**Introduction**

Exam Blueprint:

* 3 Hours exam, longer than Associate, USD $300
* Recommend study: Big data fundamentals (free) and Big Data on AWS (instructor led, not free), under aws.amazon.com/training

General information and pre-requisite

* Not much things are free, so remember to delete. To save cost, us-east-1 or us-west-2 are normally the cheapest.
* You need a EC2, a M5.large memory optimised instance would be ideal
* You need a SQL client, free options include Aginity Workbench for AWS Redshift and SQL Workbench

Generate data (You can use a native Linux machine, or a windows subsystem for Linux, and not use a EC2 at all, or ):

* Use TPC-H tools to generate dataset. TPC-H benchmark is popular to comparing database vendors (like the benchmark tools for mobile CPUs). Simply SSH into the data generator instance, install git make and GCC, clone source code from GitHub and make it.
* Create a directory to store the generated data (/emrdata) and add the directory to environment variable with export DDS\_PATH = $HOME/emrdata, so the directory is registered for use.
* Generate data by the following command, and it should go to $HOME/emrdata directly, to generate a dataset of 10G, may take about 10 mins:
  + ./dbgen -v -T o -s 10
* Copy the generated file to S3 (from CLI would be most convenient)
  + aws s3 cp $HOME/emrdata s3://mybucket\_name/mybucket\_directory
* Generate data for redshift with a similar procedure, create a directory, set environment parameter to point to that directory, and run a similar command
* This time to upload we should count the lines and split the files (learn wc and split commands in linux):
  + wc -l orders.tbl
  + split -d -l 15000000 -a 4 orders.tbl.
  + split -d -l 60000000 -a 4 lineitem.tbl lineitem.tbl.
  + The result will be files like orders.tbl.0001, lineitem.tbl.0000, etc
* Turn off the instance now.

**AWS Big Data Fundamentals:**

Big Data Overview

* About massive data that’s hard to be handled by traditional applications, in terms of: Analysis/Search/Transfer/Updating etc.
* Three key attributes: Volume (size), Velocity (frequency of being generate and handled), Variety (types of content, structured and unstructured). When these things become bottlenecks, you need Big Data solution
* Primary drive: data growth, multiple sources (automatically generated and human generated), demand for analysis (scientific and supercomputer approach is not enough, we need engineering approaches, to avoid being data rich and value poor)

Architectural components

* Parallelism: distribute the load to optimise on compute time
* Hadoop: Standard platform, not a DB or alternative, it’s a scalable data store and batch processing framework. Ingests, processes and aggregates external data and export the result for further use
* Hadoop common: common library
* HDFS: Hadoop distributed file system. Consists of data nodes and name nodes in a fleet of racks. 3 copies of each data block are stored by default, and when rack awareness is enabled (Hadoop 2.X feature), the three copies will span at least 2 racks. Client only write to HDFS once and all the location mapping and replications are handled by Hadoop.
* Hadoop YARN (Yet another resource negotiation): for scheduling and execution of data processing. Client calls a Resource Manager on a name node -> calls node Manager daemon to decide which data node runs the job -> Node manager reports container usage (Ram, Disk, CPU, Network) back to Resource Manager -> Resource Manager decide on which node to launch the application master to run the job -> application manager run the code, return the result and release the resources
* Hadoop MapReduce: YARN based system to perform processing in the cluster, parallel processing for large datasets, Map -> reduce, distribute computing -> aggregate result
* Hive: DWH Infrastructure, Use HQL, which is close to SQL, but allow unstructured data query. Enables summarization, analysis and ad hoc querying.
* Pig: open source, programming environment for Pig Latin, for queries and data manipulation, works well with MR and SQL.
* Use cases: Create a Hadoop cluster to collect and store (you can form the collect server into a storage cluster), use HQL to perform analysis

