- **DataFrames and Resilient Distributed Datasets**
- Review: Spark Driver and Workers
  - A Spark program is two programs
    - A driver program and a workers program
  - Worker programs run on cluster nodes or in local threads
  - DataFrames are distributed across workers
- Review: Spark and SQL Contexts
  - spark program first creates a SparkContext object
    - SparkContext tells Spark how and where to access a cluster
  - program next creates a sqlContext
  - use sqlContext to create DataFrames
- Review: DataFrames
  - Primary abstraction in Spark
    - Immutable once constructed
    - Track lineage information to efficiently recompute lost data
    - Enable operations on collection of elements in parallel
  - You construct DataFrames
    - by parallelizing existing Python collections (lists)
    - by transforming an existing Spark or pandas DFs
    - from files in HDFS or any other storage system
  - Two types of operations: transformation and actions
    - Transformations are lazy (not computed immediately)
    - Transformed DF is executed when action runs on it
    - Persist (cache) DFs in memory or disk
- Resilient Distributed Datasets
  - Untyped Spark abstraction underneath DataFrames:
    - Immutable once constructed
    - Track lineage information to efficiently recompute lost data
    - Enable operations on collection of elements in parallel
  - You construct RDDs
    - by parallelizing existing Python collections (lists)
    - by transforming an existing RDDs or DataFrame
    - from files in HDFS or any other storage system
- RDDs
  - programmer specifies number of partitions for an RDD
    - more partitions = more parallelism
  - Two types of operations transformations and actions
  - Transformed RDD is executed when action runs on it
  - Persist (cache) RDDs in memory or disk
- When to use DataFrames?
  - Need high-level transformations and actions, want high-level control over your dataset
  - Have typed (structured or semi-structured) data
  - You want DataFrame optimization and performance benefits
- DataFrame Performance
  - Faster than RDDs
  - Memory Usage when Caching
- When to use RDDs?
  - Need low-level transformations and actions, and want low-level control over your dataset
  - Have unstructured or schema-less data

  - Want to manipulate your data with functional programming constructs other than domain specific expressions
  - don’t want the optimization and performance benefits available with DataFrames
- Working with RDDs
  - create from a datasource
  - apply transformations
  - apply actions: collect count
  - <list> parallelize => filter => map => collect => result
- Creating an RDD
  - Create RDDs from Python collections (lists)
  - sc.parallelize(data,40
  - sc.textFile() etc…
- Spark Transformations
  - create new datasets from an existing one
  - use lazy evaluation: results not computed right away — instead Spark remembers set of transformations applied to base dataset
    - Spark optimizes the required calculations
    - Spark recovers from failures and slow workers
- RDD transformations
  - parallelize
  - map
  - filter
  - flatMap
  - distinct
- Transforming an RDD
  - lines = sc.textFile(,4)
  - comments = lines.filter(isComment)
- Spark Actions
  - cause spark to execute recipe to transform source
  - Mechanism for getting results out of Spark
- Some Actions
  - reduce(func)
    - aggregate dataset’s elements using function func, fun takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel
  - take(n)
    - return an array with the first n elements
  - collect()
    - return all the elements as an array
  - takeOrdered(n, key=func)
    - return n elements ordered in ascending order or as specified by the optional key
- Getting Data Out of RDDs
  - rdd = sc.parallelize([1,2,3])
  - rdd.reduce(lambda a, b: a * b)
  - rdd.take(2)
  - rdd.collectIO
- Spark Key-Value RDDs
  - Each element of a Pair RDD is a pair tuple
  - RDD: [(1,2), (3,4)]
- Some Key-Value Transformations

- reduceByKey(func)
  - return a new distributed dataset of (K,V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V
- sortByKey()
  - return a new dataset(K,V) pairs sorted by keys in ascending order
- groupByKey()
  - return a new dataset of (k, Iterable<V>) pairs
  - it can cause a lot of data movement across the network and create large Iterables at workers

- Spark RDD Programming Model
- Spark Programming Model
  - count() causes Spark to
    - read data
    - sum within partitions
    - combine sums in driver
- Caching RDDs
- Spark Program Lifecycle with RDDs
  - Create RDDs from external data or parallelize a collection in your driver program
  - Lazily transform them into new RDDs
  - cache() some RDDs for reuse
  - Perform actions to execute parallel computation and produce results
- Spark Shared Variables
- pySpark Closures
  - Spark automatically creates closures for:
    - Functions that run on RDDs at executors
    - Any global variables used by those executors
  - One closure per executor
    - Sent to every task
    - No communication between executors
    - Changes to global variables at executors are not sent to driver
- Consider These Use Cases
  - Iterative or single jobs with large global variables
    - Sending large read-only lookup table to executors
    - Sending large feature vector in ML algorithm to executors
  - Counting events that occur during job execution
    - How many input lines were blank?
    - How many input records were corrupt?
  - Problems
    - closures resent for every job
    - inefficient to send large data to each worker
    - closures are one way: driver => worker
- pySpark Shared Variables
  - Broadcast Variables
    - Efficiently send large, read-only value to all executors
    - Saved at workers for use in one or more Spark operations
    - Like sending a large, read-only lookup table to all the nodes
  - Accumulators
    - Aggregate values from executors back to driver
    - Only driver can access value of accumulator

        - For tasks, accumulators are write-only
        - Use to count errors seen in RDD across executors
- Broadcast Variables
    - read-only variable cached on executors
        - ship to each worker only once instead of with each task
    - example: efficiently give every executor a large dataset
    - Usually distributed using efficient broadcast algorithms
- Accumulators
    - Variables that can only be “added” to by associative op
    - Used to efficiently implement parallel counters and sums
    - Only driver can read an accumulator’s value, not tasks
    - Tasks at executors cannot access accumulator’s values
    - Tasks see accumulators as write-only variables
    - Accumulators can be used in actions or transformations
    - Types: integers, double, long, float
- File Performance
    - file
        - named sequence of bytes
        - typically stored as collection of pages (or blocks)
    - filesystem is a collection of files organized within a hierarchical namespace
        - Responsible for laying out those bytes on physical media
        - stores file metadata
        - Provides an API for interaction with files
    - Standard operations
        - open() / close()
        - seek()
        - read() / write()
- Considerations for a File Format
    - Data model: tabular, hierarchical, array
    - Physical layout
    - Field units and validation
    - Metadata: header, side file, specification, other?
    - Plain text or binary
    - Delimiters and escaping
    - Compression, encryption, checksums
    - Schema evolution
- File Performance considerations
    - Read versus write performance
    - Plain text verus binary format
    - Environment: Panda (Python) versus Scala/Java
    - Uncompressed versus compressed
- File Performance - Summary
    - Uncompressed read and write times are comparable
    - Binary I/O is much faster than text I/O
    - Compressed reads much faster than compressed writes