- Key Data Management Concepts
  - A data model
    - is a collection of concepts for describing data
  - A schema
    - is a description of a particular collection of data using a given data model
- Structure Spectrum
  - Structured
    - schema first
    - relational database
  - Semi-structured
    - schema later
    - documents xml
  - Unstructured
    - schema never
    - plain text
- Semi-Structured Tabular dat
  - A table is a collection of rows and columns
  - Each column has a name
  - Each cell may or may not have a value
  - Each column has a type (String, integer)
    - together the column type are the schema for the data
  - Two choices for how the schema is determined
    - spark dynamically infers the schema while reading each row
    - programmer statically specifies the schema
- Structured Data
  - a relational data model is the most used data model
    - relation,, a table with rows and columns
  - every relation has a schema defining each columns' type
  - the programmer must statically specify the schema
- Unstructured data
  - only one column with string or binary type
- The structure spectrum
  - extract-transform-load
    - impose structure on unstructured data

Analysis, Big Data, and Apache Spark

- What is Apache Spark?
  - scalable, efficient analysis of big data
- Some Traditional Analysis Tools
  - Unix shell commands, pandas, R
  - all run on a single machine
- Real world spark analysis use cases
  - big data genomics using adam
  - data processing for wearables and internet of things
- The Big Data Problem
  - Data growing faster than computation speeds
  - Growing data sources
  - Storage getting cheaper
  - But, stalling CPU speeds and storage bottlenecks
  - One machine can not process or even store all the data

  - solution is to distribute data over cluster of machines
  - the key is it is all about memory
- Distributed Memory
  - Big data partition into multiple frames
    - sparks data frame
- The Spark Computing Framework
  - Provides a programming abstraction and parallel runtime to hid complexities of fault-tolerance and slow machines
- Apache Spark Components
  - Spark SQL
  - Spark Streaming
  - MLib and ML (Machine Learning)
  - GraphX (graph)
  - Apache Spark
- Python Spark (pySpark)
  - pySpark provides an easy-to-use programming abstraction and parallel runtime
    - "heres an operation, run it on all of the data"
  - DataFrames are the key concept
- 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
- Spark and SQL Contexts
  - A Spark program first creates a SparkContext object
    - SparkContext tells Spark how and where to access a cluster
    - pySpark shell, Databricks CE automatically create SparkContext
    - iPython and programs must create a new SparkContext
  - The program next creates a sqlContext object
  - use sqlContext to create DataFrames
- Spark Essentials: Matter
  - The master parameter for a SparkContext determines which type and size of cluster to use

Apache Spark DataFrames

- DataFrames
  - The 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
    - by transforming an existing Spark or pandas DF
    - from files in HDFS or any other storage system
  - Each row of a DataFrame is a Row object
    - The fields of a Row can be accessed like attributes
  - Two types of operations: transformations and actions
  - Transformations
    - are lazy (not computed immediately)
    - transformed DF is executed when action runs on it
    - Persist (cache) DFs in memory or disk

- Working with DataFrames
  - create DataFrame from data source
  - apply transformations to DataFrame: select filter
  - Apply actions to DataFrame: show count
- Creating DataFrames
  - Create DataFrames from Python collections(lists)
- pandas: Python Data Analysis Library
  - Open source data analysis and modeling library
    - an alternative to using R
  - pandas DataFrame: a table with named columns
    - the most commonly used pandas object
    - represented as python Dict(column_name -> series)
    - Each pandas series Object represents a column
      - 1-D labeled array capable of holding any data type
    - R has a similar data frame type
- Creating DataFrames
  - easy to create pySpark DataFrames from pandas DataFrames
  - spark_df = sqlContext.createDataFrame(pandas_df)
  - Fro HDFS, text files, JSON files, Apache Parquet, Hypertable, Amazon S3, Apache Hbase, SequenceFiles, any other Hadoop InputFormat, and directory or glob wildcard;
    - df = sqlContext.read.text("sdfsf")
    - df.collect()
- Creating a DataFrame from a file
  - distFile = sqlContext.read.text("..")
  - loads text file and returns a DataFrame with a single string column named "value"
  - each line in text file is a row
  - lazy evaluation means no execution happens now