Database Architectures

* RDB
  + Downside of flat file: no regulation, different representation for the same thing, or same representation for different things (e.g. same name).
  + RDB solve this by using an ID, and breaking valid answers into smaller tables, and ID in these reference tables will replace the ambiguous representations
  + Each query returns a “view”, each column also called an “attribute” and each roll is also called a “tuple”.
  + SQL and most RDB has ACID compliance, which means
    - Atomicity, all done, or nothing done
    - Consistency, only valid data is saved
    - Isolation, handle concurrency so transaction won’t affect each other
    - Durability, written data will be saved even components failed
  + ACID compliance may not be necessary under some circumstances, say when data is distributed and synced asynchronously, so consistency didn’t have to be enforced at each individual write time.
* NoSQL
  + Forgo tables, use files to achieve dynamic schema. Avoid needing to alter schema and migrate data into the new schema when wanting to add more information to existing data
  + Have different types:
    - Key-value stores (like JSON, use key to access value which can be any binary. Most flexible, deal with high velocity well and less effective for transactional processing), Redis, Oracle BDB, DynamoDB
    - Document stores (Key-Value, with value being a document, consider key-object pair), MangoDB, CouchDB
    - Wide-column stores (column can have column, like recursive JSON), Apache Cassandra, HBase
    - Graph Stores (graph structure with node and edges), InfoGrid, Neo4J, Infinite Graph
  + RDB need to scale up, NoSQL can scale out (designed to be distributed). RDB SQL is more powerful in querying, NoSQL is better for lighter transaction load that requires flexibility. ACID is generally not enforced in NoSQL, although it’s achievable via design.
* DWH
  + OLTP vs OLAP high transactions low volume vs low transactions high volume. Many vs few user, short vs long term retention, operation vs reporting and analytical data (BI)
  + DWH is normally for OLAP, Source -> ETL -> DWH -> Metadata/Summary data -> OLAP/BI reporting/data mining
  + Benefit: Consolidation, Isolation from prod, Long history can be retrieved, great consistency, Performance, provide more value
* Data mart
  + Department wide instead of enterprise wide, normally regrading single subject (functional area) and less complex

Hadoop and MapReduce

* Overview
* Use Cases
* 1.X and 2.X
  + We focus on 2.X, 2.X introduced YARN to separate resource management and MapReduce computation. 2.X also enabled more applications to be perform jobs on HDFS
  + 1.X only have one Name node (and a Secondary Name node, however this is not a standby, just a backup for later rebuild the cluster), hence bottle necked and doesn’t scale well (can even be single point of failure), can only handle about 4000 data nodes.
  + 1.X only support MapReduce Jobs, Job Tracker -> task runner -> data node. 2.X uses YARN, enables more application.
  + 2.X has added a passive name node on top of 1.X, allows failure-over. All data node is aware of both name node, and reports are sent from data node to both name node. All write goes to active name node but read can go through both active and passive node.
  + 2.X has backward compatibility, works on NFS, can run on windows, can take snapshots for backups, can run general application in containers, instead of run only MapReduce Job in node manager daemon.
  + 2.X also support name node federation, allows multiple name nodes for the same cluster of data node. This provide better scalability too.

MapReduce:

* Parallel data processing software platform, process across a cluster of computers.
* App written in Java/Python/C#/C++ -> Hadoop Cluster -> HDFS
* Prior to MapReduce, massive data processing is done by moving data to one place and then process on supercomputer, application and processing/storage is tightly coupled.
* MapReduce can be deployed on cheap hardware, scale, run on node, tolerate fault on node level (each node is required to report back to master node, and if TTL based health check failed, master will re assign the work to another node), have more tools like pig and hive.

MapReduce Phases:

* MR operates on key/value pairs, takes key/value pairs in and produced key/value pairs output.
* Data -> generate key/value pairs -> mapping -> shuffling -> reducing -> report
* MapReduce Joins: Mapper side (Faster and more constrain), Reducer side (Slower but less constrain), Distributed Cache
* There’s also combiner and partitioner which are less important

Hive

* DWH framework, use HDFS for storage and MapReduce for processing, then use HQL to query. Benefit includes reuse SQL skills, manage unstructured data
* Normally use cases include daily reporting/summarisation, ad-hoc querying.
* Hive and Hadoop is generally not an alternative to SQL and RDBMS. Hive is generally slower, not ACID Compliant, and less structure due to schema on read (no gate keeping rule when data was written to HDFS).
* Component: CLI or API (Thrift server for cross language) -> Driver compile, optimize, generate a DAG to execute -> MapReduce on Hadoop Cluster
* CLI can execute DDL and metadata exploration commands
* Hive organize data into Database, Tables (can be internal/HDFS based or external), Partitions (reduces computation in mapping and reducing) and Buckets (file in a table partition)
* Hive Views is like RDB views
* Table can have indexes, to avoid scanning full data set. Compaction and Bitmap.
* Hive Metastore tells Hive where data live in HDFS

