**Data Engineer Concepts**

**Hadoop**

* Distributed system – Cluster (Contains multiple nodes/computers)
  + Store vast amount of data – for Big Data
  + Enormous computing capabilities
    - Parallel processing – divide the workload on multiple machines for time efficient parallel processing – 5TB file can be process on 5 nodes in 1 hour – 1TB file/1 hour/1 machine
    - Scalable (up or down) – Horizontal Scaling/Vertical Scaling – for timely needs
      * Scale up if large files or more times
      * Scale down if small files, save cost
* Hadoop
  + Core
    - HDFS – Storage
    - MapReduce – Processing
    - Yarn – Resource Manager (and nodes management)
  + Ecosystem (layer build on top of Hadoop Core) for ease of use for users
    - Hive – for SQL
    - HBase – for NoSQL
    - Oozie – for schedule jobs (orchestration)
    - Sqoop – for storing data from database into HDFS
    - Pig – for converting file formats
  + Hadoop disadvantages
    - MapReduce used distributed system – but slow and difficult to write
    - Study multiple frameworks in the ecosystem to handle different tasks (for multiple purposes)

**Power of Cloud**

* Need 50 computers/nodes for cluster
* On-premise
  + Actual purchase of servers
  + Need physical location
  + Requires proper cooling system
  + Required engineer teams for installation and maintenance
  + Fixed costs and variable costs are high (up to certain points)
  + Not flexible for scale up or scale down (skyrocketed data amount of Black Friday)
* Cloud
  + No need for actual purchase of nodes (servers)
  + Physical locations are provided by Cloud providers
  + Set up clusters with mouse clicks – easy, quick to set up
  + Small costs for system maintenance
  + Pay-as-you-use, Pay-as-you-go
  + Flexible for scale up and scale down (both Horizontal Scale, and Vertical Scale)
* Cloud types (AWS, Microsoft Azure, Google Cloud Platform (GCP) – major cloud providers)
  + Public – for not too confidential data – for newly started companies
  + Private – for Corporations, Banks – confidential data
  + Hybrid – combination of public and private
* **Servers vs Serverless**
  + Servers
    - Server can fail
    - Suitable for consistent data load
    - More savings in the long terms vs serverless
  + Serverless
    - Mitigate the failure
    - Scalable for flexible data load (Black Friday sales – spiking data load)
    - Saving in short terms for start-up companies

**Microservices**

* Applications Design and Development – Software architecture – Organize components
* Monolithic – Multitier – Microservices
  + Monolithic
    - Design as a single piece of code
    - Encapsulate data storage, business logic, UI
    - Limits when building complex systems – everything is tangled together
    - Difficult to maintain, evolve (add extra functionalities), and scale (for workload)
  + Multitier
    - Each component is divided into tier/layer
    - Common is 3 tiers architecture
      * Presentation layer – UI – front end developers
      * Logic layer – Business logic – back end developers
      * Data layer – Data storage – Database Administrators
    - Still centralized way of architecting – does not solve complex systems problem
  + Microservices
    - Each microservice deals with one function
      * User management microservice
      * Payment processing microservice
      * Products catalog management microservice
      * Have API for communication
* Might have cascade effect (wide effect) when a microservice is down
* Difficult to detect the issue since the design is so distributed
* Containerization
  + Deploy microservices in minimalist runtime (docker)
  + Containers orchestration (Kubernetes)
  + Pipeline automation (GitLab)
  + Message Brokers and Queues (Kafka)
  + Performance monitoring (Prometheus)
  + Logging and audit (DataDog)

**Layers of Sparks (2 primary layers)**

* Spark Core APIs (layer of programming language) – Scala, Python, Java, R
  + Handling basic I/O functions
  + Distribute task scheduling
  + Memory management
  + Implement RDD (fundamental data structure in Spark)
* Higher level APIs – easier to work with spark
  + Spark SQL – provide high level abstraction with Dataframe, Dataset – Combination of RDD and Dataframe
  + Spark Streaming
  + MLlib
  + GraphX

**Resilient Distributed Dataset (Distributed dataset that can recover)**

* Suppose a HDFS (Hadoop Distributed File System)
* Data store in HDFS will be divide to 3 blocks
* Spark handle data in-memory
* When load block of data into memory
  + The data in memory is called RDD
  + RDD.filter()
  + RDD.groupby()
  + Transformation
  + RDD.show()
  + Action
  + Analyze all the transformations to find optimal ways of doing them then action
* When an RDD is missing during transformation
  + Recover back the parent RDD and apply transformation for the child RDD

