ML Lib: Recommender System:
Content Filtering: We create profiles that require the gathering of external information.
Collaborative Filtering: It is based on past behavior.
Advantage: Domain Free Disadvantage: Cold Start
Neighbourhood Method: Computes the relationship between the user or items. It is not scalable. Similar to the k nearest neighbor algorithm.

2.1 : Latent Factor: 1. Recommendations based on characterizing ratings of both items and users.

2.20-100 factors inferred from the rating. If you have a large dataset, the matrix is sparse. In order to avoid this, we do matrix factorization where we reduce the dimension of the matrix by breaking it into 2 parts. K(Dimension) is a hyperparameter. Always less than U and I. Matrix fact Algo: ALS in Pyspark. SGD.

Market Basket Model: Support for itemset I = the number of baskets containing all items in I. Given a support threshold s, a set of items appearing in at least s baskets is called a frequent itemset.

Precision = tp/tp+fp; recall = tp/tp+fn (less precision & less recall = bad model)

Confusion Matrix:

Actual class\Predicted class	C_1	$\sim C_1$
C_1	True Positives (TP)	False Negatives (FN)
$\sim C_1$	False Positives (FP)	True Negatives (TN)

Accuracy = (TP + TN)/All | Error rate = (FP + FN)/All |

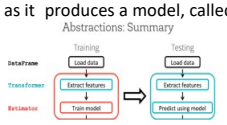
Actual Class\Predicted class	cancer = yes	cancer = no	Total	Recognition(%)
cancer = yes	**90**	**210**	300	30.00 (*sensitivity*)
cancer = no	**140**	**9560**	9700	98.56 (*specificity*)
Total	230	9770	10000	96.40 (*accuracy*)

- *Precision* = 90/230 = 39.13% *Recall* = 90/300 = 30.00%

Quiz: silhouette distance (low values of silhouette are NOT desirable); ALS code running slow – to help - increase value of **numBlocks** ; ALS true – (user column datatype canNOT be string or obj)

```
als = ALS(maxIter=5, regParam=0.01, userCol="userID", itemCol="movieID",
ratingCol="rating", rank = 5,coldStartStrategy="drop")#create ALS model
model = als.fit(training)
```

Pipelines: Transformer: transform() from one df to another(Feature Trans, Learning Model) , Estimator .fit() accept df and produce model. Overall, a Pipeline is an Estimator as it produces a model, called PipelineModel which is a transformer.



Parameter Tuning Steps : 1 Create a Pipeline with training stages. This should include model creation Estimator. **2** Create a parameter grid using the ParamGridBuilder class. This is a grid for all values of the parameters that you want to test. **3** Define an evaluator, such as BinaryClassificationEvaluator, which will be used to evaluate the model. **4** Create a CrossValidator object, which will split data into training and testing parts with a choice for folds. **5** Call the CrossValidator.fit() method and it will try all possible choices of parameters and give you the best choice

GraphX: `graph frame library:`

```
graphframes.graphframe.GraphFrame v =
spark.createDataFrame([(id, property)], ["id", "name",
"age"]) sqlContext.createDataFrame([],[])
e = spark.createDataFrame([(srcid, destid, relatshp),["src", "dst",
"relationship"])
```

Graphframes can be created from single df containing edges info becoz vertices can be inferred from src and dst of edges df. **g = GraphFrame (v, e)**

Breadth First Search: g.bfs("id = 'a'", "id='c'").show() from a to c (fromExp , toExp, edgeFilter, maxPathLength)

Shortest Path: results = g.shortestPaths(landmarks=["a", "d"]) - from evry node to a and d results.select("id", "distances").show() No of indegrees more more pagerank.

PageRank: results = g.pageRank(resetProbability=0.15, maxIter=10,sourceID="a") vertices table – pageRank col and edges - weight

motifs = g.find("(x)-[e]->(y); (y)-[e2]->(x)") can work (a)-[]-(b) in this case no edge column in df. It is acceptable to omit names for vertices or edges in motifs when not needed. E.g., "(a)- []->(b)" expresses an edge between vertices a, and b but does not assign a name to the edge. There will be no column for the anonymous edge in the result DataFrame. Similarly, "(a)-[e]->()" indicates an out-edge of vertex a but does not name the destination vertex. These are called anonymous vertices and edges.An edge can be negated to indicate that the edge should not be present in the graph. E.g., "(a)-[]->(b); !(b)-[]->(a)" finds edges from a to b for which there is no edge from b to a.

