**CH-1-Big Data:** 5 V’s: **Volume** (huge size- 40000 Exabytes of data), **Variety** (Structured, Semi-structured & Unstructured: RDBMS,csv,xlsx,avro,orc,parquet,text,audio,video,social networks), **Velocity**(speed of data in and out), **Veracity**(refers to the quality and accuracy of data), **Value**(something valuable to extract Information). Kilo Byte: 10^3, Mega Byte: 10^6, Giga Byte: 10^9, Tera Byte: 10^12, Peta Byte: 10^15, Exa Byte: 10^18, Zetta Byte: 10^21. **OLTP (RDBMS) -> OLAP (Data Warehouse) -> RTAP (Big Data Architecture & Tech)**. Big Data is more real-time in nature than traditional DW applications. Traditional DW architectures like Teradata, Redshift, Exadata, Hive-not suited for RT BD apps. Shared nothing, massive parallel-processing and scaled-out architectures(in scale out we can add more commodity hardware). BD use: DW – OLAP, Pattern-matching XML, RDF (Semantic Webpage), Knowledge Discovery: Data Mining, Statistical Modelling. File in HDFS access through HDFS:// hdfs configuration managed through xml files

Hdfs popular :- fault tolerance, unstructured data, scalability, low commodity hardware.

**CH-2-HDFS:** DFS provides orgs with a **scalable system** (new datanodes-workers can be added) to manage huge amounts of a variety of data remotely. Companies save lot of money by using legacy storage devices/**commodity hardware** and DFS also improves availability of data through replication across nodes. DFS features: **Mounted on multiple servers**-transparent to client, the entire multi-nodal system must appear as a homogeneous structure to client – **single consistent namespace, Fault & Error tolerant, Remote Access, ability to manipulate Big Data**. #HDFS Design: *Very Large files*: Hadoop clusters running today store petabytes of data. *Streaming Data Access*: HDFS is built around write-once, read-many times pattern. *Commodity Hardware*: Hadoop doesn’t require expensive reliable hardware, designed to run on cheaper Linux servers for which the chance of node failure across the cluster is high, at least for larger clusters. HDFS designed to continue working in such failure – Fault Tolerance. **HDFS not a good fit:** \*Low-latency data access: HDFS won’t work well for apps req latency in milisec. It is designed to deliver a high throughput of data HDFS has high latency. \*Lots-of-small-files: Since the masternode stores filesystem metadata, the limit to the number of files is governed by the memory of the namenode. \*Multiple-writers, arbitrary file mods. **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. **Moving Computation:** The data is processed where it is stored-locality optimization, which saves time-cost of moving data and bandwidth. **HDFS Assumptions & Goals:** # Hardware Failure: HDFS instance consists of hundreds/thousands of servers (datanodes-commodity hardware) having non-trivial probability of failure, some component of HDFS always non-functional. Detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS. # Streaming Data: HDFS is designed for batch-processing rather than interactive use by users, emphasis on processing huge amts of data (TBs) over low latency data access. # Moving Computation cheaper than Moving Data: App executed near the data it operates, minimizing network congestion and improving the overall throughput of the system. **HDFSArch:** HDFS has master-slave architecture, one NameNode(master) managing file-system namespace and regulating file-access by clients, multiple DataNodes(workers) which manage and compute data (blocks) stored on them. UserData split into blocks are often replicated across the DataNodes **(replication factor = 3)** to reduce Data Loss. **NameNode:** Centrepiece of HDFS, primary task is to manage all the metadata(in-memory): consists of file name, file path, number of blocks, block Ids, replication level. All information regarding the logs of the transactions happening in a Hadoop cluster (when or who read/wrote the data) will be stored in MetaData. It also keeps info on state of worker nodes. As Namenode works on the Master System, the Master system should have good processing power and more RAM than Slaves. **DataNode:** responsible for processing all userdata, this is where actual computation takes place, store blocks, delete blocks and replicate those blocks upon instructions from the NameNode. Also sends blockreport to NN. **Secondary NN:** used for taking the hourly backup of the data. In case the Hadoop cluster fails, or crashes, the SecNN will take the hourly backup or checkpoints of that data and store this data into a file name fsimage. EditLog– All the file write operations done by client applications are first recorded in the EditLog. 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.