**Dynamic Frame – Glue – (vs Spark Dataframe)**

* Dynamic Frame
  + Table-like structure that is part of AWS Glue and is built on top of Apache Spark
  + Use to handle varying data structure (semi-structured, unstructured) (JSON, XML)
  + Automatically infer schema and support schema evolution, can handle schema inconsistencies (missing cols, different data types)
  + Have several methods specific to AWS Glue
    - resolveChoice() – Resolve ambiguities in schema
    - applyMapping() – Map fields from one structure to another
    - unbox() – Extract nested fields
* Spark DataFrames
  + Distributed collection of data organized into named columns, built on top of RDD
  + Structured way to manipulate data and is optimized for large-scale data processing in Spark
  + Commonly used for structured, and semi-structured data where schema is well-defined, such as CSV, Parquet, Avro files
  + Required explicit schema, either inferred from data or defined manually
  + Does not handle schema evolution, so schema changes will require additional steps or transformations
  + Optimized for performance through Catalyst query optimization and Tungsten execution engine

**Spark Session (vs Spark Context)**

* Entry point – Gate for operations with Spark
* Main gate – cover multiple Contexts
  + Spark Context
  + Hive Context
  + SQL Context
* 3 levels of architecture
  + RDD – Can only be manipulate by SparkContext
  + Dataframe – Can only be used by SparkSession
  + Spark SQL

**Hadoop Architecture**

* Cluster architecture
  + Storage – HDFS
  + Processing – MapReduce
  + Resource Management – Yarn
* Cluster
  + Combination of nodes
  + Master node
    - Name node
      * Metadata
    - Resource manager
      * Manage the resource for the entire cluster
      * Application manager
        + Receive the client request and check for its capability
      * Provide some resource for the node manager
    - Client request will reach the master node first
  + Worker node
    - Data node
      * Block data
    - Node manager
      * Receive some resource from the resource manager and create an app container (environment) to calculate resource required for tasks
      * Application master
        + Manage the execution of tasks
        + Divide the client requests into multiple small tasks for execution
        + Sent the analysis of resource required back to the resource manager
      * Start to create app container across different nodes to execute task (map task, reduce task)

**Spark Architecture**

* **Storage, Processing, Resource Management**
* Spark is not a replacement for Hadoop, just for the MapReduce, and it can be compatible with any storage and resource management service, Hadoop is fixed (HDFS, MapReduce, YARN)
* Storage
  + HDFS
  + Cloud Storage (S3, Blob Storage)
  + HBase
* Resource Management
  + YARN
  + Standalone (built in lightweight)
  + Mesos
* Cluster
  + Driver node
    - Resource manager sent the client request information to driver node so it creates Spark Session and calculate workload distribution
  + Cluster Manager
    - Manage resource and distribute resource for other worker nodes (CPU, RAM)
    - YARN
    - Resource manager
      * Application manager will analyze the client request to make sure the request can be performed
      * Provide some resource for node manager to create app container for setting up environment of application master
  + Worker node
    - Node manager
      * Application master
        + After calculating and analyzing, sent the information back to resource manager and requests precise amount of resource for the task
      * App container
        + Contain the environment to execute the task
        + Executors

Tasks

* + - Data node
      * Store block of data
  + Name node – because Spark does not have master node
    - Store metadata
* Architecture of Spark varies based on the choices for storage and resource management

**Lazy Transformation of Spark**

* A dataset of sales
* Transformation
  + Load data from Data Lake into memory -> RDD1
  + RDD1.groupby() -> RDD2
  + RDD2.filter() -> RDD3
* Action
  + RDD3.show()
* Spark does not perform transformation step by step when they are called, but rather only do so when there is an action
* Allow for optimal transformation of data
  + Not lazy – Load -> Group by (all data) -> Filter -> Show (part of data) – More Shuffle
  + Lazy – Load -> Filter -> Group by (Part of data) -> Show – Less Shuffle
  + Optimized plan

**Narrow and Wide Transformation**

* Map, filter, reduceByKey, SortBy
* 50k records – 5 nodes – 10k records per node
  + Each node has certain amount of Closed, Pending, Complete, Cancelled, Processing
  + Use map which create Key, value pairs then reduceByKey to reduce them down
  + So, map is narrow transformation, reduceByKey (Groupby) is wide transformation
  + Save wide transformation to the last
* **Shuffle** – Moving of data across nodes
* Narrow transformation (no shuffle)
  + Transformation of data within that single node
  + Map, filter,
* Wide transformation (do shuffle)
  + Transformation of data across multiple nodes
  + groupByKey, reduceByKey, join
* Try to avoid data shuffling as much as possible

**Broadcast Join (vs Join) between 1 big and 1 small file**

* Suppose orders data split into 5 nodes – customers data split into 2 nodes
* Join on customerID – requires shuffle of data with similar customerID to the same node
* Broadcast Join – narrow transformation (Join – wide transformation)
  + Instead of dividing customers data to 2 nodes
  + Spark duplicate the customer file to equal numbers of node
  + And each node will have a copy of the customers file
  + Join without shuffle