Restrictions: Motifs are not allowed to contain edges without any named elements: "()-[]->()" and "!()-[]->()" are prohibited terms. Motifs are not allowed to contain named edges within negated terms (since these named edges would never appear within results). E.g., "!(a)-[ab]->(b)" is invalid, but "!(a)-[]->(b)" is valid.

Label Propogation: Run static Label Propagation Algorithm for detecting communities in networks.Each node in the n/w is initially assigned to its own community. At every superstep, nodes send their community affiliation to all neighbors and update their state to the mode community affiliation of incoming messages. It is very inexpensive computationally, although (1) convergence is not guaranteed and (2) one can end up with trivial solutions (all nodes are identified into a single community).

Quiz: correct way to load – v=sqlContext.createDataFrame([…]),[…]
e= sqlContext.createDataFrame g= GraphFrame(v,e)
g.vertices.filter("age>50").show() || to show num of likes by each person with their name sorted by num of likes in des→ joined = g.edges.join(g.vertices,g.edges.src == g.vertices.id)
joined.groupBy("name").agg(sum("likes").alias("outLikes")).orderby(Desc("outlikes")).show()||
display the name & PageRank in desc → results.vertices.orderBy(desc("pagerank")).show()||
num of people liked bob's post → joined = g.edges.join(g.vertices,g.edges.dst==g.vertices.id)
joined.groupBy("name").count().filter("name=='bob'").show()
shortest directed path → r = g.shortestPaths(landmarks=["1"]) r.select("id","dist").show()|
1 g.vertices.show(...) 2 g.edges.show(...) 3 g.find(pattern) 4 g.filterVertices(criteria)
5 g.connectedComponents.run() 6 g.stronglyConnectedComponents.run(...) 7 g.pageRank(...)
8 g.triangleCount.run()
edges.join(vertices.edges("dst")===vertices("id")).groupBy("src").avg("age").orderBy("src")

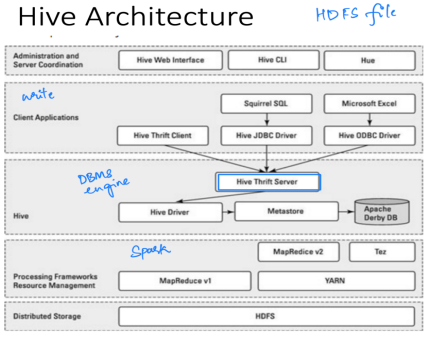
Structured Streaming: fast, scalable, fault-tolerant due to checkpointing, end-to-end exactly-once stream processing engine built on the Spark SQL engine. Internally Structured streaming queries are processed using a micro-batch processing engine which processes data streams as a series of small-batch jobs. Programming Model Structured streaming is to treat a live data stream as a table that is continuously appended. Every data item that is arriving on the stream is like a new row being appended to the input table.

The result is different from the output. Output has 3 modes: Complete mode, Append mode, and Update mode (if no agg then append). Streaming Data frames can be created through the Data Stream Reader interface. Input Sources: File source, Kafka source, Socket source (no fault tolerance), Rate source Output Sinks for streaming: File, Kafka, Foreach, ForeachBatch, Console, Memory

Quiz: features: use DataFrame API,scalable&fault-tolerant, built on spark sql engine, agg,event-time.. can be performed| guarantees- end-end exactly once processing, fault-tolerance, process end-end data in 1 ms, streaming data expressed same as batch data|valid parameter for spark.readstream – socket, kafka, filesource|valid output modes- complete,update&append| schema of file based sources should be specified | true: specify param maxFilesPerTrigger, SS can read from Kafka,File system| True: NOT incoming stream data is written to static table and purged correct- add multiple property fields on each node, g=GF(v,e), 2 dfs are needed- 1 for edg and 1 for nodes, NOT we need atleast 1 field with id of the node, NOT we require atleast 2 cols for each edg

Hive: Data warehousing infrastructure based on Hadoop, designed to enable easy data summarization, ad-hoc querying, and analysis of large volumes of data. Supports Map reduce capabilities. Provides a query language called Hive QL based on SQL. High Latency, cannot perform Online Transaction Processing, No real-time queries, No ACID compliance.