**CH-3-Hadoop MapReduce:** MR is the processing engine of HDFS, helps achieve concept of “moving computation” rather than “moving data” => Locality of Computation. MR makes efficient use of multiple nodes in the cluster-parallel computation. **MR Design** – vast amount of data, parallel processing, large clusters of cheap commodity hardware, reliable, fault-tolerant, should be able to increase processing power by adding more DataNodes **(Scaling-out)** without disturbing current cluster **(Hot-Swap)** , **sharing data or processing between nodes is bad – ideally want shared-nothing architecture**, batch-process(bulk trans only) no random seeks. MR – origins in Functional Programming – Map&Reduce are higher-order functions – apply lower-order fx trans on sequence data(lists). Map applies transf and returns (k,v) pair data, Reduce takes (k,v) return a scalar val. Once all MAP tasks are done, only then can REDUCE tasks be executed to aggregate data based on ‘key’. Inp DataBlock-1 map task identifies useful-key. 1 worker node can work on multiple-map tasks. **Reduce will work on a PER-KEY basis.** Intermediate output gen by Mappers is **Shuffled (all k,v grouped together based on similar KEY)** and then **Sorted** before given input to Reducers which aggregates based on KEY.

**Hadoop 1.0:** supports only MapReduce-based Batch/Data Processing Applications. No support for RL Data Proc. Two daemons: Job-tracker and Task-tracker. Job tracker is the single point of failure which has to perform multiple activities such as Resource Management, Job Scheduling, and Job Monitoring, etc. To overcome these, **Hadoop 2.0:** Apache Hadoop 2.0 represents a generational shift in the architecture of Apache Hadoop. With YARN, Apache Hadoop is recast as a significantly more powerful platform – one that takes Hadoop beyond merely batch applications to taking its **position as a ‘data operating system’ where HDFS is the file system and YARN is the operating system**. **Processing Engines:** MapReduce (batch-proc), Tez(interactive), Spark(in-memory), Storm(Streaming), Dask, Ray, Giraph etc. The fundamental idea of YARN is to split up the two major responsibilities of the JobTracker and TaskTracker into separate entities: **Resource Manager and Node Manager** – separate daemons.

**Resource Manager:** The ResourceManager is the ultimate authority that arbitrates resources among all the applications in the system. The ResourceManager has two main components: **Scheduler** and **ApplicationsManager**. The #Scheduler is responsible for allocating resources to the various running applications subject to familiar constraints of capacities, queues etc. **Scheduler allocates resources based on the idea of container** which incorporates elements such as memory, cpu, disk, network etc. Scheduling policies can be **FIFO scheduler, CapacityScheduler and the FairScheduler.** It is a pure scheduler, means it does not perform other tasks such as monitoring or tracking and does not guarantee a restart if a task fails. #Application Manager: It is responsible for accepting the job-submissions and negotiating the first container (AppMstr) from the resource manager. It also restarts the Application Master container if a task fails. **Node Manager:** is the per-machine framework agent responsible for containers, monitoring resource usage (cpu, memory, disk, network) and reporting the same to the ResourceManager/Scheduler. **It registers with the Resource Manager and sends heartbeats with the health status of the node**. It monitors resource usage, performs log management, and kills a container based on directions from the resource manager. It is also responsible for creating the container process and start it on the request of Application master. **Application Master:** The per-application ApplicationMaster has the responsibility of negotiating appropriate resource containers from the Scheduler, tracking their status and monitoring for progress. The application master **requests the container from the node manager by sending a Container Launch Context(CLC**) which includes everything an application needs to run. Once the application is started, it **sends the health report** to the resource manager from time-to-time. **Container:** It is a collection of physical resources such as RAM, CPU cores and disk on a single node. The containers are invoked by Container Launch Context(CLC) which is a record that contains information such as environment variables, security tokens, dependencies etc. **MR Processing:** Input and final Output are stored on DFS, **Scheduler tries to schedule map-tasks close to physical storage location of data. Intermediate results are stored on local FS of Map and Reduce workers.** How many M-R jobs? M map tasks, R reducer nodes. Make M much larger than the number of nodes in the cluster. **Improves dynamic load balancing and speeds up recovery from worker failures. Usually R is smaller than M.** #Map-worker failure: Map tasks completed or in-progress at worker are reset to idle, Reduce workers are notified when task is rescheduled on another worker. #Reducer-failure: Only in-progress tasks are reset to idle, Reduce task is restarted. **Master failure: MR task aborted completely.**