**Reduce vs ReduceByKey**

* Use for grouping of values to get one result
* Reduce
  + Work on value, only on value
  + Output is a single value
  + Action – because output is single value
* ReduceByKey
  + Work on key value pairs
  + Output for each key
  + Transformation – need an action (collect()) – output is multiple, can apply more transformation onto it

**ReduceByKey vs GroupByKey**

* Wide transformations – shuffle – network i/o, more traffic – memory i/o, more memory usage
* ReduceByKey
  + It will reduce inside its own partition first
  + Then it shuffles data to aggregate values
  + Less network i/o since values are already aggregate inside its own partition
* GroupByKey
  + Instantly create shuffle by aggregating each value individually
  + Each value gets its own network i/o

**RDD vs Dataframe vs Spark SQL (Table)**

* Dataset can only be written by Java and Scala (not by Python)
* RDD
  + Data – RAM – only exist in SparkContext
  + Metadata – No (no schema, etc.….)
  + Works with both structured and unstructured
  + Low optimization
* Dataframe – RDD + metadata
  + Data – RAM
  + Metadata – Yes
  + Efficient in data processing of structured data thanks to metadata
  + Optimize Catalyst (Spark query optimizer) and Tungsten (Optimized execution engine)
* Spark SQL
  + Data – Disk – Not only store in SparkSession
  + Metadata – Yes

**Optimize Spark**

* Spark works better with
  + Evenly distributed partitions – avoid data skew
  + More partitions – better parallel processing
* Optimize shuffles (Avoid full nodes shuffles)
  + Increase parallelism (increase number of shuffle partitions) using rdd.repartitions or shuffle.partitions
  + Use repartitions or coalesce to increase partitions or reduce partitions after shuffling to optimize resources usage – to avoid full shuffle
* Optimize transformation (hold all wide transformation until the end to avoid shuffle)
  + Avoid using wide transformation, or save them until the end
  + Avoid groupByKey cause a lot of shuffling, use reduceByKey instead
* Broadcast Joins for small tables, dataset
* Salting
  + Alleviate data skew in distributed system
  + Modifying keys used in operations like “join” or “group by” by adding a random number (the “salt”) to it
* Optimize data format (Columnar data format, Parquet, ORC)
* Utilize Adaptive Query Execution (AQE)
* Prefer Dataframe over RDD
  + Use Catalyst and Tungsten

**Functions for working with RDD**

* Requires a Python Lambda function inside (except Distinct)
* Have a clear image of inputs and potential outputs
  + Map
    - Number of inputs = Number of outputs (1000 – 1000 records)
  + Reduce
    - Reduce into just one output (1000 – 1 result)
  + ReduceByKey
    - Like reduce – but the number of outputs = number of keys (100 keys – 100)
  + Filter
    - Filter based on condition – Number of inputs >= Number of outputs (100 – 90)
    - Should be used first to limited amount of data requires to work with
  + SortBy
    - Sort to rank or finding min max, number of inputs = number of outputs (100 – 100)
  + Distinct
    - Grab only distinct values – number of inputs = number of outputs

**Repartition vs Coalesce**

* Suppose a 5GB file, distributed to 40 partitions into 40 nodes in a cluster of 100 nodes
* Inefficient management of resources
* Avoid shuffle or overload data for nodes – adjust the number of partitions
* Large file size – increase number of partitions for increase processing
* Small file size – reduce number of partitions – to avoid shuffle
  + If small files are distributed to too many nodes
    - Adding overhead for I/O
    - Inefficient resources utilization
* Repartition (for increase or decrease number of partitions)
  + rdd.repartition(100, “column”) – Select specific number of partitions by random or by specific column(s)
  + When use for decreasing number of partitions also cause shuffle
  + Depends on Spark to try to create evenly distributed partitions
* Coalesce (only for reducing numbers of partitions)
  + rdd.coalesce(50) – Specify the number of partitions
  + Reduce the number of partitions without causing shuffle – just stack the data from other partitions into existing partitions
  + Can lead to uneven distribution of data across the nodes

**Cache (vs Persist)**

* Cache
  + Copy and store data entirely in RAM for faster execution
  + Reduce execution time when performing transformation
  + Default storage is RAM (cannot be changed)
* Persist
  + Same as Cache
  + But have custom for storage location
  + StorageLevel.MEMORY\_ONLY, MEMORY\_AND\_DISK, DISK\_ONLY

**Spark Read**

* Standard
  + df = spark.read.format(“csv”).option(“header”, “true”).option(“inferSchema”, “true”).load(filelocation)
* Shortcut
  + df = spark.read.csv(filelocation, header = “true”, inferSchema = “true”)
* Spark Read (spark.read)
  + Files – csv, text, json, orc, parquet
  + Database – jdbc (should not directly read from database)
  + Table – Spark table
* Spark Table <-> Spark Dataframe
  + orders.createOrReplaceTempView(“orders”)
  + df = spark.sql(“SELECT \* FROM orders”) – Create Spark Dataframe from Spark Table
  + df.show(5)