Apache Spark Transformations

- Spark Transformations
  - Create a new DataFrame from an existing one or source
  - lazy evaluation result not computed right away
  - spark remembers set of transformations applied to base DataFrame
    - spark uses Catalyst to optimize the required calculations
    - spark recovers from failures and slow workers
  - Think of this as a recipe for creating result
- Column transformations
  - apply method creates a DataFrame from one column
    - ageCol = people.age
  - You can select one or more columns from a DataFrame
    - df.select('*')
      - * selects all the columns
    - df.select('name','age')
      - * selects the name and age columns
    - df.select(df.name, (df.age + 10).alias('age'))
      - * selects the name and age columns, increments the values in the age column by 10, and renames (alias) the age + 10 columns as age
- More Column transformations
  - The drop method returns a new DataFrame that drops the specified column:
    - df.drop(df.age)

- Review: Python lambda Functions
  - small anonymous functions (not bound to a name)
    - lambda a, b: a + b
      - returns the sum of its two arguments
  - can use lambda functions wherever function objects are required
  - restricted to a single expression
- User Defined Function Transformations
  - transform a DataFrame using User Defined Function
    - from pyspark.sql.types import IntegerType
    - slen = udf(lambda s: len(s), IntegerType())
    - df.select(slen(df.name).alias('slen'))
      - * creates a DataFrame of [Row(slen=5), Row(slen=3)]
  - UDF takes named or lambda function and the return type of the function
- Other Useful transformations
  - filter(func)
    - returns a new DataFrame formed by selecting those rows of the source on which function returns true
  - where(func)
    - where is an alias for filter
  - distinct()
    - return a new DataFrame that contains the district rows of the source DataFrame
  - orderBy(*cols, **kw)
    - returns a new DataFrame sorted by the specified column(s) and in the sort order specified by kw
  - sort(*cols, **kw)
    - Like orderBy.sort returns a new DataFrame sorted by the specified colun(s) and in the sort order specified by kw
  - explode(col)
    - returns a new row for each element in the given array or map
  - func is a Python named function or lambda function
- Using transformations
  - df = sqlContext.createDataFrame(data, ['name','age'])
    - [Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
  - from pyspark.sql.types import IntegerType
  - doubled = udf(lambda s: s * 2, IntegerType())
  - df2 = df.select(df.name, doubled(df.age).alias('age'))
    - [Row(name=u'Alice', age=2), Row(name=u'Bob', age=4)]
    - *selects the name and age columns, applies the UDF to age column and alias resulting column to age
  - df3 = df2.filter(df2.age > 3)
  - example
    - data2
    - df = sqlContext.createDataFrame(data2,['name','age'])
    - df2 = df.distinct()
      - * only keeps rows that are distinct
    - df3 = df2.sort("age", ascending=false)
      - * sort ascending on the age column
  - example
    - data3 = [Row(a=1, intlist[1,2,3])]

        - df4 = sqlContext.createDataFrame(data3)
            - [Row(a=1, intlist=[1,2,3])]
        - df4.select(explode(df4.intlist).alias(“anInt”))
            - [Row(anInt=1), Row(anInt=2), Row(anInt=3)]
            - * turn each element of the int list column into a Row, alias the resulting column to anInt, and select only that column
- GroupedData Transformations
    - groupBy(*cols) groups the DataFrame using the specified columns, so we can run aggregation on them
    - agg(*exprs)
        - compute aggregates (avg, max, min, sum, or count) and returns the result as a DataFrame
    - count()
        - counts the number of records for each group
    - avg(*args)
        - computes average values for numeric columns for each group
- Using GroupedData (1)
    - Example
        - data
        - df = sqlContext.createDataFrame(data, [‘name’,’age’,’grade’]
        - df1 = df.groupBy(df.name)
        - df1.agg({“*”: “count”}).collect()
    - Example (same results as the previous example)
        - data
        - df = sqlContext.createDataFrame(data, [‘name’,’age’,’grade’]
        - df.groupBy(df.name).count()
    - Example (computes the average across all the numerics)
        - data
        - df = sqlContext.createDataFrame(data, [‘name’,’age’,’grade’]
        - df.groupBy().avg().collect()
            - [Row(avg(age)=2.5, avg(grade)=7.5)]
    - Example (computes the average for age and grade grouping by name)
        - data
        - df = sqlContext.createDataFrame(data, [‘name’,’age’,’grade’]
        - df.groupBy(‘name’).avg(‘age’, ‘grade’).collect()
- Transforming a DataFrame
    - linesDF = sqlContext.read.text(‘…’)
        - creates a lines DataFrame
        - each line of the file
    - commentsDF = linesDF.filter(isComment)
    - Lazy evaluation - means nothing executes - Spark saves recipe for transforming source

