**Contents**

[**Big Data and Hadoop Features**](#Big_Data)

[**Hadoop**](#Java8_Features)[**Cluster**](#Hodoop_cluster_setup) **Setup**

[**Map**](#Struts) **Reduce** [**Framework**](#MR_Framework)

[**HBASE**](#Hbase)

[**Zookeeper**](#Zookeeper)

**[Apache Drill](#Drill)**

**Big Data Infrastructure at AirBnb**

[**Hive**](#Hive)

[**Sqoop**](#Sqoop)

[**Chef**](#Chef)

**[Docker](#Docker)**

**Big Data and Hadoop Features**

1. Big Data refers to data generated in order of terabytes, petabytes etc. which cannot be stored or processed using traditional methods i.e. using RDBMS queries, or centralized storage. Storage capabilities are being continuously scaled vertically i.e hard disk capacities in a system have now become in order of few TB. However rate of increase in data generation with advent of internet connected smart phones and other devices has been much faster than increase in storage capacities. Plus a centralized storage becomes **a single point of failure** with loss of lot of data. Backing up on other redundant servers is a time consuming task and does not scale well.
2. In businesses today there are several scenarios where **huge amount of unstructured data** gets generated and needs to be analysed to obtain meaningful information
   1. In **telecom industry**, a service provider has a huge user base with each user having different usage and time preferences. Some users use more mobile data, some are active on messages, and some like to make more calls at nights. Based on this varying information, voice and data plans need to be targeted as per user behaviour, requiring a quick sifting through user data.
   2. In **retail**, this analysis is needed to finding out which **products are brought by which users**, which branch has sold which kind of items, how much has a customer spent at the store outlet, in order to provide relevant offers or stock up certain items at certain branches
3. Tools to analyse Big Data: Since data is generated in huge order, **a distributed scalable system** is needed to store and process. Hadoop is one such platform, allowing multiple commodity hardware to be combined into a **cluster**. Hadoop is based on HDFS, a distributed file system. Once a data is moved to HDFS, it can be analysed through **Map Reduce programs**, however language is not constraint
4. Hadoop system as a whole is installed on multiple Linux machines. It has a **name node and several data node**. A name node is a high end resilient machine, since it requires coordination with all other cluster nodes, and can be a single point of failure if it goes down. **Data nodes are commodity machines**, containing actual data. A big data file of order of terabytes is split into blocks, each block stored on different data node. While a block is being copied, **replication factor configured on name node** decides on which nodes the block has to be replicated. Generally a replication strategy is chosen such that one **block is replicated in different rack and a different data centre**, to make it available in case of rack failure as well as data centre failure.
5. In Hadoop 2.0, YARN has been introduced for managing resources. Several high level packages are now made available for non-java users. **HIVE,** developed by FB on top of MR, provides **an interface similar to SQL**, to be used by data analysts. HIVE queries are internally run over Hadoop as **MR jobs**. PIG is another addition for developers, as scripting tool used for developing **quick prototypes**.
6. **Hadoop has various vendor distributions**, most commonly used the one from Cloudera, and primarily it was first enterprise distribution. While Cloudera express is free edition, enterprise comes with a trial period. Hortonworks is another vendor, whose distribution is completely open source, and is gaining market share. MapR provides a paid distribution, has taken apache Hadoop source and had made customizations to improve performance and reliability. One optimization is **replacement of HDFS with NFS**, however development on MapR can result in vendor lock in
7. Storage on Master nodes should comprise of **SAS drives deployed as RAID array**, to give Hadoop Management services a redundant store for mission critical data. SAS drives are faster than SATA, though handle lesser space. **Name node cannot be commodity hardware like data nodes**, as it is single failure point for entire HDFS. Recommended Specifications for name node system are CPU with 6 to 8 cores, 64-128 GB RAM as Name node itself uses 1GB RAM per million of HDFS blocks and 10 GB Ethernet connection for services on Master nodes to communicate
8. Master Nodes in Hadoop cluster
   1. **Name node & Standby Name node** used for for managing HDFS storage
   2. **Checkpoint node** (backup), provides services i.e. reading edit log for file changes since last checkpoint, and applying changes to Name node’s master file
   3. **Journal node,** receives edit log modifications from Name node
   4. **Resource Manager,** oversees app task scheduling & cluster resource management
   5. **Job tracker,** used in Hadoop 1.0, for cluster resource management and scheduling
   6. **HMaster**, handles metadata changes
   7. **Zookeeper** keeps distributed components in sync, used especially with Hbase. Zookeeper and Journal services are collocated on Master nodes with other services
9. Hadoop Features
   1. Hadoop uses **Map reduce,** enabling it to divide query into smaller parts, and process in parallel. New nodes can be added without altering data format or existing applications. If a node is lost, work is automatically relocated to other node. Hadoop 2.X enables **Speculative execution** where if a data node is slow, master node **redundantly executes same task** on another instance of the & takes first output
   2. Hadoop Provides Fault Tolerance, as **in automatic replication of file** in other locations. It works with commodity hardware having high chances of node failure. Default replication factor of 3. **Name node uses** **Rack awareness** in deciding how to replicate blocks based on rack definitions. Any computations are attempted on original data only, with master node having info on node having original data. If this node doesn’t respond it's assumed failed, then node with replica is used.
   3. Hadoop is suitable **for large datasets in a single file**, rather than several small files, as metadata in latter case will be large, occupying space on name node
   4. **Check pointing** enables loading the final in-memory state directly from the fsimage and edit log delta instead of replaying entire edit log
   5. Hadoop has **a rich ecosystem consisting** of related projects i.e. Map reduce, Hive, Hbase, Zookeeper, HCatalog, Apache Pig. To handle **streaming data**, utilities like use of Apache Kafka, Apache Flume, and Apache Spark are provided
10. Hadoop 2.0 supports **Name node Federation** where Name node is partitioned into namespaces in order to reduce metadata storage requirements for cluster with large no. of data nodes, a bottleneck in Hadoop 1.0. Each federated Name node is **assigned a block pool**, list of data blocks maintained by a name node. In Hadoop 1.0 block pool was entire list of data blocks, as there was a single name node. A namespace further comprises of an **Active & Standby name node**: both having mappings to same data blocks at any time. **Edit logs get shared** through high available shared storage. **Fencing ensures active machine is brought down** if failover happens due to network delay even while active machine is still up
11. Allocation of containers to perform Map Reduce takes significant upfront time, hence in Hadoop 2.0, **app master has provision for uberization** i.e. scheduling tasks on single JVM for small data
12. In Hadoop 2.0 **YARN** splits up functionalities of resource management and job scheduling/monitoring into separate daemons i.e. **global ResourceManager (RM) and per-application Application Master (AM).** Each application has **its own AM instance** running in container process on slave node unlike job tracker that monitored all applications from Master node. **AM sends heartbeats to RM** based on which it RM assigns container resources to AM instance.