**Infer Schema in Spark**

* Dataframe – Data file + metadata
* When used on unprocessed data files, schema is more likely to corrupt
* Read with spark.read
  + inferSchema = “true”
  + Spark auto infer schema
  + When reading csv – Spark scan the entire content of the csv to understand schema
  + Large files -> Longer time to scan the entire csv file
* Resolve
  + .option(“inferSchema”, “true”).option(“SamplingRatio”, .01)
  + Scan only portion of the files to infer schema
  + More likely that Spark can wrongly infer the schema of the files
* Alternatives to automatically infer schema
  + Create a definition of schema
    - schema = “order\_ID long, order\_date date, customer\_ID long, status string”
    - df = spark.read.option(“headers”, “false”).schema(“schema”).load(filelocation)
  + Use StructType
    - import pyspark.sql.types
    - schema = StructType(StructField(“order\_ID”, LongType(), StructField(“order\_date”, DateType())))
    - df = spark.read.option(“headers”, “false”).schema(“schema”).load(filelocation)
  + Faster than infer schema – no need to full scan to auto infer schema
  + When data is not match in data types to schema provided
    - All the values in that column will be NULL

**Spark SQL Managed table and External table**

* RDD vs Dataframe vs Spark SQL
  + Data: ram, ram, disk
  + Metadata: no, yes, yes
* Spark table = Data file + Metadata
  + Managed table
    - Normal create table syntax
    - Default in location where you config the Spark Session
    - Spark manage both the metadata (schema) and the data file
    - When dropped – Spark delete both the data file and schema
    - Can SELECT, UPDATE, DELETE – since it is a table
  + External table
    - Add external before table clause (“CREATE EXTERNAL TABLE”)
    - Specify the location of the file which the table is created from
    - Spark manage only the schema
    - When dropped – Spark only delete the metadata
    - Can only SELECT – UPDATE and DELETE are not allowed (since it is the file data)
    - Databricks can solve this – with Delta table – can UPDATE and DELETE now
  + Temp table – Gone when session ended

**Spark Executors Optimization**

* Code optimization
  + Code to minimize the time for data processing
  + Reduce resources usage
    - Clusters – CPU and RAM
* Executors design
  + Thin executors
    - Fewer cores and less RAMs per executor
    - Suitable for tasks with high shuffle intensity
    - Workloads with smaller tasks that need to be processed in parallel
    - Many executors with fewer cores can handle more small tasks simultaneously
    - Advantages
      * Better parallelism
      * Efficient resources utilization
    - Disadvantages
      * Increased management overhead
      * Higher network traffic
  + Fat executors
    - Fewer executors and each executor contain more RAMs and cores
    - Suitable for workload with a few large tasks
    - Tasks that require heavy-work load
      * Process large datasets
      * Machine learning model training
      * Large scale aggregation
    - Advantages
      * Better task throughput (how many tasks can be handled over a given period)
      * Reduced shuffling overhead
      * Lower management overhead
    - Disadvantages
      * Increased garbage collection overhead (GC)
      * Limited parallelism

**Spark UI**

* Job (Spark application can have multiple jobs)
  + Jobs are made by actions – the number of jobs = the number of actions (collect, reduce, count, first, take, saveAs,
* Stage (A job can have multiple stages)
  + Stages are made by wide transformations – the number of stages = the number of wide transformations + 1 ( since always have stage 0, which is the first stage)
* Task (A stage can have multiple tasks) – Smallest unit of work
  + Tasks are made by partitions – the number of tasks = the number of partitions

**Spark Execution and Cache Optimization**

* Two ways
  + Number of partitions depending on the number of cores (4 cores = 4 partitions)
  + Number of partitions = file size / 128mb (file size of a partition)
* The more partitions for parallel processing, the faster the execution time
* Spark will choose the option that create more partitions for faster execution time
* Cache
  + df = df.cache() - (lazy transformation)
  + df = df.collect() - (action)
  + df = df.unpersist() – (release cache)
  + Cache can take quite some time and memory – should only cache data that will be used multiple times
* Serialized
  + Data that are compressed in the binary form
  + Less memory space
  + More compute time – requires unzip first and then calculation
* Deserialized
  + More memory space
  + Faster compute time

**Spark Read Mode**

* Read entire folder instead of just a file, just put in the path of the folder
* permissive
  + Ignore unmatched data type records
  + Display them as NULL
* failfast
  + Error when trying to read data frame contain unmatched data type records
  + Show the error and cannot read file
* dropmalformed
  + Automatically drop records of unmatched data types
  + Read file normally with dropped records