Pig

* Big Latin script language for analysing large datasets, Pig compiler compiles it into MR jobs, and optimisation happens automatically.
* Easy to pick up by programmers familiar with scripting languages and SQL. Need only 5% lines of code for the same functionality compare to Java.
* Simplifies complex data analysis, fewer lines than writing from scratch.
* Pig is developed for Hadoop but is not limited to be used with Hadoop
* Pig code once compiled, can run on both MapReduce 1.X or 2.X
* Pig and pipelining data???(Kinesis??)
* Pig Latin supports nested data types like tuples, bags and maps. Supports user defined functions (UDFs)
* Identifier is similar as variable names
* Simple data type like int (4 byte), long (8 byte), float (4 byte), double (8 byte), bytearray, Boolean, chararray (both string or array of characters).
* Complex data type: data atom (string that can be used as string or number), tuple (order set of any data, can be compared with rows of RDB), data bag (a collection of any tuples), data map (a set of key/value pair)
* Pig Relation vs RDB Relation: Bag of tuples can have duplicated tuples, # of columns can vary, data type for each column can vary, Pig Latin can be added anywhere(?), native ETL.
* Even with all the flexibilities, Pig Latin can have schema (optional), which regulates names and types to fields (type is optional, when not specified, default is bytearray). Provide parse time error checking.
* Input uses LOAD, LOAD ‘source.txt’ USING myLoad() AS (tuple1, tuple2, tuple3). myLoad() function can be replaced by pigStoreage() for delimited flat file.
* Output uses dump, store data uses store, print file uses cat
* Pig has local/MR mode, local mode will use local computation power and file system, MR mode will use distributed computational power and HDFS.
* Invocation can be step-by-step manual, using Pig’s Grunt shell (For trouble shooting), or batch Mode (for production)
* User defined functions: Built in ones are distributed with Pig releases, User contributed (community repository called piggybank), Customer
* Pig Joins: join tuples by keys, supports Self Join, Inner Join, outer join
* Fragment Replicate Joins: improve performance when the join relation is small enough to fit in memory. The process will send the join table to each node (Hadoop distributed cache) and join them,

AWS and Big Data ecosystems

* AWS can handle capacity, can handle experimental cases with on demand resources, can use elasticity and scalability to cope with variable workload, can handle both structured and un-structured data
  + Collect: Kinesis/firehose for real-time, snowball for import, IoT
  + Store: S3, RDS, DynamoDB
  + Process & Analysis: EMR, Lambda + kinesis Analytics, Redshift for DWH, Amazon ML, Elastic Search, Athena,
  + Visualisation: QuickSight
* Can also leverage 3rd party tools like:
  + Collect: Fluentd, Flume, Log4j Sqoop
  + Store: HDFS, Cassandra, Kafka, Hbase
  + Process & Analysis: Hadoop, Spark, Hive, Pig
  + Visualization: SAS, Tableau, Qlik
* some of them available at AWS market place:
  + Integration: Informatica, Splunk, etc.
  + Advanced analysis: Zementis, SAS, Digital Reasoning, BigML, etc.
  + BI: Tableau, SAP, MicroStrategy, Logi analytics, etc.