**S****etting up a multi node Hadoop cluster on VMs**

1. Hadoop Installation modes
   1. Standalone Mode in which Hadoop configured to run as Java process and HDFS not used
   2. Pseudo Distributed mode in which Hadoop daemon runs on local machine **simulating a cluster** (different JVM instances on same m/c).
   3. Fully Distributed mode is production mode in real scenario
2. Use 2 Ubuntu VMs, one for name node and a data node. Clone existing Ubuntu VM with Hadoop installed, setting it as name node, name UbuntuNN1. Created a clone of Ubuntu VM to be used as Data Node, naming it UbuntuDN1
3. Set hostname & IP Address of each VM
   1. specify static IP address in **/etc/network/interfaces file**, enabling same IP address every time VM boots
   2. **/etc/hosts** on each VM should contain **hostname and IP Address mappings for all VMs in cluster**, including self, enabling hostname of other VMs instead of IP address to ping /transfer data
   3. specify **hostname in /etc/hostname file** i.e. UbuntuNN1, UbuntuDN1 etc, enabling hostname usage in Hadoop configuration files instead of IP
   4. In "slaves" file on NN, specify hostname of each Data Node
4. Set Hadoop configuration on all nodes
   1. **Configure core-site.xml on name node**, replace default property value of <localhost:15XXX> with <namenode\_hostname:9000>, replicate same file on all data nodes
   2. **Configure hdfs-site.xml on name node,** set replication factor & name node directory here (e.g. file:/usr/local/hadoop/hadoop\_data/hdfs/namenode), whose path on Linux file system will have name node info. For data node, DN directory is specified (e.g. file:/usr/local/hadoop/hadoop\_data/hdfs/datanode), instead on NN directory. Specified directories to be then physically created on NN and DN nodes
   3. Other config files are identical on all nodes
   4. Set up an **SSH key on name node** and **copy it on all data nodes** through ssh-copy-id command as below, and then test connection through ssh
      1. ssh-copy-id -i ./.ssh/id\_rsa.pub hduser@ubuntudn1
      2. ssh ubuntudn1 //will not require password of node ubuntudn1, if key gets copied
   5. **Format name node** **&** **run start-dfs command** on name node host, Data Node services automatically get started on each Data Node
   6. Launching **Web UI of Name Node Manager** daemon should show live data nodes
5. Replication factor is minimum 1, which means there is a single copy of a file in HDFS, changing replication factor updates **replication copies of files subsequently added to HDFS**. Replication of old files can be changed through external command as below