**Creating Dataframe in Spark**

* Method 1
  + Read from files (csv, json, xml, parquet, etc…)
  + spark.read.format(“file\_format”)
* Method 2
  + Create from a temporary table (temp view)
  + createOrReplaceTempView(“temp\_view\_name”)
  + spark.sql(“SELECT \* FROM temp\_view\_name”)
* Method 3
  + Create from a table
  + spark.table(“table\_name”)
  + Use when data comes in many files and format – data already in a table
* Method 4
  + Create a dataframe from scratch
  + spark.range(number\_of\_rows) – create a dataframe with col (id) and rows (num\_rows)
  + range() function == python range() function – range(start, end, step)
* Method 5
  + Create from lists of tuples – [(1, ‘2023-07-05’, ‘Closed’), (2, ‘2022-12-10’, ‘Pending’)]
  + spark.createDataFrame(name\_of\_list) – dataframe without proper schema
  + df.toDF(column\_names) – add column names
* Method 6
  + Create like the above method
  + Create list schema beforehand
  + spark.createDataFrame(name\_of\_list, schema\_of\_list)
  + Can also predefined data types in the schema as well
* Method 7
  + Create Dataframe from RDD – difference lies in the metadata (schema)
  + Create a RDD first
  + rdd = spark.sparkContext.textFile(“path/to/files”)
  + rdd.map(lambda x: x.split(“,”)[0], x.split(“,”)[1], x.split(“,”)[2], x.split(“,”)[3])
  + Turn the rdd into list of tuples
  + spark.createDataFrame(list\_of\_tuples, schema)
* Summarization
  + **spark.read**
  + **spark.sql**
  + **spark.table**
  + spark.range
  + spark.createDataFrame(list).toDF(schema)
  + spark.createDataFrame(list, schema)
  + spark.createDataFrame(rdd, schema)

**Nested Schema in Spark**

* Defined schema ahead
  + Schema = “customer\_id long, name struct<first\_name: string, last\_name: string>, city string”
  + df = spark.read.format(“json”).schema(Schema).load(file\_path)
  + Can also use StructType to define the schema for reading file
* The column names must match

**Date Format in Spark**

* Acceptable date format in Spark
  + yyyy-mm-dd – only accept this format
* How to handle inconsistent date format
  + spark.read.format.schema.option(“dateFormat”, “mm-dd-yyyy”).load
  + Will convert date format to yyyy-mm-dd
  + Leave the date format as string then cast the data type to date
  + From pyspark.sql.functions import to\_date
  + df = df.withColumn(“date\_col” , to\_date(“date\_col”, “mm-dd-yyyy”)
  + If provided with wrong date format, the values of the col will be NULL

**Handle duplications in Spark**

* df = df.distinct() – duplication on all columns
* df = df.dropDuplicates([columns\_needs\_for\_duplicates]) – duplication for selected columns
* Window function
  + from pyspark.sql.window import \*
  + from pyspark.sql.functions import row\_number
  + window = window.partitionBy(‘id\_col’).orderBy(‘some\_col’) – can use desc(some\_col)
  + df = df.withColumn(‘row\_number’, row\_number().over(window))
  + df = df.where(‘row\_number = 1’).drop(‘row\_number’)

**Differentiate Cluster, Node, Executor, Partition**

* Cluster
  + Combination of multiple nodes (workers)
  + Worker node for Spark, data node for Hadoop
  + Driver node for Spark, job checker for Hadoop
* Node
  + Machine inside a cluster and will be assigned job
  + Are either worker node or driver node (Spark)
  + Driver node will make sure that work is evenly distributed across the worker nodes
  + No worker nodes will have to overwork or underwork
* Executor
  + A node can contain 1 or more executors
  + Executors are created when a spark session is created
  + An environment for the worker nodes to execute the tasks
  + An executor can contain one or more tasks
* Partition
  + A task is a partition
  + Parallel processing will depend on the number of nodes and cores
  + So multiple nodes mean parallel processing, one node but multiple cores also mean parallel processing

**How Spark calculates number of partitions**

* Depends on file size and worker node configuration (ram, cpu, cores)
* 1 file
  + If the file is not splitable then 1 partition
  + If the file can be split then n\_partitions = MAX(file\_size / 128mb, number of cores)
  + Either take the file\_size / 128mb or the number of cores, which is greater
* Multi files
  + n\_partitions = MAX(num\_files / 128mb / (file\_size + 4mb, number of cores)
  + 4mb is the cost of opening a file in bytes
* Wide transformation
  + spark.sql.adaptive.enabled – Spark auto optimization of number of partitions

**Spark Deployment mode**

* The only difference between the two deployment mode lies in the driver program
* Cluster mode
  + Cluster is a combination of bunch of nodes
  + A primary node will have a driver program that initiate the SparkContext
  + The node that contains the driver program to run on it is called the driver node
  + The remaining nodes are the worker nodes that execute the tasks
  + When driver program resides in one of the nodes in the cluster – cluster mode
  + Primarily for production
    - Can setting for auto run/execution
    - Better management and optimization with complex applications
    - Better for expanding/scaling the resources for processing
* Client mode
  + Also, a cluster with nodes
  + But all the nodes in the cluster will be worker nodes
  + The driver program will be outside of the cluster and will be sent by client
  + The driver program will reside inside a local machine of the client and initiate the SparkContext
  + Primarily for development phase – small workload and easy to debug