Architecture:



Data Units in Hive: Database->Tables->Partition->Buckets

Partition: Each table can have one or more partition keys that determine how the data is stored. Buckets: Data in partition may in turn be divided into buckets based on the value of the hash function of some column of the table. Data types: Integer: Small int, Tiny int, Big int, int | Boolean: True/false, Floating: Float, double String: string, Struct, Map, Arrays | Path: hdfs://master_server/user/hive/warehouse/database/table |
user/hive/warehouse/database/table/partitions

Local signifies local file system, if it is omitted then it looks for the file on HDFS Command: CREATE TABLE business (businessid STRING, fullAddress STRING, categories STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY '1';
LOAD DATA LOCAL INPATH '/home/hadoop/business.csv' OVERWRITE INTO TABLE business;

```
CREATE TABLE employees (
  name         STRING,
  salary       FLOAT,
  subordinates ARRAY<STRING>,
  deductions   MAP<STRING, FLOAT>,
  address      STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>
)
PARTITIONED BY (country STRING, state STRING);
```

Joins involve MapReduce jobs in the background. EXPLAIN {query} explains to you how a query works.

Quiz-1. built on top of Hadoop; mechanism to impose structure on a variety of data formats; can link to files stored in HDFS or import files from local system; data warehouse technology. 2.Results of a hive query can be stored as : HDFS file, another table, by storing output in a table and then download locally 3.Not a complex type in hive : matrix (complex types are : structure, maps, arrays) 4.Use of partition keys in hive : it determines how data will be physically stored. 5.Tables are created in /user/hive/warehouse directory on HDFS 6.Executable engines in Hive : Tez, MapReduce, Spark (not python) 7.Features of the hive metastore: **data abstraction**: keeping track of the metadata of the tables **Data discovery** : enables users to discover and explore relevant and specific data in the warehouse
- main adv of creating partitions – faster query performance| player with highest runs for each year SELECT a.year,a.player_id,a.runs from batting a JOIN (SELECT year,max(runs) runs FROM batting GROUP BY year) b on (a.year = b.year AND a.runs = b.runs)| **False** : hive data must be copied into hive warehse|

HIVE QUERY:

```
List the userid, name, and number of reviews of the top 8th user who has written the
most reviews.
select i.userid, i.cntReview, u.name from (select r.userid as userid,
count(r.reviewid) as cntReview from review r group by r.userid order by cntReview
desc limit 8) as i inner join users u on i.userid = u.userid order by i.cntReview
limit 1;
List the businessid , full address and categories of the Top 10 highest rated
businesses using the average ratings.
select b.businessid, b.fulladdress, b.categories from business b INNER JOIN (select
b.businessid, AVG(r.stars) as avgStars from review r inner join business b on
r.businessid = b.businessid GROUP BY b.businessid ORDER BY avgStars DESC LIMIT 10)
AS i ON b.businessid
= i.businessid;
```

NotOnlySQL : refers to family of DBs **BASE(**Basically Available Soft state Eventual Consistency **):**

CAP Theorem: (Brewer's) : only two among C,A,P for very large systems

CA: Microsoft SQL Server, MariaDB, MySQL, PostgreSQL, Aster Data,Greenplum, Vertica, Neo4j.|**CP:** Apache HBase, MongoDB, Redis, BigTable, HyperTable, Scalaris,BerkeleyDB, MemcacheDB, Terrastore.| **AP:** CouchDB, Cassandra, Riak, Dynamo, Voldemort, Tokyo Cabinet, KAI, SimpleDB **| Only A:** each client can always read & write| **Only C :** All clients always have the same view of the data | **Only P:** The system works well despite physical n/w partitions.|**Relaxing C** makes replication easy, facilitates fault tolerance.| **Relaxing A** reduces (or eliminates) need for distributed concurrency control**.**

