**CH-7 Spark Pipelines:** A simple pipeline, which acts as an estimator. A Pipeline consists of a sequence of stages, each of which is either an Estimator or a Transformer. When Pipeline.fit() is called, the stages are executed in order. If a stage is an Estimator, its Estimator.fit() method will be called on the input dataset to fit a model. If a stage is a Transformer, its Transformer.transform() method will be called to produce the dataset for the next stage. The fitted model from a Pipeline is a PipelineModel. **Pipeline Components = Transformer:** A Transformer is an abstraction that includes feature transformers and learned models. Technically, a Transformer implements a method transform(), which converts one DataFrame into another. Eg: Tokenizer, StopWordsRemover, n-gram, PCA, StringIndexer, OneHotEncoder, VectorAssembler. **Estimator:** An Estimator abstracts the concept of a learning algorithm or any algorithm that fits or trains on data. Technically, an Estimator implements a method fit(), which accepts a DataFrame and produces a Model, which is a Transformer. For example, a learning algorithm such as LogisticRegression is an Estimator, and calling fit() trains a LogisticRegressionModel, which is a Model and hence a Transformer. A Pipeline chains multiple Transformers and Estimators together to form a **ML Workflow.** PipelineModel produced training phase is used for making predictions in the test phase. Best set of hyperparams for ML model using **CrossValidator – TrainValidationSplit.** ParamTuning using **ParamGridBuilder.**

**Associative Analysis & Frequent Pattern Mining: Support:** Proportion of transactions contain itemset, NoOfTr contain I1 / Total Tr. **Confidence:** Conf of rule A -> B measures how often rule is true. Conf(A->B) = Support (ItemA AND ItemB) / Support(ItemA). **Apriori Algorithm** used to mine assoc rules from large DS. FP pipeline: **fp\_basket = FPGrowth(itemCol=’items’, minSupport=0.2, minConfidence=0.5) 🡪 RM\_model = fp\_basket.fit(df) 🡪 RM\_model.freqItemsets** and **RM\_model.associationRules.** Concept of Recommender System: Recommender systems play a pivotal role in suggesting relevant items to users. Context-based recommenders factor in user context, enhancing precision. Collaborative filtering predicts preferences through user-item interactions, employing a neighbourhood approach that draws on similar users or items. Matrix factorization, a latent factor model, uncovers underlying preferences from complex data. However, the **cold start problem arises with limited initial data**, impacting recommendation accuracy. Spark.ml uses **AlternatingLeastSquares – (numBlocks, rank, maxIter, regParam, implicitPrefs, alpha)** Increasing maxIter may allow the model to converge to a more accurate solution, but it can also lead to longer training times and potential overfitting if set too high. A **higher regParam value increases the strength of regularization**, which can lead to a simpler model that is **less prone to overfitting**. However, setting it too high may result in underfitting, reducing recommendation accuracy.

**CH-8 Spark GraphFrames:** Work on DataFrames in Spark to represent graph structure: vertex(users) and edges(reln b users). **Vertices**: Vertices represent the nodes or entities in a graph. Each vertex typically has an identifier and can contain additional properties or attributes. **Edges:** Edges represent the relationships or connections between vertices. They have a source vertex, a destination vertex, and can also have properties. **Pagerank:** PageRank is an algorithm that measures the importance of vertices in a graph. It assigns a numerical score to each vertex based on the number and quality of incoming links. **Triangle count** is the number of triangles formed by three vertices in a graph. A triangle is a set of three vertices connected by three edges. To create a graph, you need two DataFrames - one for vertices and another for edges. **page\_rank\_df = graph.pageRank(resetProbability=0.15, tol=0.01) 🡪** page\_rank\_df.vertices.select("name", "pagerank").show(). Conn\_df = GraphFrame.connectedComponents(). GraphFrame.traingleCount().show(). GraphFrame motif finding uses a simple Domain-Specific Language (DSL) for expressing structural queries. For example, graph.find("(a)-[e]->(b); (b)-[e2]->(a)") will search for pairs of vertices a,b connected by edges in both directions. It will return a DataFrame of all such structures in the graph, with columns for each of the named elements (vertices or edges) in the motif. **BreadthFirstSearch: g.bfs("name='Esther'", "age<32", edgeFilter="relationship != 'friend'", maxPathLength=3).** results = g.shortestPaths(landmarks=["a", "d"]) Computes shortest paths from each vertex to the given set of landmark vertices, where landmarks are specified by vertex ID. NoOFTriplets: g.triplets.count()