*hadoop dfs -setrep -w 2 /user/hduser/gutenberg/5000-8.txt*

1. New Data Node can be added **without need to reformat name node** while updating slave file and restarting cluster

**Hadoop Map Reduce Framework**

1. Map/Reduce Process: Data nodes processes the assigned tasks, make a key value pair & return output to a reducer. Reducer collects key value pairs from multiple data nodes and combine into a final output
2. A **Job Tracker service** **running on name node** submits and tracks map reduce jobs splitting a job into tasks & assigning each task to a task tracker. **Task Tracker service running on data nodes**, manages execution of task and **sends periodic heartbeat to Job tracker**. If Job tracker does not receive heartbeat it assumes that task Tracker has crashed and reassigns its task to another task Tracker. If task restart fails more than specific number of times, Job Tracker kills entire job.
3. **HDFS Block Size** specifies minimum amount of data that can be read or written in an HDFS block (64MB by default). A **file larger than 64MB is split into multiple blocks** and stored as independent units across data nodes. If file size less than 64MB, remaining block space on disk is not wasted.
4. **Indexing on Hadoop:** Last part of data on a block contains info on where in cluster next block is stored
5. **A Combiner** is an optional **mini-reduce process** operating on data generated by mapper, to reduce network traffic between nodes. A **Partitioner** in Hadoop ensures a group of keys go to a particular reducer
6. Map Reduce phases
   1. Input splits: chunk of input consumed by a Map
   2. Mapping: data in each split passed to a mapping function to produce output values
   3. Shuffler: consolidates relevant records from mapping phase output
   4. Reducer: Aggregates outputs values from shuffling phase to return a single output value. Reducer before processing aggregates key value pairs having same key, making values for a key accessible through iterator. Hadoop architecture ensures, **key value pair having same key goes to same reducer**, to enable proper aggregation
7. Joins in Map Reduce
   1. Data comes from various sources and to get a holistic view of data it is required to perform Join
   2. For **map side join,** a Distributed cache is used. It is done when one of data sets is small to fit in memory e.g. abbreviation for all states. **Distributed cache is used to distribute a small dataset across all mappers** for reference. Adding file to a Distributed cache is done through Java command

*DistributedCache.addCacheFile (new URI("/abc.dat"), job.getConfiguration())*

* 1. To retrieve distributed file from Distributed cache use java command

*Path [] files = DistributedCache.getLocalCacheFiles (context.getConfiguration ())*

* 1. **Reduce side join** preferred when datasets involved can be large enough for memory. E.g. Customer names and transaction made at store, both can span multiple Hadoop blocks. Multiple input files need to be processed by Mapper, using JAVA commands below

MultipleInputs.addInputPath(job, new Path(args(0)), TextInputFormat.class, CustMapper.class)

MultipleInputs.addInputPath(job, new Path(args(0)), TextInputFormat.class, TxnMapper.class)