**CH-4-Apache Spark:** in-memory computing & data-transfer, limit disk I/O jobs, can be run on any datasource: HDFS, Hive, S3. Spark has 4 major components: Spark SQL, Spark MLlib, Spark GraphX and Spark Streaming. SparkContext object (SC) represents the connection to a Spark cluster, and can be used to create RDDs, accumulator(to perform counter and sum efficiently) and broadcast(to gove everynode a copy of large input dataset) variables on that cluster. RDD- fault-tolerant in-memory collection of elements that can be operated on in parallel, created using parallelize() or sc.textFile(). RDDs support two-types of ops: **Transformations and Actions. Transformations:** which create a new dataset from an existing one, **Actions:** which return a value to the driver program after running a computation on the dataset. **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, in that they do not compute their results right away. The transformations are only computed when an action requires a result to be returned to the driver program. By def, each transformation is recomputed when action is called – to make it persist in-memory use the cache() method. RDD – immutable collection – partitioned using Hash/Range partition – operated on bulk-transf. Inexpensive fault tolerance (log lineage rather than replicating/checkpointing data). **Driver Program accesses Spark** using sc object-> requests cluster manager (YARN, Mesos) to provide resources for launching executors -> provide in-memory storage for RDD.Lost partitioned is recovered by loggnig its lineange and using That information to reconstruct that partition.

**CH-5-Spark DataFrame:** DF like RDD are immutable, distributed, partitioned collection-> lazy evaluation, recovery through lineage graphs -> specialized APIs to work with tabular data w named cols. Created using RDD.toDF([‘col1’,’col2’]), **spark.read.option(“inferSchema”,”true”).option(“header”,”true).csv(“PATH”).**

A) ratings.groupBy(“movieID”).count(). To display desc sort: A) ratings.groupBy(“movieID”).count().orderBy(desc(“count”)).

To join: ratings.join(movies, ratings.movieID == movies.movieID, “inner”) OR ratings.join(movies, ratings.col(“movieID”) === movies.col(“movieID”)).

To aggregate: df.groupBy(“dept”).agg(sum(“salary”).alias(“Sum\_Sal”)).orderBy(“Sum\_Sal”, ascending=False).

**CH-6-SparkML:** ML is a subset of AI-which uses Statistics, Linear Algebra and Numerical Optim to give the computers ability to learn, without being explicitly programmed to do so. (Recommenders, Stock-predict, Fraud-detect etc). **Machine is said to learn from class of tasks T, as measured by performance P, if it improves with experience E. DataMining:** Non-trivial extraction of insights, from previously unknown raw big data, by auto or semi-auto means. 3types ML: **Supervised** (Classification: Naïve Bayes, Logistic Reg, Decision Trees, Random Forests; Regression: Linear Reg, SVM), **Unsupervised** (k-means Clustering, Hierarchical Clus, Hidden Markov Models, Associative Analysis) and **Reinforcement Learning** (Temporal Diff, Deep Adversarial Net, Q-learning).