Categories of NoSQL Databases:

1) Key-Value Stores: SimpleDB(+:very fast, scalable, simple model, -ve: many data structures, cant be modelled as kv pairs), **2) Document-Oriented:** MongoDB **3)Column-Oriented:** Cassandra,HBase,BigTable **4)Graph NoSQL Database:** Neo4j

Review/Quiz : not a reason NoSQL is popular- improved ability to keep data consistent, vertical scalability, ability to use traditional SQL| Preference: Availability > eventual Consistency > Cost of hardware> Normalised data|no high availability as master node is single point of failure| instance of single SQL server – normalized data, consistency| Options in traditional RDBMS for storing BD – 1. Master-Slave (prob : reads-inconsistent, write propogation, prob with very large datasets) 2. Sharding – (prob : can no longer join across shards, loss of referential integrity) | incase of BD- **P**artition is absolutely needed| by adoptiong NoSQL, giving up – NOT speed(RDBMS)

Mongo DB (C+P)

Provides – open source doc; high performance - high availability - automatic scaling - non-relational - can be distributed / partitioned easily

Hierarchy: Databases>collections(tables)>documents(entries)>fields

Bulk Import: mongoimport command can work with JSON, CSV, and other formats.

Values always in " ".

```
→Queries: let say table name is listing.
→db.listing.count(); # find the no of docs in the collection
→db.listing.insertOne({name: "ABC", age: 22, address: "India", status: "D"});
→db.listing.find({age: {$gt: 18}, name: 1, address: 1});
→ db.listingsAndReviews.find( { "address.country": "United States" } );
```

```
→ db.listingsAndReviews.distinct( "address.country")
→db.listing.aggregate([{$match: {status: "A"}},
    {$group: {_id: "$cust_id", total:{$sum: "$amount"}}}]);
→db.listing.mapReduce(function() { emit (this.cust_id, this.amount); }
  function(key, values){return Array.sum(values);}
   {query: {status: "A"}, out: "order_details"}).find();
→db.listingsAndReviews.find({"reviews.comments":{$regex:
/bedbug/i}}) Find the count of listings grouped by country
db.listingsAndReviews.aggregate([
{ $group: { "_id": "$address.country", "count": { $sum: 1 } } ]);
```

9. Find the average review scores.review_scores_rating grouped by each country `db.listingsAndReviews.aggregate([ { $group: { "_id": "$address.country", "avgRating": { $avg: "$review_scores.review_scores_rating" } } }]);`

10. From the listings that match following criteria: description should contain 'new york city' and amenities should contain 'kitchen', find the count of each property type.

```
db.listingsAndReviews.aggregate([{ $match: {
"description":{$regex: /new york city/i}, "amenities":{$regex: /kitchen/i} }
},{ $group: { "_id": "$property_type" , "count": { $sum: 1 } }
}]);
```

11. Insert the orders records and find the total amount grouped by customer id.

```
db.orders.insert({cust_id: "A123", amount: 500,
status: "A"}) db.orders.insert({cust_id: "A123", amount: 250,
status: "A"}) db.orders.insert({cust_id: "B212", amount: 200,
status: "A"}) db.orders.insert({cust_id: "A123", amount: 300,
status: "D"})
db.orders.mapReduce( function() { emit(this.cust_id, this.amount);
},function(key, values) { return Array.sum(values)},{query: {status: "A"},
out: "order_totals"}).find()
```

Name	Description	Usage
ANY (writes only)	Write to any node, and store *hinted handoff* if all nodes are down.	Highest availability and lowest consistency (writes)
ALL	Check all nodes. Fail if any is down.	Highest consistency and lowest availability
ONE (TWO, THREE)	Check closest node to coordinator.	Highest availability and lowest consistency (reads)
QUORUM	Check quorum of available nodes.	Balanced consistency and availability
LOCAL_ONE	Check closest node to coordinator, in the local data center only.	Highest availability, lowest consistency, and no cross-data-center traffic
LOCAL_QUORUM	Check quorum of available nodes, in the local data center only.	Balanced consistency and availability, with no cross-data-center traffic
EACH_QUORUM	Only valid for writes. Check quorum of available nodes, in each data center of the cluster.	Balanced consistency and availability, with cross-data-center consistency
SERIAL	Conditional write to quorum of nodes. Read current state with no change.	Used to support linearizable consistency for lightweight transactions
LOCAL_SERIAL	Conditional write to quorum of nodes in local data center.	Used to support linearizable consistency for lightweight transactions