Use Cases:

* On demand Analytics: Source -> S3 -> EMR on Hadoop -> RedShift -> Reporting and BI
* Source -> kinesis -> DynamoDB -> Client access
* DWH: Source -> S3 -> EMR/Glue perform ETL -> S3/RedShift -> QuickSight
* Click Stream Analysis: Kinesis -> Custom App with Kinesis Client Library -> customer suggestions
* Event driven ETL: Event (DynamoDB add) -> Lambda -> RedShift -> QuickSight

**Collection**

Kinesis concepts & architecture

* Stream for real time collection (think about power exchange), Kinesis Analytics for processing real-time data with SQL, Kinesis Firehose fire data into redshift
* Use when you need large amount of data processed quickly and need to perform customised analysis against them
* Kinesis use cases (understand this for VS SQS questions):
  + Fast log and data feed intake
  + Real-time metrics and reporting
  + Real-time data analytics
  + Complex stream processing
* Benefits
  + Real-time aggregation of data, for further processing like DWH, map reduce cluster, etc.
  + Durable and elastic, cope with changing data stream density
* Methods to load/get data
  + Kinesis Producer Library allows data producers to achieve high throughput to send data to Kinesis
  + Kinesis Client Library allows application to be consumer of kinesis stream
  + Kinesis Agent used to collect and send data to Kinesis streams
  + Kinesis REST API can put/get records over HTTPS
* Architecture: Producer (KPL, Agent, REST) -> Kinesis with several shard -> Consumers (KCL, Agent, REST) -> storage
* Shard: group of data records in a stream, can be used as processing unit. However, read can be in pieces but write are in chunk: for each shard you get
  + 1M/s input
  + 2M/s output
  + 1000 record/sec write
  + 5 transaction/sec read
* To cope with stream pattern change, you can re-shard, thus splitting or merging shard(s).
* A stream record is: Partition Key, Sequence Number, Data Blob. Note this is not a user record, which is the actual user data itself
  + Partition key decide which shard a record go to, and you should look up this when you want to identify which shard a record came from
  + Sequence Number is a unique key to identify a data blob, and is assigned when a producer calls PutRecord/PutRocords
  + Data Blob: The actual drop of data, max size 1MB (base64 encoding)
* Retention Period: Kinesis is not persistent by any means. How long data is kept in shard, default is 24hrs, can be changed in CLI to up to 7 days, however keeping data for longer in Kenises will incur additional cost.
* Kinesis Streams API: User PutRocords over PutRecord whenever possible, this give you better throughput. You can use AWS SDK for Java (also available in .NET, Node.js, Python and Ruby).
* Goal – to build reliable and efficient producer, read KPL documentation.
* KPL: Can provide
  + configurable auto-retry (by default, on failed stream record will not stop other operation and still return HTTP200. In this case, PutRecordResult can be used to detect if each individual records got passed in, and retry)
  + use PutRecords to write multiple records to multiple shards per resuest
  + Aggregate (batching uses records into one stream record to efficiently use shard, or collecting multiple stream records to save http handshake overhead) to increase payload size to improve throughput
  + Integrate with KCL to de-aggregate records
  + Shoot metrics to CloudWatch, at stream, shard or producer level
* Don’t use KPL where
  + Real time is strictly important, because record can wait for RecordMaxBufferTime when using KPL, before being put to a shard. You don’t want to set it too low as the throughput will be impacted. Any setting it higher increases delay for each individual record.
* Kinesis Agent:
  + Standalone Java program, can be installed on server, can monitor multiple directories (log stream for example) and write to multiple streams
  + Allow you to pre-process data like convert multi-line data to single line, converting delimiter to JSON, convert Apache log/ Syslog format to JSON/CSV, etc.
  + Agent shoots CloudWatch metrics for monitoring
  + Allow you to send data to both Kinesis stream and kinesis firehose stream
* KCL consume and process data, handles complex tasks like
  + load balancing among consumers
  + react to re-shard
  + react to instance failure/ instance autoscaling
  + Checkpointing last processed record (backed by DynamoDB), and resume in the case of processing halted
  + De-aggregate records that’s aggregated by KPL
* Application name/state (and supporting DynamoDB table):
  + KCL uses the name of streams application to create the supporting DynamoDB table, the application names should be unique
  + Each row in the backing DynamoDB table representing a shard, and the hash key for the table is the shard ID.
  + DynamoDB table throughput might be a bottleneck as by default you only get 10 read and 10 write capacity units, especially if your app does frequent checkpointing or you got too many shards (provisioned throughput exception). Obviously to resolve this you need more provisioned throughput to the DynamoDB table.