**CH-9 Structured Streaming:** provides fast, scalable,built on the spark SQL engine, you can use dataframe API, aggregation , event time wondows, stream to batch join can be performed, fault-tolerant*, end-to-end exactly-once stream processing* Structured Streaming queries are processed using a micro-batch processing engine end-to-end latencies as low as 100 milliseconds and exactly-once fault-tolerance guarantees. outputMode: Complete, Append, Update. Spark runs it as an incremental query on the unbounded input table. Every trigger interval (say, every 1 second), new rows get appended to the Input Table, which eventually updates the Result Table. Event Time: Time of event origin at source, Arrival Time: Time at which data comes to Unbounded Table. Watermarking: Allows users to specify threshold of late data – dictates how long processing eng wait for late-data before result op. streaming data or not by using df.isStreaming. **watermarking**, which lets the engine automatically track the current event time in the data and attempt to clean up old state accordingly.| What are the guarantees of Structured Streaming? Streaming data can be expressed in the same way as batch data |Using continuous processing, it can be process end-to-end data in times as low as 1 millisecond |End-to-end exactly-once processing |Fault Tolerance Q) spark.readsteam :- kafka, file source, socket. Q) Structured Streaming from file based sources requires you to specify the schema

**CH-10 Apache Hive:** DWH based on Hadoop, batch-processing, analyze large historical data, has MR capabilities. **Properties:** High-latency query response due to HDFS, no OLTP – no real-time data access. Not 100% available – HDFS NameNode bottleneck master-slave architecture and no ACID compliance. Hive supports **Scaling Out / Horizontal Scaling:** increase CP add commodity servers. Data Units in HIVE: Databases 🡪 Tables 🡪Partitions🡪Bucket/Cluster🡪FileBlock. **Complex DataType: Struct, Map, Arrays.** It can link to files stored in HDFS or import files from local filesystem. mechanism to impose structure. Partition Keys determine how Data physically stored. By def, **table stored in: /user/hive/warehouse directory on HDFS. Metastore –** component stores structure information of the various tables and partitions in the warehouse including column and column type information, the serializers and deserializers necessary to read and write data and the corresponding HDFS files where the data is stored. **Data Abstraction & Discovery.** Buckets are specified by "CLUSTERED BY" clause after specifying the "PARTITIONED BY" clause. Bucket stored as file in partition dir. When the user creates a table by default, it is an internal table. HIVE for creating DW+ETL app using Big Data. The result of HIVE query can be stored as Another table, HDFS file, storing data in table and download the table. Possible execution engines in HVE :- MapReduce, Tez, Spark **CASACADE is use to drop database and it’s associated tables.**

**CH-11 No-SQL:** Horizontal Scaling / Scaling Out, Auto-Provisioning, HighPerf, Availability, Fault-Tolerance, Inexpensive, Schema Flexibility-Evoluition, handle HV data. **Data Sharding:** Horizontal Partitioning of huge Datasets across multiple DB instances or node. Determined by hash fn or logical divisions. Helps in making large db manageable. **Not Good: Structured Data, Real-time OLTP, ACID transactions, Complex Joins and Aggregations.** ACID no longer desired, sw to **BASE: Basically Available, Soft-State, Eventual Consistency.** Soft-state means copies of dataitems spread across multiple nodes may be inconsistent. **Brewer’s CAP Theorem: For any system sharing data it is impossible to guarantee simultaneously all of these three properties: Consistency:** Every read receives the most recent write ie all data copies reflect same state. **Availability:** system remains responsive and accessible even in the presence of network failures or node crashes. **Partition Tolerance:** ensures that write and read operations are redirected to

available replicas when segments of the network become disconnected. **Categories NoSQL:** key-value stores (SimpleDB, DynamoDB, Redis, Riak), document-stores(MongoDB,CouchDB,Couchbase), wide-column stores(BigTable,HBase,Cassandra), graph database(Neo4j, AmazonNeptune), XML database.