**HBASE**

1. HBASE is Open source **distributed column store** used in big data ecosystem, though not a complete substitute for HDFS when doing large batch Map Reduce. It is a good **candidate for a variable schema** with each row slightly different, which may also include metadata for some column values. It uses **Zookeeper for various distributed coordination services** such as **master election** and requires proper network services in place such as NTP and DNS as nodes in the cluster need **closely synchronized clocks** while referring to each other consistently.
2. **Monitoring** is important for successful Hbase operations as like distributed systems; Hbase is susceptible to **cascading failures**. Memory, CPU, I/O and network latency and bandwidth on each of Hbase nodes needs to be monitored
3. In HBASE there is a **column family & column qualifier**. Within a cell there can be multiple pieces of data, enabling **recording history on a given value using timestamps**. Tables are sparse & different rows may have different sets of columns i.e. while one Row key can have value for column (info:state), another Row key can have values for columns (info: state) and (info: city)
4. For data, **a key prefix** should be chosen **that distributes well** based on a use case, and **does not overloaded** specific servers. **Compaction impacts performance** & should be scheduled to run every day at an off-peak time rather than relying on auto compaction. Number of regions should be kept to a reasonable number. **HBASE and MR Jobs should run on separate clusters** in case they are independent
5. Hbase runs on master slave model. There are 2 types of processes - **master & region servers**. A table is broken into regions; different machines serve different regions. To achieve balancing, each region a table is broken into should be served by different region server. Master acts as a mediator i.e. performs operations like table creation, deletion etc. Master can become a **single point of failure**. **Zookeeper** takes care of this by electing new master, or moving regions from a region server to another
6. HBASE doesn’t work with SQL, and does not have an **optimiser for supporting cross record transactions or joins**. It should be avoided with apps having **SQL intensive queries.** Some SQL projects on Hbase exist like Phoenix, Tripodium
7. HBASE supports interaction through **JAVA, Rest API & thrift API**. While JAVA APIs are full featured, it requires client libraries to be installed and is susceptible to incompatibilities in case APIs change with HBASE versions. Rest API is easy to use but is slower & require Rest API server. Thrift API is lightweight and has multi-language support but has limited calls.
8. **Hbase architecture** comprises of a zookeeper quorum (HQuorum), HMaster and Region Servers. **Client communicates with zookeeper** while HMaster communicates with both zookeeper & region server. Based on zookeeper response client can know which region server to go to. **Master server (containing HMaster) governs region servers via HQuorum** peer which is also watcher, enabling client to go to appropriate location and get data. Region server works with HDFS client (that could be a java app) which then connects to data node in HDFS context. When data is stored in Hbase it is divided into no. of records coming in. Based on data that comes in, Hbase writes stuff into Hfile at end, before that it writes to **Hlog and stores same copy in memstore**. If there is insufficient data to flush, Hbase doesn’t flush data to Hfile. If Hbase goes down data is retrieved from Hlog. Only if **there is sufficient data in memstore** does it flush data to Hfile. **Hfile is a partition** in which all data with set of keys very closely knit is kept. At time of retrieval Hbase knows in which file it has kept data corresponding to particular keys. **HRegion server helps in storing data** in related contiguous manner and also in retrieving data effectively. Whatever data is read **gets into memstore** and from here multiple reads for multiple clients are served. Memstore and Hfile together **form an Hstore**. Hlog and entire Hstore becomes an HRegion
9. Hbase has HMaster i.e. **master node**, all **region servers are slave nodes**, having regions where file chunks get stored. Each region stores a set of rows, with row data separated into multiple column families. Groups of related data is stored in single CF. Particular CF data is stored in each Hstore. **Hstore consist of memstore & Hfiles**. Memstore tracks all log operations & stores data on HDFS ordered by key. It helps in efficient retrieval as Hbase can’t write to disk in real time. It flushes data at regular intervals **after sorting**, writes to HDFS using sequential writes. A new Hfile is created on each flush. Data consistency is ensured on node failure by writing **into Write Ahead Log** (Hlog) in HDFS
10. Map Reduce with HBASE: Hbase provides a **class for each Map Reduce phase** which enables it be used as a source or sink to map reduce job. TableMapReduceUtil class has static methods InitTableMapperJob & InitTableReducerJob