Apache Spark Actions

- Spark Actions
    - Cause Spark to execute recipe to transform source
    - Mechanism for getting results out of Spark
- Some useful actions
    - show(n, truncate)
        - prints the first n rows of the DataFrame
    - take(n)

        - returns the first n rows as a list of Row
    - collect() (should never use collection in production applications) (make sure will fit in driver program)
        - return all the records as a list of Row
    - count()
        - returns the number of rows in this DataFrame
        - count for DataFrames is an action, while for GroupedData it is a transformation
    - describe(*cols)
        - Exploratory Data Analysis function that computes statistics (count, mean, stddev, min, max) for numeric columns - if no columns are given, this function computes statistics for all numerical columns
- Getting Data Out of DataFrames
    - Example
        - data
        - df = sqlContext.createDataFrame(data, [‘name’,’age’]
        - df.collect()
        - [Row(name=u’Alice’, age=1), Row(name=u’Bob’, age = 2)]
        - df.show()
        - df.count()
        - df.take(1)
        - df.describe() (only works on numerical columns)
- Spark Programming Model
    - linesDf = sqlContext.read.text(‘….’)
    - print linesDF.count()
        - count() causes Spark to:
            - read data
            - sum within partitions
            - combine sums in driver
    - commentsDF = linesDF.filter(isComment)
    - print linesDF.count()
    - commentsDF.count()
        - Spark recomputes linesDF:
            - read data (again)
            - sum within partitions
            - combine sums in driver
- Spark Program Lifecycle
    - Create DataFrames from external data or createDataFrame from a collection in driver program
    - Lazily transform them into new DataFrames
    - cache() some DataFrames for reuse
    - Perform actions to execute parallel computation and produce results

Best Programming Practices

- Local or Distributed?
    - Where does code run?
        - Locally, in the driver
        - Distributed at the executors
        - Both at the driver and the executors
    - Very important question:
        - Executors run in parallel

        - Executors have much more memory
- Where Code Runs
    - Most Python code runs in driver
        - except for code passed to transformations
    - Transformations run at executors
    - Actions run at executors and driver
- Examples
    - a = a + 1
        - runs in Your application (driver program)
    - linesDF.filter(isComment)
        - Spark executor
    - commentsDF.count()
        - Runs at both driver and executor
- How Not to Write Code
    - Let's say you want to combine two DataFrames: aDF, bDF
    - You remember that df.collect() returns a list of Row, and in Python you can combine two lists with +
    - A naive implementation would be
        - a = aDF.collect()
        - b = bDF.collect()
        - cDF = sqlContext.createDataFrame(a+b)
    - Where does this code run?
- a + b
    - a = aDF.collect()
    - b = bDF.collect()
        - all distributed data for a and b is sent to driver
    - What if a and/or b is very large?
        - Driver could run out of memory:
            - Out Of Memory error(OOM)
        - Also, takes a long time to send the data to the driver
- a + b
    - cDF = sqlContext.createDataFrame(a+b)
        - all data for cDF is sent to the executors
    - What if the list a + b is very large?
        - Driver could run out of memory:
            - Our of Memory error(OOM)
        - Also, takes a long time to send the data to executors
- The Best Implementation
    - cDF = aDF.unionAll(bDF)
    - use the DataFrame reference API
        - unionAll():
            - return a new DataFrame containing union of rows in this frame and another frame
        - Runs completely at executors:
            - Very scalable and efficient
- Some Programming Best Practices
    - Use Spark Transformations and Actions wherever possible
    - Never use collect() in production, instead use take(n)
    - cache() DataFrames that you reuse a lot
-

-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
- sdfsdfs