**Ways of accessing columns in PySpark**

* df.select(“column\_name”).show(5)
  + No need for complex computation
  + Display values only
  + Create new data frame with columns from the old data frame
  + Swap orders of the columns
* df.select(df.column\_name).show(5)
  + When using Join – specifying the join on column from which df
  + Auto recommend the column names, use tab to insert them
* df.select(df[“column\_name”]).show(5)
  + When using Join – specifying the join on column from which df
  + When the column in the df has space in between
* Import required
  + from pyspark.sql.functions import \*
  + df.select(col(“column\_name”)).show(5)
    - Apply other functions on the columns – col(col\_name).alias(new\_name)
  + df.select(expr(“column\_name”)).show(5)
    - Perform calculation on entire column and access them
    - Select new columns as expression from existing columns

**Aggregate functions in Spark (PySpark)**

* Simple
  + N inputs
  + 1 output
  + Min, max, sum, avg, count
* Grouping
  + N inputs
  + < N outputs
  + Group by
* Window
  + N inputs
  + <= N outputs

**Windowing functions in Spark (PySpark)**

* Need to create a window beforehand, or identified a window first
  + Partition column (column to group by)
  + Sort column (column to order by)
  + Optional – Window size
  + My\_window = Window.partitionBy(“partition\_col”).orderBy(“sort\_col”)
  + My\_window\_2 = Window.partitionBy(“partition\_col”).orderBy(desc(“sort\_col”))
* RANK, DENSE\_RANK, ROW\_NUMBER
  + Df.withColumn(“rank”, rank().over(My\_window)).show()
  + Df.withColumn(“dense\_rank”, dense\_rank().over(My\_window)).show()
  + Df.withColumn(“row\_number”, row\_number().over(My\_window)).show()
  + Df.where(“row\_number = 1”).show()
* LEAD, LAG
  + Comparison in time series data
  + LAG
    - Retrieves the value from the previous row
    - LAG(“col\_name”, offset)
  + LEAD
    - Retrieves the value from the next row
    - LEAD(“col\_name”, offset)
  + Df.withColumn(“prev\_day”, lag(“val\_col”).over(“My\_window”))\
  + .withColumn(“diff\_val”, expr(“val\_col – prev\_day”)).show()
  + Display an extra column to show differences between two points in time

**Internal Reading vs External Reading in Spark (PySpark)**

* Workflow
  + Create a data frame
  + Transformations
  + Write back to the target
* Internal Reading – data format that have high compatibility with Spark
  + HDFS
    - Distributed File System
  + Delta Lake
    - Parquet data file
    - JSON logs file – ensuring ACID
* External Reading – remaining data formats
  + Database
  + Web API
  + Cloud Storage (S3, Azure Blob, …)
  + Diverse structures – not natively for Spark
* If data in External, use ETL tools to bring it back to Internal for optimized Spark data formats

**Writing in Spark (PySpark)**

* Workflow
  + Create a data frame
  + Transformation
  + Write back to the target
* Writing mode
  + Overwrite
    - df.write.format(“file\_format”).mode(“overwrite”).option(“path”, “location\_path”).save()
  + Ignore – ignore because records already exist
  + Append – add new records
  + errorIfExists – cause error if records already exist -> use overwrite
* When write data, can only write data as a folder – the files inside will be automatically adjusted by Spark, not as a file
* If the format is not specified, it will be parquet by default

**Optimization when writing in Spark (PySpark)**

* When writing data to files, specify the partition columns
  + .partitionBy(“partition\_col”)
* Spark automatically create partitions based on provided columns
  + The partitions will be in their separate folders
  + And Spark will remove that column in the data files
  + Assuming that all the value in that column for all the files in the same partitioned folders are the same value
  + Save resources
* Partition by multiple columns
  + Create another level of folders based on sequence of the columns specified for partitions
  + The more partition columns applied; the faster Spark will be able to located your required files
  + The less data files Spark will have to scan
  + Sort of like creating a detailed address – indexing your data for Spark to locate

**Spark GroupBy**

* GroupBy by itself is a transformation – not an action
* Actions that turn into transformations when combined with groupBy
  + Agg()
  + Count()
  + Sum()
  + Avg()/mean()
  + Max()
  + Min()
  + Pivot()
  + Apply()/mapGroups() – for Spark RDD
  + aggByKey() – for RDD
  + reduceGroups – in Datasets
  + withColumn()
  + cogroup() – pair RDD
* To create job on Spark UI, requires an action
* When wide transformation is performed, auto create 200 partitions, even though we might not use them all
  + Ineffective if we do not use them all
  + Waste of resources