**CH-12 Mongo DB:** Stores objects like JSON, BSON ; k-v pairs: Dictionaries, HashMaps, Associative Arrays. **Documents (Records/Rows) stored in Collections (Tables).** To select a database: **use <dbname>** Creating database and collection automatically using insertOne(): **db.mycollection.insertOne({ x:1 })** Explicit Creation: **db.createCollection()**. Collections are assigned immutable UUID which remains same across all members of replica set and shards in cluster. Collections require NO SCHEMA. **CRUD Ops: Create:** db.users.insertOne( { name:’Sue’, age:19, status:’pending’}) **Read:** db.users.find( { age: {$gt : 18}}, {name:1, addr:1}).limit(5) **Update:** db.users.updateMany( { age: {$lt : 18}}, {$set : {status:’reject’}}) **Delete:** db.users.deleteMany({ status:’reject’})

**CH-13 HBase:** A Distributed wide-column store built on HDFS. Derived from Google’s Big Table. **Provides quick random access to huge amts of structured data.** Low latency , semi structured , internally uses hash tables and random access. Emphasizes on **Consistency + Partition Tolerance. Schema-less, non-transactional, no ACID compliance.** It is Horizontally(linearly) scalable across various nodes, HBase provides automatic failure support between Region Servers. Tables: Data is stored in a table format in HBase. Row Key: Row keys are used to search records which make searches fast. Column Families: Various columns are combined in a column family. These column families are stored together which makes the searching process faster because data belonging to same column family can be accessed together in a single seek. Column Qualifiers: Each column’s name is known as its column qualifier. Timestamp is a combination of date and time. **Accessing order for cell value: (RowKey,ColFamily,ColQualifier,Version)** HBase Architecture: HBase tables are divided into regions 🡪 Each region is served by a RegionServer 🡪 RegionServer serves data for reads and writes 🡪 The RegionServers are coordinated by a HMaster 🡪 Master also assigns regions, detects failures of RegionServers. To manage master election and server availability we use zookeeper. **CRUD ops: Put =** Inserts data into rows both add & update, **Get =** Access to single row, **Scan =** Access data from range of rows, **Delete =** delete row/range. **Sparse and Distributed + Multidimensional Sorted Map + ALL.** Hbase does not support traditional joins.

**CH-14 Cassandra:** Cassandra(fault tolerant,scalability,Linearly Scalable,high availability) Cluster->Data Centres->Racks->Nodes. **Massively Linearly Scalable,** fully distributed with no single point of failure, due to horizontal scaling horizontal scaling: add commodity hardware to the cluster , vertical scaling: add RAM and CPU to specialized high performance box. Cassandra does require some sort of schema to adhere. Emphasizes **Availability + Partition-Tolerance.** RLDA. **Cluster:** Nodes join a cluster based on the configuration of their own conf/cassandra.yaml file, key settings: cluster\_name=to logically diff set of nodes, seeds=IPaddr of initnode for new node to contact discover topology, listen\_address=IPaddr of node. **Coordinator:** The node chosen by the client to receive a particular read or write request to its cluster. The coordinator manages the Replication Factor (RF). Range: 1 to TotalNodesinCluster. The coordinator also applies the Consistency Level (CL) - how many nodes must acknowledge a read or write request. **Possible Consistency Levels: ANY, ONE, QUORUM, ALL.** Data is stored on nodes in partitions, each identified by a unique token. Partition – a storage location on a node (analogous to a "table row"). Token – integer value generated by a hashing algorithm, identifying a partition's location within a cluster. The 2^64 value token range for a cluster is used as a single ring. Partitioner hashes tokens from designated values in rows. **Partitioner configured in Cassandra.yaml – Murmur3Partitioner (64 bit hash)**default. **Replicas:** All partitions are "replicas", there are no "originals", First replica – placed on the node owning its token's primary range, Closest node – replicas placed in same rack and data center, if possible. **Stale-Read Problem: Read-CL + Write-CL < Replication Factor. QUORUM CL avoids stale reads and provides balance b/w Availability & Consistency.** Default CL=1. As a node joins a cluster, it GOSSIP PROTOCOL with the seed nodes set in its cassandra.yaml to learn its cluster's topology. It is legal to partition on more than one cols of dataset. Clustering key column provides an efficient way to lookup. **Snitch: Determines request goes to which node.** PK = (PartitionKey,ClusteringKey), Peer-to-Peer, all nodes accept R-W requests. Hash value = 64. Cassendra architecture :- all nodes accepts read and write, peer to peer, any one node can serve as client request. Cassandra uses a component known as the snitch to direct read and write operations to the appropriate nodes. When a request is sent to a cluster, who determines which node to forward that request to it’s snitch. primary key in a Cassandra data model is composed of (partition key and clustering key).