11. **Hbase supports random reads** as data is stored in Key Value pair, while Hadoop supports sequential data reads. Data in Hbase is maintained in **Write ahead logs** and **memtable**, and there is a Flushing mechanism for writing data to **hfiles on disk**. Since data is stored in sorted order and in multiple hfiles, we know where a row exists, thereby reducing read latency
12. Hbase permits keeping **multiple versions of same column data.**  A Row, Column Family, Version, and timestamp define a single cell. Hbase sorts data in **descending order of Key Value pair** with one having most recent timestamp coming first. KV wraps a byte array & takes offset lengths at where to start interpreting contents. **KV format** in byte array comprises of **key length, value length, key, value**. Key is further decomposed as {row length, row, column family length, column family, column qualifier, timestamp, key type (whether it got entered through Put, delete, deleteCol, deleteFamily operation)}
13. KV instance is not split across blocks, rather stored in coherent block i.e. if Value length is more than free space in current block it will be stored in next block. KV is for each column even if it has same row, and KV pairs are stored in columnar fashion**. Column families can be created on fly** giving Hbase flexible model. Table & column family are defined at structural level, rest are dynamic
14. **Hbase vs. Hadoop:** Hadoopsupports **sequential read**, while Hbase provides random read as data is in KV pair, allows multiple versions of column data. A single cell is defined by a **row, column family, version & timestamp**. Data is sorted in descending order of key value pairs to retrieve latest rows
15. A Key value is wrapped as a byte array, takes offset of length into passed array to start interpreting contents as KV. KV format in byte array contains **key length, value length, key, value.** Key is further decomposed into row length, row key, Column Family length, Column Family, column qualifier, timestamp & key type (i.e. whether key has gone in through a PUT, Delete, Del column, del family). KV is read as coherent block i.e. not split across blocks. Hbase stores **KV pairs in a columnar way** enabling low latency during reads. Only tables and CFs are defined at structural level, **columns can be added on fly.**
16. **Hotspot in Hbase**: This problem generally occurs with time series kind of data, where new data is added frequently with new key values getting created i.e. intraday stock quotes. Even with a multi node cluster one node is busy while others remain idle due to row key design, such that data coming is partitioned w.r.t. a time interval ranges i.e. from 12.00PM to 3PM all data will land on Server1
17. **Compactions:** In Hbase optimized read performance comes from **having only one file per Column Family** but this is not possible during periods of heavy writes. As part of compaction Hbase **tries to combine Hfile to reduce the maximum number of disk seeks** needed for a read. It involves Reading Key Values in input files, writing out any Key Values that are not deleted, are inside of the time to live (TTL), and don’t violate the number of versions. Trade off with compaction is it involves disk IOs which is costly operation. Hbase supports several user supplied compaction policies
    1. **FIFO compaction:** doesn’t do any compaction, just tracks store files and completely deletes files that expire. If FIFO compaction is set on a Column Family and TTL set to 24hrs, after TTL expires first file that is older will get deleted. Good if it is needed to **store huge amount of incoming data but only for a limited period**
    2. **Size Tiered compaction**: can be used for compressed & aggregated data. It doesn’t preserve temporal locality as it combines data which is recent with data that is old when performing major compaction. It merges 4 similar sized Hfile into one. When compaction finishes all data from block cache is evicted. It involves significant IO overhead and also impacts performance.
    3. **Date Tiered compaction:** can be a choice for compressed & aggregated data, offers better performance than size tiered. It is also ideal for time series kind of data. Files are **grouped in windows based on how old the data is** in the file and then that window is compacted i.e. all files written in last one hour can be compacted into one file
    4. **Delayed compaction:** allows specifying delay for column families, all new created store files will be eligible for compaction if they are older than specified delay. Mostly data is accessed for last 24 hrs. In a blockage caused during compaction data from these files will always be accessible
    5. **Major compaction:** merges all Hfile into one
    6. **Minor compactions**: combine a configurable number of smaller HFiles into one larger Hfile as per configuration. They are important for reducing disk seeks for reading a row. They do not remove the delete flags and the deleted cells
    7. **In memory compaction**: exploit redundancies in workload to eliminate duplicates in memory. If same key is updated again there are **multiple versions in memory**. On deletion of key there will be tomb stones. In such cases there is gain on in memory compaction. Instead of writing on disk and compacting, **in memory flush is performed** keeping aside compacted memory segments

