take the TensorFlow computation graph output and push it back into DataFrames.
  - Parallel training to determine optimal hyperparameters (configuration, learning rate, number of neurons in each layer, etc.)

Polyglot Persistence with Blaze

- Solve your problems with the tools that are designed to solve them easily. But, this does not make your company agile and is prone to errors and lots of tweaking and hacking needing to be done.
- Our world is complex and no single approach exists that solves all problems. There is no such thing as a one-size-fits-all solution.
- Polyglot persistence
  - Polyglot programming: using multiple programming languages that were more suitable for certain problems.
  - In the parallel world of data, adapt a range of technologies that allows it to solve the problems in a minimal time, thus minimizing the costs.
  - Polyglot persistence: always chooses the right tool for the right job instead of trying to coerce a single technology into solving all of its problems, e.g., persisting transaction records in a relational database.
- Blaze abstracts most of the technologies and exposes a simple and elegant data structure and API.
  - Working with databases: Blaze can read from SQL databases such as PostgreSQL (http://localhost:5432/) or SQLite, and NoSQL such as MongoDB database (http://localhost:27017).

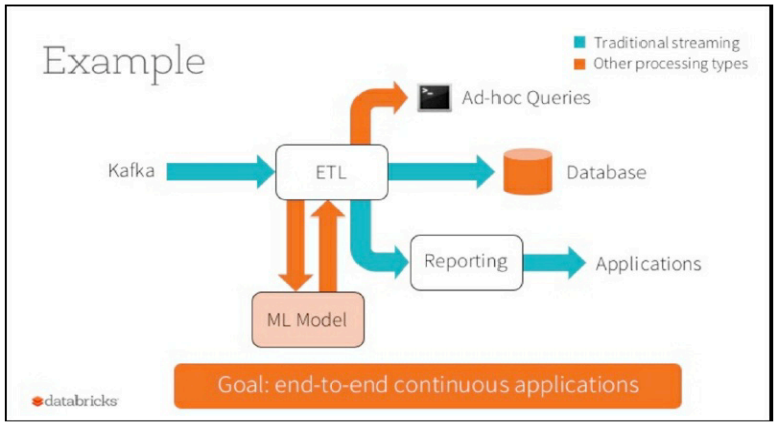
Spark Streaming

- Spark Streaming: a single API addresses both batch and streaming (including late events, partial outputs to the final data source, state recovery on failure, and/or distributed reads/writes.) and the benefits are performance, event time, windowing, sessions, sources, and sink.
  - Spark Streaming is a scalable, fault-tolerant streaming system that takes the RDD batch paradigm (in Scala) and speeds it up.
  - Spark Streaming is a high-level API that provides fault-tolerant exactly-once semantics for stateful operations.
  - Spark Streaming operates in mini-batches or batch intervals (from 500ms to larger interval windows) at scale and in real time.
  - Microbatches: Spark runs each streaming computation as a series of short batch jobs.
  - The key abstraction for Spark Streaming is Discretized Stream (DStream) built on RDDs.
  - Spark Streaming integrates with MLlib, Spark SQL, DataFrames, and GraphX. Apache Spark unifies all of these disparate data processing paradigms within the same framework.
  - Spark Streaming has built in receivers that can take on many sources, with the most common being Apache Kafka, Flume, HDFS/S3, Kinesis, and Twitter.
  - Currently, there are four broad use cases surrounding Spark Streaming:
    - Streaming ETL: Data is continuously being cleansed and aggregated prior to being pushed downstream.
    - Triggers: Real-time detection of behavioral or anomaly events trigger immediate and downstream actions.
    - Data enrichment: Real-time data joined to other datasets allowing for richer analysis.
    - Complex sessions and continuous learning: Multiple sets of events associated with real-time streams are continuously analyzed and/or updating machine learning models.
  - Spark Streaming application data flow



1. starts with the Spark Streaming Context, ssc.start()
2. When the Spark Streaming Context starts, the driver will execute a long-running task on the executors (that is, the Spark workers). The Receiver on the executors (Executor 1 in this diagram) receives a data stream from the Streaming Sources. With the incoming data stream, the receiver divides the stream into blocks and keeps these blocks in memory.
3. These blocks are also replicated to another executor to avoid data loss.
4. The block ID information is transmitted to the Block Management Master on the driver.
5. For every batch interval configured within Spark Streaming Context (commonly this is every 1 second), the driver will launch Spark tasks to process the blocks. Those blocks are then persisted to any number of target data stores.

- End-to-end applications that continuously react to data in real-time:



- Structured Streaming (Spark 2.0)
  - Structured Streaming extended support for data source and data sinks, and buttressed streaming operations including event-time processing, watermarking, and checkpointing.
  - Structured Streaming utilizes Spark DataFrames. It allows you to take the same operations that you perform in batch mode using Spark's structured APIs, and run them in a streaming fashion. This can reduce latency and allow for incremental processing.
  - Incremental Execution Plan: structured streaming repeatedly applies the execution plan for every new set of blocks it receives.
  - When is a stream not a stream: treat a stream of data not as a stream but as an unbounded table. As new data arrives from the steam, new rows of DataFrames are appended to an unbounded table. Developers can express their steaming computations just like batch computations, and Spark will automatically execute it incrementally as data arrives in the stream.
  - 3 built-in data sources:
    - File source (local drive, HDFS, S3 bucket) implementing the DataStreamReader interface which supports data formats (avro, JSON, CSV).
    - Apache Kafka source: poll or read data from subscribed topics adhering to all Kafka semantics.
    - Network socket source: read UTF-8 text data from a socket connection (IP and port pair), does not guarantee any end-to-end fault-tolerance as the other two sources do.
  - 4 built-in data sinks implementing foreach interface:
    - File sink: directories or files
    - Foreach sink implemented by the developer, a ForeachWriter interface writes your data to the destination (NoSQL/JDBC sink, listening socket, REST call to an external service) using open(), process(), and close().
    - Console/stdout sink for debugging (incurs heavy memory usage on the driver side)
    - Memory sink for debugging: data is stored as an in-memory table
- Streaming operations on DataFrames and DataSets: select, projection, aggregation

```
# streaming DataFrame with IOT device data with schema
# {device: string, type: string, signal: double, time: DateType}
devicesDF = ...
# Select the devices which have signal more than 10
devicesDF.select("device").where("signal > 10")
# Running count of the number of updates for each device type
devicesDF.groupBy("type").count()
```

- Windowing event time aggregations to handle out-of-order or late data

```
import spark.implicits._
// streaming DataFrame of schema
//   { timestamp: Timestamp, word: String }
val words = ...
// Group the data by window and word and compute the count of
// each group
val deviceCounts = devices.groupBy(
  window($"timestamp", "10 minutes", "5 minutes"), $"type"
).count()
```

- Watermarking: mark or specify a threshold in your processing timeline interval beyond which any data's event time is deemed useless (i.e., retain only late data within that time threshold or interval), it can be discarded without ramifications.

```
val deviceCounts = devices
  .withWatermark("timestamp", "15 minutes")
  .groupBy(
    window($"timestamp", "10 minutes", "5 minutes"), $"type")
  .count()
```

- Continuous applications problems: late events, partial outputs, state recovery on failure, distributed reads and writes, and so on.

SparkR

PySpark