Structured Streaming requires Schema to be specified. Socket-Kafka-FileSource for readStream.

**HDFS Block Size: 128 MB** Blocks are large to minimize the cost of seeks which incur more overhead. Transfer time increases but HDFS is suited for write-1-read-many. Less singular edits/update ops, mainly used for analysis of Big Data. Major checkpoints in fsImage and incremental changes in EditLog. Secondary NN **merges FSimage+EditLog(Checkpointing)** and this merged fsimage is sent back to the NN at intervals. **NN Startup:** NN loads fsImage into memory-> loads editLog -> replays journaled changes to update block metadata. **Small Reg.coeff -> Overfitting. Large Reg.coeff -> Underfitting. Transformations:** map, flatMap, MapPartition, Filter, Sample, Union, ReduceByKey, GroupByKey, sortByKey, Join, Cartesian, Distinct, Intersection. **Actions:** count, collect, take, top, countByValue, reduce, fold, lookupKey, aggregate, foreach. # All transformations in Spark are lazy.

statement of CAP theorem For a distributed system, it is impossible to achieve all three of CAP properties at the same time. | neo4j is a graph database |

**Questions :-** Not a reason that **NOSQL** become solution for organization (Improved ability to keep data consistent) |**difference** HIVE and HBASE (HIVE is slow ) | spark reconstruct lost partition (using lineage information) |true about **dataframes** in spark (all except reading data from file necessary to mentions data type) |true about spark structured streaming concept (while reading, structured streaming, \*If a static query is of the form) | benefits OF **IMPALA** (read and work, supports real time, SQL like syntax) | Hadoop cluster resource manager is (**YARN**) | **Cassandra** which is used for spread network topology (gossip) | **cassendra** what causes stale ( read-1 write-1 RF-3) | advantage of creating partition in HIVE (faster query performance) | bread milk CI (3/4,1.0,2/4) | CA = MySQL, PostgreSQL, MariaDB. AP = Cassandra, SimpleDB, CouchDB, Dynamo. CP = HBase, MongoDB, Redis, BigTable | graphframe in spark correct choice ( all 6) | true about HBASE ( only define column families, indexed and sorted rows) | which is distributed column oriented ( **HBASE**) | tow tab separated files employee and sarary ( use inner join) | **mongoDB** which of the following it accompolish (adds a document , created a collection) | you have 2 hive tables ( The customers and orders table) |True regarding regions in HBASE (region can be spread across, manage by region servers, \*represents an index) | **motif** (there would be one pattern matching the criteria) | true descriptions of data model of HBASE ( sparse, multi dimentional , each cell contains multiple values) | true about pipelines in spark ( all-> no matter, by default, by default, the learned) |

True about columns in Cassandra ( to retrieve a column value, the column key in row) | How client reads file from HDFS (The client queries the NameNode) | false about HIVE (HIVE data must be copied) | arrange Hadoop data technologies (Impala > sparkSQL > HBASE > HIVE) | true about mongodb ( all except that schema must be known before)

| true about spark structured streaming approach (there are 3 ways ,stream data written to unbounded table, after processing) | bread milk (support 40%, confidence 66.6)

| true about **impala** (all except impala has fixed coordinator) | true regarding partition and clustering column in **cassendra** (it is legal to partition, clustering key column provides , It is good idea ) | if you create data warehouse which technology will be used (HIVE).