**Zookeeper**

1. **Zookeeper is a coordination service** for a distributed application, taking care of all hard stuff inside DA such as synchronization, serialization and other nitty-gritty side of DA such as partial failure, race conditions, networking issues. A zookeeper cluster **(ensemble) comprises of an odd no. of voting nodes** having a leader node and follower nodes with an observer node which is a non-voting node. Writes are handled by leader and committed to follower
2. Zookeeper provides **basic admin commands** to check statistics such as conf, ruok etc. Zookeeper APIs are in C & JAVA with external contributors making some available in REST, python & perl. **For each API method there is sync & async signature** with async taking in call back handler as parameter
3. **Watch events** are associated with read ops like getData(), exists() and getChildren(), which **get triggered** when node is created, deleted or has its data updated, or any of its children are created or deleted

**Apache Drill**

1. Drill is a distributed SQL query engine, providing connectivity and query options for data on hadoop distributions. It allows querying self-describing data like json and uses a single SQL query i/f for multiple data types including Hbase, json etc. Drill has an execution engine that uses storage plugin libraries for files, Hbase, hive, mongo etc.
2. Drill comprises of Drill bit that is to be installed on all nodes in Hadoop cluster. Though It is not necessary to install drill bit on all nodes but that gives it ability to **achieve data locality at query execution time** (push down process to node where data lives).
3. SQL query comes from client and is accepted **by a drill bit that acts as a coordinator** for that request. Zookeeper quorum could be used to route request to specific Drill bit node. Once Drill bit receives query request it determines best way to execute that query. It makes rule based / cost based optimizations during query planning. It splits plan into query plan fragments. Coordinator Drill bit along with zookeeper determines what all Drill bit nodes are available and capable of executing Query Plan fragments It then distributes to other Drill bit in cluster. These Drill bits return results to coordinator Drill bit that returns it to client. During execution each of Drill bit could interact with underlying file system (Hbase, hive etc.). There is no master slave architecture
4. Apache Drill is **columnar in storage** as well as in memory. It allows CPU to operate on vectors or record batches. Drill does **not use Map Reduce or spark**. It assumes probability of hardware failure to be small during short execution of a query, **so it does as** **much as possible execution in memory** without writing to disk for checkpointing or recovery purposes. Model is based on pipeline execution, scheduling all tasks at once. Drill allows generating highly efficient custom code for every query dynamically