12. Run mapreduce on the restaurants dataset to return number of restaurants grouped by cuisine in the Manhattan borough

```
db.restaurants.mapReduce( function() { emit(this.cuisine, 1); }, function(key,
values) { return Array.sum(values)}, {query: {borough: "Manhattan"}, out:
"cusine_totals"}).find()
```

Quiz:list all distinct markets under the address → db.listingsAndReviews.distinct("address.market")| count→db.listingsAndReviews.aggregate([{$group:{"_id":"$address.market","count":{$sum:1}}}]) ;| just prop_type & price of NY in desc → db.listingsAndRevws.find({"adr.market":"NY"},{"prop_type":1,"price":1," "_id":0}).sort({"price":-1})| num of bdrms→ db.listingsAndReviews.aggregate([{$match:{"adr.mkt":"Montreal", "prop_type":"Cond"}}, {$group:{"_id":"$bdrms","avgPrice":{$avg:"$price"}}}]);| count cancellation in desc db.listingsAndReviews.aggregate([{$group:{"_id":"$can_pol","count":{$sum:1}}}]).sort(:-1)) MongoDB docs are represented in JSON format| True : NOT collections require complete schema| record consists of **kv pairs** and is refered as **document**|

HBase: (C+P) (Low Latency) Distributed & Sparse Column-oriented database built on HDFS which is horizontally scalable and provides quick random access to huge amounts of data. HBase is schema-less, doesn't have ACID properties, is non-transactional and is good for semi-structured data. HBase is, tables are sorted by row and the table schema only defines families. Each family is identified by the prefix. The key consists of Row Key, Column family name, Column name, and version. HBase is best when you need random write, random read or both, therefore every cell has timestamps. Access patterns are simple.

Architecture: HBase tables are divided into regions. Regions represent a subset of the table's row. Each region is served by a region server. The region server serves data for reads and writes. The region Servers are coordinated by a master. Master also assigns regions, detects failures of region servers. Regions are the units that get distributed over an HBase cluster.

Internally, HBase keeps a special catalog table named hbase:meta within which it maintains the current list, state and locations of all user-space regions afloat on the cluster. HBase has HBase master node orchestrating a cluster of one or more region server workers.Master node roles: Bootstrapping install, Assign regions to region servers, Recovering from region server failures | Region Server roles: Manages region splits informing master about changes. HBase depends on ZooKeeper and by default it manages a zookeeper instance as the authority cluster.

CRUD Operations:

- **Put** – add/update; - **Get**- access data from only one row; **Scan** – access data from a range of rows; **delete** – del a row

```
while creating table: at least 1 column family should be defined.
create a table called customer with two column families – "addr" and "order"
create 'customer', {NAME=>'addr'}, {NAME=>'order'}
put 'customer', 'jsmith', 'addr:city',
'nashville' get 'customer', 'jsmith'
get 'customer', 'jsmith', {COLUMNS=>['addr']}
get 'customer', 'jsmith', {COLUMNS=>['order:numb']}
Get all the table data and also count the number of rows in each
table scan 'clicks' | count 'clicks' | count 'students'
Display the details for user 'njones' and then for all those users where the name
starts with 't' | get 'customer', 'njones' |scan 'customer', { STARTROW => 't'}
```

Quiz: HBase modelled after Google Bigtable| Data stored order: Row->ColumnFamily -> Column Qualifier-> Timestamp in reverse order| HBase doesn't support traditional joins| True : indv col can be added dyna w/o req a schema change, each row is indexed and sorted| True : sparse and distributed, multidimensional, each cell contains mult values NOT indexed by mult keys| True : represent indexed, ordered subset of rows, one region can contain rows having diff column fams and can spread across multiple datanodes of hdfs, managed by region servers|