**Spark MLlib (RDD-based, deprecated) and Spark ML (DF based-> O(n) scale-out, multi-node parallel compute). Transformer:** .transform() accepts DF as input and return new DF. Do not learn anything new from data. **To either prepare data for model training OR generate predictions using a trained MLlib model. lrModel = new LogisticRegression().setMaxIter(10).setRegParam(0.3) .setElasticNetParam(0.8) -> lrModel.transform(test). Estimator:** Learns or “fits” parameters from your DataFrame via .fit() method and return a model which is a transformer. **Pipeline:** Organises series of Transformers & Estimators into single model. **Small Reg.coeff -> Overfitting. Large Reg.coeff -> Underfitting.** Pipelines in sparkMlLib :- represent workflow of machine learning project, represents series of pipeline stages

Difference RDD and DSM:- RDDs are immutable,rdd based on coarse gained,RDD can be recovered The DataNodes are responsible for serving read and write requests from the file system’s clients

Traditional mp is inefficient for multipass, lacks primitives for data sharing, inefficient in sense that to achieve fault tolerance

How to get the tilte from the movies dataset :- movies[“title”] , if user gives rating then it’s grouo by(userID)

To train model = lr.fit(train) groupBy creates dependency between parent and child transformers in MLIB :-index to string, tokenizer,stopwordremover

the rank parameter indicate in the Alternate Least Squares (ALS) algorithm :- number of latent factor in the model

Existing storage abstractions have interfaces based on fine-grained updates to mutable state – RAMCloud, databases, distributed mem, Piccolo • Requires replicating data or logs across nodes for fault tolerance – Costly for data-intensive apps – 10-100x slower than memory write Solution: Resilient Distributed Datasets (RDDs) • Restricted form of distributed shared memory – Immutable, partitioned collections of records – Can only be built through coarse-grained deterministic transformations (map, filter, join, …) • Efficient fault recovery using lineage – Log one operation to apply to many elements – Recompute lost partitions on failure – No cost if nothing fails

Queries :- find average length of words that start with z

zWords = words.map(lambda x: x.lower()).filter(lambda x: x[0] == 'z') zWordLengths = zWords.map(lambda x: len(x)) zWordLengths.mean()

Which movie has the highest count of ratings 1. sc.parallelize(ratingsData.countByKey().items()).sortBy(lambda x: -x[1]) moviesData.filter(lambda x: x[0]=='356').collect()

-> **avg ratings for each movie** avgRatings = ratingsData.mapValues(lambda x: (x, 1)).reduceByKey(lambda x, y: (float(x[0])+ float(y[0]), x[1]+ y[1])).mapValues(lambda x: float(x[0])/float(x[1])) | **highest avg rating** avgRatings.sortBy(lambda x: -x[1]) **for lowest remove negative sign** | **movies and ratings tables, and names of the top 10 movies with the highest ratings** sortedRatings.join(moviesData) | **only movies that have at least 100 ratings** counts.filter(lambda x: x[1] > 100)| highCountRatings = sortedRatings.join(moviesData).join(highCount)| **NAMES all movies with the tag 'mathematics'** mathMovies = moviesData.join(tagsData).filter(lambda x: x[1][1] == "mathematics") | **average ratings of movies that contain the tag 'artificial intelligence'** -> moviesData.join(tagsData).filter(lambda x: x[1][1] == "artificial intelligence").join(avgRatings)| **most popular tag**? -> = tagsData.values().map(lambda x: (x, 1)).reduceByKey(lambda x, y: x + y).sortBy(lambda x: -x[1])

**By pyspark** :- **movie has the highest count of ratings** -> ratings.groupBy("movieId").count().orderBy(desc("count")) |**movie with the lowest count of ratings** -> ratings.groupBy("movieId").count().orderBy("count")| **avg ratings for each movie** -> ratings.groupBy("movieId").avg("rating") | the highest avg rating

ratings.groupBy("movieId").avg("rating").toDF("movieId", "avgRating").orderBy(desc("avgRating")) **| the lowest avg ratings just put** orderBy("avgRating") |

**the names of the top 10 movies with the highest ratings ->** ratings.groupBy("movieId").avg("rating").toDF("movieId", "avgRating").orderBy(desc("avgRating"))avgRatings.join(movies, "movieId").orderBy(desc("avgRating")).show()| **the NAMES all movies with the tag 'mathematics'** -> tags.join(movies, "movieId")

tagMovies.filter("tag like 'math%'").show()` | **average ratings of movies that contain the tag 'artificial intelligence'** -> tagMovies.filter("tag like 'artificial intell%'").join(avgRatings, "movieId").show()| **most popular tag** -> tags.groupBy("tag").count().orderBy(desc("count")).show()