**AirBnb Big Data Infrastructure**

1. Data Sources: data comes into system from 2 sources
   1. Event logs coming from website or mobile device, giving info on how user behaves at website. Even experiments go through event logs i.e. creating fictitious use cases
   2. MySQL data dumps, containing info that needs to be loaded into data warehouse for downstream analysis
2. There are 2 Hadoop clusters in Data Warehouse – Gold cluster and Silver cluster. Silver cluster serves as backup of Gold cluster. As per user experience HDFS is not completely reliable, and users have reported instances of missing blocks. Second cluster also isolate important jobs from ad hoc workload. There is an open source in-house **replication service "ReAir"**, creating **one way sync from gold to silver cluster**. S3 is also leveraged to do additional backup of raw data, also important data is archived to s3 after some time period. **Spark cluster and Presto cluster** serve as computational workload clusters, reading data from either Gold or silver cluster in Data Warehouse. There is a **java scheduling service Airflow**, and reporting services **Caravel & Tableau.** For interactive query **Airpal** is used.
3. **Streaming Architecture in AirBnb:** Event log is already in streaming form and it comes to Kafka as stream. MySQL data is not in streaming form so backup/restore is used, combined with **Spinal Tap,** an in-house listener. This service listens to DB change and writes it to Kafka. Centrepiece of cluster is Hbase where all streaming data is tied with other data from warehouse
4. **Hbase suitability to use as a state store**: It is reliable and scalable to 100 Terra Bytes of size. It Integrates with Hadoop system very well i.e. has Spark connector, Hfile input format that enables querying **Hbase directory through Hadoop hive system.** Hbase provides rich functionalities i.e. update, bulk upload, counter. It is easy to manage: through **cdh cloud era manager**, a few clicks can set up Hbase running. Operations set up with Hbase are Full table scan, Key/prefix lookup, Normal update, Bulk upload and Simple aggregation
5. As input data is received, based on key look up from Hbase is performed, if state is found in Hbase its value is retuned else null is returned. Based on result it is decided what new state would be written to Hbase, and Hbase is updated
6. **Key space design consideration:** hash partitioning is used to do load balancing. Composite key to support Key lookup, support prefix based scan on top of this schema
7. **Write performance consideration:** partitions are created based on key before write so that key value pair goes to a pre-designated region server. For large volumes of data, bulk upload is used as it generates Hfile directly and loads into Hbase
8. **Real-time data ingestion in AirBnb**: Streaming data coms from Kafka and spark & lands into Hbase. Presto/Hive/Spark query Hbase in real time, take a snapshot from Hbase into HDFS either hourly or daily. Some kind of table mapping to unify all these snapshots and real-time data, so best job, interactive job can be queried through view. It is important to have an efficient snapshot mechanism for dumping data from Hbase to HDFS. Normally **Hfile input format** is used to dump snapshot. Users will go against each region and scan data to dump to HDFS. Dump to HDFS is important since many of down streaming pipeline analysis rely on data store on HDFS and also dumped data can be archived and saved offline. To populate Hbase initially it is done through snapshot & for that **bulk uploader** is used. To speed up snapshot, Hbase has a feature, through which one can take Hfile snapshot and then restore snapshot to HDFS location. Then Hfile snapshot input format can be used to build a view on top of restored data, enabling direct query against Hfile, improving performance.

**Hive**

1. Hive system deals with **files like tables in DB**, including partitions. It Stores files in Hadoop, and metadata in a JDBC compliant DB (normally MySQL that has metadata store). **Metadata** store is a system catalog through which Hive interacts with DB getting database statistics. It has a Serialize/De-serialize mechanism where it converts CVS, Thrift, RegEx APIs to communicate with metadata store. It uses thrift to execute HQL statements.
2. **Data in HIVE is read using HQL**. HQL has a Parser, Planner, Optimizer which together help in execution. It supports DDL, DML and query language. Hive processes Text, Sequential files and Row Columnar files. Also supports UDFs and User Defined Application functions (UDAF) and user defined MR scripts. Hive has CLI & JDBC/ODBC connectivity.
3. Data is structured into **tables, columns, rows, partitions, buckets** etc., each level being a subfolder in HDFS. Tables have columns and Data type for cols, including serialize and de-serialize info. Tables can handle array, maps, and json data. **Partitions** separate data into range of values. They have own columns, data types, serialize and de-serialize info. Partitions are subdivided into small data ranges i.e. hash partitions, helping in sampling and joins. **Warehouse directory** on HDFS is flexible as it can store table as flat files with partitions and buckets and Database with tables
4. When a hive table is created, it is a **blank folder in Hadoop**. To insert data into a hive table on HDFS from local file system use command

*LOAD DATA LOCAL INPATH './moviedata/u.data' into TABLE usermovieratings;*

1. In case of **text file data is a blob** i.e. whatever data is given, it gets stored. For **sequential** **file internally hive generates a key** with which it can uniquely identify each record. It gets added to each row that helps in doing things faster. In **row columnar** form, row is stored vertically like a column rather than horizontally. Selections can be done quicker in this form. Table is created as type RCFILE. **Optimized RC form** is further optimization over original row columnar form for better retrieval. Table gets created as type ORCFILE
2. For insertions in sequential and RC files use command

*INSERT OVERWRITE TABLE usermovieratings\_seq SELECT \* FROM usermovieratings*;