Kinesis data emission

* Consumers can emit to S3/DynamoDB/ElasticSearch/Redshift/EMR. The first 4 need KCL and Kinesis connector library (can be found on GitHub, with samples for each use case), which is Java based. EMR does not at the moment but may change in the future.
* For each consumer emission options, here’s the use cases:
  + S3 mostly for archiving
  + DynamoDB for data needed for further API use like dashboard/visualisation/reporting
  + Elasticsearch for searching and indexing
  + Redshift Micro batch loading (small ETL batches running in near real time)
  + EMR for Process and analyse
  + Lambda for serverless automated emission, but only to DynamoDB, S3 and Redshift
* Lambda can pool Kinesis shards directly and process/load other AWS storage services, without needing Kinesis connector library.
* Kinesis connector library (highlighted in the below workflow):
  + Connector Library Pipeline: Kinesis Stream -> iTransformer -> iFilter -> iBuffer -> iEmitter -> S3/DynamoDB/ElasticSearch/Redshift
  + iTransformer defines transformations, iFilter is a final filter, when buffer is full, IEmitter call relative API to emit data to configured endpoint of storage services

Kinesis Firehose

* Load data into S3, Redshift and Elasticsearch (and kinesis analytics) in near real time (small batches with buffersize or interval triggers), no need for a consumer. Use existing BI and dashboard tools, highly available and fully managed (Scaling, sharding, monitoring is automatically handled to meet your need).
* If S3 is the destination, firehose can batch and compress (zip/gz) your data to save storage cost.
* Data can be encrypted (KMS based) before loaded so secure in transit and at rest after loaded
* Can create in console or by API.
* Data transformation in Lambda is possible too, but the record must have the following parameter, otherwise will report error.
  + recordId: this must stay consistent before and after processing.
  + Result: transformed record status must be Ok/Dropped/ProcessingFailed. Ok/Dropped will be considered as successful, and ProcessingFailed obviously is unsuccessful.
  + Data: payload being transformed, has to exist
* Use case: producer produces data, send to Kinesis delivery stream (AWS SDK/Kinesis agent), when buffer size (1~128M) /interval (60~900s) condition is met, Kinesis flushes data to target.
* Failure handling:
  + When processing failed with lambda: 3 retires, log error to CloudWatch, unsuccessful records arrives to S3 “processing\_failed” folder (bad file directory)
  + When delivery to S3 failed, Firehose will retry up to 24 hours
  + When delivery to Redshift failed, you can specify a retry period of 0-7200s, after that it skips and list in the manifest file for manual backfill
  + When delivery to Elasticsearch fail, you can specify a retry period of 0-7200s, after that it skips the request, send skipped documents to S3 bucket called Elasticsearch failed, then it can be manually backfilled.
* You have the choice to backup original data to S3 as well (ELT instead of ETL).
* Frequency:
  + If destination is S3/Elasticsearch, this depend on buffer size and buffer interval. However Firehose can raise buffer size dynamically if data arriving at the destination is falling behind
  + If the destination is Redshift, it depends on how fast the Redshift can finish the copy operation, once a copy is finished, the next one is automatically issued.

SQS (VS kinesis come later)

* Sometimes you even want to choose SQS over Kinesis
* SQS is reliable and scalable, hosted by AWS, 256 block size, larger than that can be managed by SQS Extended Library, which uses S3 (for what?).
* SQS ensures delivery at least once, unless using FIFO queue (once and once only with strict order). Supports multiple reader/writer to share the queue.
* Message can be retained for up to 14 days, default 1 day
* long pooling available (up to 20 seconds, cost the same each pool)
* Typical architecture: use two queues to de-couple component, one for input one for getting result back, in this case:
  + Both the upstream server and downstream server can scale up/down independently
  + When performance is not a issue, just use the queue as a buffer
* Queue length can be monitored by CloudWatch and can tell you a lot of things.
* SQS does not support priority, but you can use multiple queues to simulate priorities, thus, only process lower priority queue, when there’s no message in the higher priority queue.
* SQS Fanout: A way to send the same message to multiple SQS queues (image sent to three queue architecture), check later for detailed mechanism.

**Storage**

**Processing**

**Analysis**

**Visualization**

**Security**

**Exam**