CASSANDRA

Massively linearly scalable NoSQL database. Fully distributed with no single point of failure. Any node can serve as a coordinator to cater requests. No single point of failure, due to horizontal scaling.Sharding (horizontal partitioning). Cass excels when you need: No single point of failure| Real-time writes with live operational data analysis| Flexible, easily altered data models| Near-linear horizontal scaling across commodity servers |Reliable replication across distributed data centers| Clearly defined table schema in a NoSQL environment

A peer to peer set of nodes: • Node – one Cassandra instance • Rack – a logical set of nodes • Data Center – a logical set of racks"• Cluster – the full set of nodes which map to a single complete token ring. Nodes join cluster with this file : conf/cassandra.yaml file : Key settings include :: **• cluster_name • seeds • listen_address**

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

A system on each node which hashes tokens from designated values in rows being added is called partitioner. • Hash function – converts a variable length value to a corresponding fixed length value.

Node tokens are the highest value in the segment owned by that node.

A node's partitioner hashes a token from the partition key value of a write request.

Replication factor is configured when a keyspace is created:

SimpleStrategy (learning use only) – one factor for entire cluster

NetworkTopologyStrategy – separate factor for each data center in cluster

Immediate Consistency – reads always return the most recent data

• Consistency Level ALL guarantees immediate consistency, because all replica nodes are checked and compared before a result is returned

• Highest latency because all replicas are checked and compared

• Eventual Consistency – reads may return stale data

• Consistency Level ONE carries the highest risk of stale data, because only one replica node is checked before a result is returned

• Lowest latency because the response from one replica is immediately returned

Consistency Level ONE	Consistency Level QUORUM	Consistency Level ALL
Lowest latency	Higher latency (than ONE)	Highest latency
Highest throughput	Lower throughput	Lowest throughput
Highest availability	Higher availability (than ALL)	Lowest availability
Stale read possible (if read CL + write CL < RF)	No stale reads (if read and write at quorum)	No stale reads (if either read or write at ALL)

Nodes exchange data and info between them using the Gossip protocol.

• Once per second, each node contacts 1 to 3 others, requesting and sharing updates about

• Known node states ("heartbeats") • Known node locations

A keyspace defines a cluster's replication strategy and replication factor

The Murmur3Partitioner is the default and best practice

The nodetool repair command makes stale data consistent for a node or set of nodes

• IF nodes_written + nodes_read > replication_factor THEN results are immediately consistent Nodes continually exchange state and location information via the **Gossip protocol**.

The Cassandra data model defines: 1. Column family as a way to store and organize data 2. Table as a two-dimensional view of a multi-dimensional column family 3. Operations on tables using the Cassandra Query Language (CQL)

Row is the smallest unit that stores related data in Cassandra:

• Rows – individual rows constitute a column family

• Row key – uniquely identifies a row in a column family

• Row – stores pairs of column keys and column values

• Column key – uniquely identifies a column value in a row

• Column value – stores one value or a collection of values

Column keys are inherently sorted

A row can be retrieved if its row key is known

A column value can be retrieved if its row key and column key are known

Any component of a row can store data or metadata

• Simple or composite row keys

• Simple or composite column keys

• Atomic or set-valued (collection) column values

Column family – set of rows with a similar structure

Partition key used for data placement on nodes, does not have to be unique

data with same value of partition key is stored together any filter query should specify partition key. Composite partition key (state + city)

PRIMARY KEY: Partitioning key + clustering key(unique)

minimum thing to give in filter query is partition key

A CQL table is a column family. CQL table and column family are largely interchangeable terms.