**Questions** :- TRUE about columns in **cassendra** data model (to retrive column value, column key in row are sorted) discover that in your live in HDFS cluster ( increase block size on the DATA NODE) | minimum number of parameters that K means algo requi red (value of K) | Three service deamons that are part of **IMPALA** (all except process deamon) | You have two tab separated files - employee and salary (Select ename, s.salary from employee e inner join salary s on e.empid = s.empid , mployee.join(salary).orderBy(x >= -x.\_2).take(5) ) | You have two Hive tables - customers and orders with the following schema (The customers and orders tables are joined on the customeriD field) |

Which of the following lines following code would return the id and average age for the persons whose post each user likes. oT clarify. fi 1likes 2, 3, and 4s' posts, then we want to compute the average age of 2, 3, and 4 (edges.join(vrtices,edges(“dst”)==vertices(“id”) ) | Finally Icreate a Graph as below: val g = GraphFrame(v, e)

Select the correct choices below: ( I would like to perform triangle count around each node. Remember that triangle count is not directional. I would like the result sorted by the descending order of triangles around each node.va l r e s u l t s - g. t r i a n gl e Co u n t 7 u n 0 , results.select("id", "count").order By(desc("count\*).show)) |

Queries :- **create a dataframe with 2 columns** : person name and total number of likes received (total likes = edges.groupBY(“dst”).sum(“likes”).toDF(“dst”,”likes”) val result=totalLikes.join(vertices,totaLikes(“dst”)==vertices(“id”)).select(“name”,”likes”)impala

Confusion Matrix: Accuracy = (TP + TN)/All, Sensitivity = TP/P, Precision = TP/(TP+FP), Recall = TP/(TP+FN), F = 2\*Prec\*Rec/Prec+Rec Min params for **KMeans NOTHING REQUIRED . SELECT a.yeara.player longest one.**

# Flights between DFW and LAS from pyspark.sql.functions import desc airportGraph.edges.where("src = 'DFW' AND dst = 'LAS'").groupBy("src", "dst").count()

# find indegrees and outdegrees inDeg = airportGraph.inDegrees | inDeg.orderBy(desc("inDegree")).show(5, truncate = False)

# ranks = airportGraph.pageRank(resetProbability=0.15, maxIter=10) | ranks.vertices.orderBy(desc("pagerank")).select("id", "pagerank").show(10)

#stationGraph.bfs(fromExpr="id = 'Townsend at 7th'", toExpr="id = 'Spear at Folsom'", maxPathLength=2).show(10) | cc.where("component != 0").show() | scc = minGraph.stronglyConnectedComponents(maxIter=3) | stationVertices = bikeStations.withColumnRenamed("name", "id").distinct() tripEdges = tripData\ .withColumnRenamed("Start Station", "src")\

.withColumnRenamed("End Station", "dst")

HIVE :- spark.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) USING hive") | spark.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src") |

stringsDS = sqlDF.rdd.map(lambda row: "Key: %d, Value: %s" % (row.key, row.value)) | for record in stringsDS.collect(): |print(record). | spark.sql("SELECT \* FROM records r JOIN src s ON r.key = s.key").show(). |

cyclic rank by year ->

select \* from cyclic\_rank\_by\_year\_and \_name where race\_year=2014; | select \* from cyclic\_rank\_by\_year\_and \_name where race\_year=2014 and race\_name like “$4th tour”;

A paper with text and symbols

Description automatically generatedA piece of paper with writing on it

Description automatically generatedORDER BY RATINGS