**Skew partitions in Spark (PySpark)**

* Spark was unable to split data into multiple partitions for equal distribution, and therefore all data focus on only a few partitions
  + Depends on the key used for transformation

**Adaptive Query Execution (AQE) – only available at Spark 3.0 and above – automatically enabled at Spark 3.2**

* Handle two main problems of Spark
  + Unsuitable number of partitions
    - By default, Spark automatically create 200 partitions when shuffle happens
    - Despite whether the data will be equally distributed to all the partitions or not
    - Lead to unused partitions
  + Skew partitions
    - When data are not equally distributed
    - Some partitions will have significantly more data
    - Lead to skewed partitions
* Check for availability of AQE and turn on the AQE
  + spark.conf.get(“spark.sql.adaptive.enabled”, “false”) – check
  + spark.conf.set(“spark.sql.adaptive.enabled”, “true”) – turn it on
* How does AQE optimize resources usage
  + Number of records
  + Size of data
  + Min & Max of each column
  + Number of keys

**3 Join techniques in PySpark – Distributed strategy (Broadcast/Shuffle) + Join Type (Hash/Sort Merge/Nested)**

* Broadcast Hash Join (between one small table and one large table)
  + Broadcast join
  + But the smaller table (smaller than 10 mb) will be turned into a full hash table
  + Reduce search time, but increase memory consumption
* Shuffle Soft Merge Join (between 2 tables)
  + Shuffle the data
  + Then sort them for better matching
  + Slower in terms of search time in comparison to hash table
* Shuffle Hash Join (between 2 tables)
  + Shuffle the data
  + The partition of one table on that node will be turned into a full hash table
  + Reduce search time, but increase memory consumption

**Join two large tables in Spark (PySpark)**

* BucketBy
  + Faster and more efficient than Broadcast Hash Join
  + Can be used for when both tables are large
  + Used a hash function to bucketing the value – no need to shuffle
  + Can be applied to GroupBy, etc.…

**PartitionBy vs BucketBy**

* PartitionBy
  + Use to partition data based on one or multiple columns
  + Useful for columns with finite number of distinct values
  + Better for querying, only read data from searched partitions
  + Save data as folders – representing partitions
* BucketBy – can also be used with sortBy
  + Specify how many buckets you want the data to be distributed into
  + Suitable for columns that have almost indefinite number of distinct values
  + Distributed data based on the columns specified and a hash function
  + Save data as files (inside one large folder) – representing buckets
  + Required to save as table – will also save metadata

**Memory management in Spark (PySpark)**

* Heap memory (JVM)
  + Manage by JVM – Java Virtual Machine
    - Frequent scanning for trash to clean up memory
      * Can cause stopping of application
      * Reduce performance
  + Components
    - Reserved memory – default 300MB
      * For critical operations of JVM
      * Ensure stability
    - User memory – default 40% - can change
      * Memory for user – Python (not Spark code)
      * spark.memory.fraction for change
    - Unified memory – default 60% - can change
      * Execution memory – 50% – for execution
      * Storage memory – 50% – for storage
      * spark.memory.storageFraction for change distribution
* Over Head/ Off-Heap memory – Max 384MB – 10% of Heap memory
  + Stand independently to support the process of Heap memory
  + Manually scanning for trash – trigger by user
* Adjust the memory distribution accordingly to usage and requirements

**Logical and Physical plan in Spark (PySpark)**

* Logical plan will come first, then Physical plan (Logical plan -> Physical plan)
* Logical plan (3 steps)
  + Parsed logical plan
  + Analyzed logical plan
  + Optimized logical plan – Use Catalyst Optimizer
    - Rule 1: Prioritize filter data before further execution – where clause
    - Rule 2: Try to join all the select statement into just 1 select – even when use sub queries
    - Rule 3: Group all the filter into just 1 filter for filter data – where clause
* Physical plan
  + Can have multiple physical plans, based on logical plan
    - Join techniques, etc
  + Then use cost model to calculate least cost and choose the physical plan that cost the least
  + Then convert the code into RDD for actual execution

**File Format**

* How data is organized and stored in a file for a computer to read and process it
* Each file format will have different structures and identified by the tails
* Different file formats
  + Efficient storage of data
  + Improve the data processing speed
  + Spend less time for I/O
* Functionality of file formats
  + Faster reads
  + Faster writes
  + Can be distributed (for Big Data) for parallel processing
  + Support of compress techniques (zip, gzip, snappy, …)
* Primary data storage of files
  + Row based
    - Insert – Better optimized for row based
    - Select – When selecting certain fields, row based will return an entire row
    - Compression – Worse in compare to Column based because data stored in adjacent may be of different data types
    - Common file types : csv, txt
  + Column based
    - Insert – not optimized for column based
    - Select – Scan only selected fields and retrieve them, better optimized for select
    - Compression – Easier compression cause all the data already stored in adjacent
    - Common file types : parquet, orc, avro
* Most optimized for Spark is Parquet – distributable, have schema, etc…