Quiz : prop- NOT master-slave| **snitch**- determines which node to fwd req to| primary key – (partition key, clustering key) | Gossip – spread n/w topology to all peers| true : to retrive col v, row and col key must be known, col keys in a row always sorted | true : legal to partition on >= 1 col; clustering key provides efficient way to look up data ; good idea to partition on primary key; NOT clustering key should be unique |

Lab:

```
CREATE KEYSPACE IF NOT EXISTS cycling WITH REPLICATION = { 'class' :
'SimpleStrategy', 'replication_factor' : 1 };
USE cycling;
CREATE TABLE cycling.cyclist_category ( category text, points int, id UUID, lastname
text, PRIMARY KEY(category, points))WITH CLUSTERING ORDER BY(points DESC);
INSERT INTO cycling.cyclist_name (id, lastname, firstname) VALUES (5b6962dd-3f90-4c93-
8f61-eabfa4a803e2, 'VOS','Marianne');
first entry of primary key is partition key that determines which node stores data.
SELECT lastname, firstname FROM cycling.cyclist_name WHERE id = 6ab09bec-e68e-48d9-
a5f8-97e6fb4c9b47; W
SELECT lastname, firstname FROM cycling.cyclist_name WHERE firstname = 'Alex'; DW
Which of the following queries would work?
a. SELECT * FROM cycling.cyclist_category;
b. SELECT category FROM cycling.cyclist_category;
c. SELECT category FROM cycling.cyclist_category WHERE category='GC';
d. SELECT category FROM cycling.cyclist_category WHERE points=1269;
e. SELECT category FROM cycling.cyclist_category WHERE lastname='TIRALONGO';
filter on non-partitioning key if we use the keyword ALLOW FILTERING and index
SELECT * FROM cycling.calendar WHERE race_start_date='2015-06-13' ALLOW FILTERING;
CREATE TABLE cycling.rank_by_year_and_name ( race_year int, race_name text,
cyclist_name text, rank int, PRIMARY KEY ((race_year, race_name), rank) );
SELECT * FROM cycling.rank_by_year_and_name WHERE race_year=2015; DW
CREATE INDEX ryear ON cycling.rank_by_year_and_name (race_year); ; W
CREATE INDEX rrank ON cycling.rank_by_year_and_name (rank); ; W
SELECT * FROM cycling.rank_by_year_and_name WHERE rank = 1;
INSERT INTO cycling.last_3_days (race_name, year, rank, cyclist_name) VALUES ('Giro
d''Italia Stage 16','2015-05-26',1,'Mikel Landa') USING TTL 259200;
SELECT TTL(race_name) FROM cycling.last_3_days;
SELECT * FROM cycling.last_3_days; // WILL ONLY SHOW NON-EXPIRED ROWS
```

S.No	Cassandra	Hbase	MongoDb
	P2P	Master/Slave	NA
	Linear Scalable - for throughput	X	X
	Horizontal scalable		
CAP Theorem : Brewer's Theorem	A+P	C+P	C+P
	Column Based	Column Based	Document based
Hierarchy	Keyspace, Column Family, Partition Key, Clustering key	(Row key, Column Family, column qualifier,Version):Value	: db, collections,documents,fields
Architecture	Cluster>Datacenter>Racks>Node	Master(HDFS-ZooKeeper)>Region Server>DataNodes	NA
NoSQL	No Joins, Integrity constraints & No ACID , Follows BASE(Basically Available soft State Eventual		
Based On	Google Big Table : Data model, Amazon Dynamo : Distribution Backbone	Google BigTable : Datamode	NA
Sorting Order	Column Key sorted(asc)	Rowkey : Asc , Version : Desc	NA
Structured	Semi Structured & Structured	Semi Structured & Structure	Unstructured,Semi Structured & Structu

1. Hive vs HBase: hive is slow& doesn't provide real time interactive queries like HBase | **2.** Spark reconstruct lost portions by using lineage info present in each obj| **3.** distributed, column-oriented, scalable, NoSQL – Hbase (not spark, RDBMS,Hive)| **4.** Hive : Select e.name, s.salary from employee e inner join salary s on e.empid orderby s.salary desc limit 5; Spark: employee.join(salary).orderBy(x=>-x._2).take(5)| **5.** ML techs : classification – SVM model; Clustering – Kmeans; Recommender – matrix; regression – DecisionTree| **6.** Create dw and ETL app – Hive| **7.** Spark reconstruct lost portions – using lineage info present in each obj| **8.** Inc query times : impala<HBase<(?)SparkSQL<Hive|