**Why Parquet is suitable for Spark**

* Size
  + A parquet file is often lightweight
  + Compared to CSV, JSON (10 times lighter)
* Distributable (Có thể chia nhỏ)
  + Suitable for Spark – Distributed computing
* Column based
  + Store data by column
    - Data points next to each other belongs to the same column
  + Still have some characteristics of row-based
    - Data inside are stored as row groups
    - Each row group store data in column
  + Sort of hybrid format
* Schema evolution
  + Has metadata, structured
  + Adding new column – Drop existing column – Changing data type
    - Metadata auto update
  + When reading files with change in schema
    - Add .option(“mergeSchema”, True)
      * New column – existing row values will be Null
      * Drop existing column – nothing changes

**Cluster in Databricks**

* Compute tab in Databricks – Clusters
  + All-purpose cluster
  + Job cluster
* Create cluster
  + Specify Databricks runtime version (LTS – Long time support)
  + Specify Spark config
  + Set Environment Variables
  + Set number of workers
  + Set the Driver type
* All-purpose cluster
  + Create manually
  + Can use anytime – terminate
    - When not in use can shut the cluster down
    - Or set the timer to automatically shut down
  + Useful for dev environment
  + Can be shared across multiple users
  + High cost – for dev
* Job cluster
  + Create automatically – JSON file which contains configurations
  + Only used when the job starts and will turn off when the job end
  + Suitable for automation in the production environment
  + For specific job only
  + Low cost – since less time usage and low availability

**Notebooks in Databricks**

**DBFS – Databricks File System**

* %fs – magic command in the notebook, used for watching the files inside DBFS
* Can use Catalog tab to access the DBFS
  + But not by default
  + Requires adjustments in Settings, Advanced part – (DBFS File Browser)
* DBFS – contain FileStore – Databricks (just Spark on Cloud)
  + Virtual File System (Hệ thống tệp ảo)
  + Built by Databricks using Cloud Storage (AWS, Azure, GCP)
  + For managing data on Databricks – no need to directly interact with the Cloud Storage
* How does Databricks store, compute – with Community Edition (free edition)
  + Since Cloud services cost
  + Go to Compute tab
    - The three dot on the top right
    - View JSON
    - See the detailed configurations
      * Default would be by AWS – without paying so no direct access
      * Everything managed by Databricks

**dbutils in Databricks**

* Notebooks
  + Run dbutils.help() – for all commands
  + For further detailed commands – run dbutils.{commands}.help() – dbutils.fs.help()
    - Can incorporate dbutils.fs.help({method\_name}) for even further details
    - dbutils.fs.help(“ls”) – list all the contents of a directory
    - dbutils.fs.ls(“/”) – all the files – or use magic command (%fs – ls /File)
      * Magic commands will provide better visual display
    - Use display to wrap around – for better visual representation
    - Can even display the contents of the files

**Data Lake – Strengths and Weaknesses**

* Strengths
  + Can store all types of data
    - Structured
    - Semi-structured
    - Unstructured
  + Reasonable cost
  + Dynamic expand/narrowed capabilities depending on requirements
* Weaknesses
  + Cannot guarantee ACID
    - Atomicity
    - Consistency
    - Isolation
    - Durability
* Situations
  + A job failed while appending data
    - 5 csv files already appended
    - 4 more csv files needed to be appended
    - Failed when appending 2 files
    - Violates ACID – No proper handle of the situation
    - With ACID – the transaction will be roll back (status before job)
  + A job failed while overwriting data
    - Delete what is already there
    - Write new data
      * Failed while writing new data
    - With ACID – the transaction will be roll back (status before job)
  + Simultaneous read – write
    - With ACID – Lock transactions for read, write (ensure consistency and isolation)
* Delta Lake
  + Have ACID

**Delta Lake**

* Resolve common problems of traditional **Data Lake**
  + ACID transactions
    - No consistency
    - No reliability
  + Better than the normal **Parquet** format
    - Df.write.format(“parquet”)
    - Df.write.format(“delta”)
  + Delta = Parquet + Transaction Logs
    - Data files – parquet files
    - Transaction Logs – delta logs – json files
  + Append mode
    - Df.write.format(“parquet”).mode(“append”)
      * More parquet files
      * Create new parquet files
    - Df.write.format(“delta”).mode(“append”)
      * Still more parquet files
      * And another transaction logs (json file)
      * Detailed write steps
        + Write data to parquet files
        + Then write logs to json file
        + If write failed midway – no new json file be created
      * Detailed read steps
        + Read from transaction logs first – json file
        + Then read data in parquet files