1. Partitioning is segregating row to make it retrieval easy. A **Partitioned table** in Hive is created using command

*create table usermovieratings\_txtpart (*

*movieID int,*

*ts string)*

*PARTITIONED BY (userID int, movierating int)*

*CLUSTERED BY (movieID)*

*SORTED BY (timestamp) INTO 5 BUCKETS*

*row format delimited*

*fields terminated by '\t'*

*lines terminated by '\n'*

*stored as TEXTFILE;*

**PARTITIONED BY** clause contains those columns on which table created is partitioned

**CLUSTERED BY** clause does grouping by specified field

**SORTED BY** specifies how or on which field grouping is to be sorted and into how many buckets

1. A text file cannot be directly loaded in partitioned table as in normal case. Data should be in another table, and then **insert overwrite** is to be performed that table. Within a partition, a specified numbers of buckets get created, and **values get sorted and stored in these buckets** w.r.t specified field. There may be empty buckets towards end if no. of elements in partition is less than no. of buckets.

*INSERT OVERWRITE TABLE usermovieratings\_txtpart PARTITION(userid=93, movierating = 5)*

*SELECT umrs.movieId, umrs.ts FROM usermovieratings\_seq umrs*

*WHERE umrs.userid=93 and movierating = 5;*

1. Hive supports compact and bitmap indexes to speed query execution

**Sqoop**

1. Sqoop is a utility, enabling **data import from RDBMS to HDFS or vice versa**. It also enables data import to Hive DB. For tables to get imported from RDBMS they should have **a unique index**. Sqoop has been written on top of Map reduce. By default it works with **4 mappers & zero reducers**, which means output is directly given by mappers. No. of **mappers can be configured** through command line argument based on available system or cluster configuration
2. Sqoop allows importing **entire table, a part of table** using “where” or “column” clause or importing **all tables** from a specific database
3. Sqoop allows import of data from RDBMS to an existing table in Hbase or it creates a new table in Hbase loading in RDBMS data

*sqoop import --connect jdbc:mysql://localhost/test --table mysqltable --hbase-table newtable --column-family cf --hbase-row-key id -m 1*

*sqoop import --connect jdbc:mysql://localhost/test --table mysqltable --hbase-table newtable1 --column-family ntf* ***--hbase-create-table*** *--hbase-row-key id -m 1*

**Chef**

1. Chef is an **automation platform** that configures & manages infrastructure on premise of in cloud. It can be used to speed up application deployment and create continuous delivery pipeline. It enables **automating web infrastructure** letting a centralized configuration to spin up configured web servers very fast. It also creates **development & test environments** that are exact **replicas of production.** Chef comes as Open source chef and enterprise chef flavour
2. Chef turns infrastructure into code. It has several standard infrastructure configurations and **tasks defined in cookbooks** that are base units for configuration. These tasks are referred to as **recipes**, programs that get run when a cookbook is installed. Cookbooks are created on workstation and safely stored in repository as structured data in JSON. A command line utility **Knife** runs on workstation that allows **sending** **updated cookbook settings to chef server**
3. Chef components include **chef server, workstation & nodes** Chef Server manages **nodes that make up infrastructure**. Chef client runs on each node. A node can be anything in computing environment i.e. a laptop, VM, server etc. **Chef client pulls chef server periodically** to see if there is any change to cookbooks or settings & chef server sends latest versions to client which applies configuration changes to node. **Work Station** is machine where **DevOps admin writes infrastructure code** such as cookbooks, recipes etc. & then uploads on server. Chef client has a component OHAI that discovers machine specific data i.e. IP, CPU, RAM etc.
4. Setting up infrastructure with Chef
   1. On Linux, **chef server and chef client packages** are available for downloads as RPMs. After downloading, chef server is configured through command **chef-server-ctl reconfigure**. It starts several services on chef server, can be checked though **chef-server-ctl status**
   2. Next configure chef workstation on a different machine. Install chef package on workstation. **Copy certificates from chef server to workstation** to make workstation secure. Certificates by default are located on server in **/etc/chef-server**, files admin.pem, chef-validator.pem, chef-webui.pem. These files should be copied on workstation to a folder .chef
   3. Configure knife on workstation so that it should be able to communicate with chef server. Execute commands from .chef folder on workstation.

*knife ssl fetched*

*knife configure -i*

This configures workstation successfully. It **creates a trusted\_certs folder**, which contains certificate for server

* 1. Next **install and configure nodes**. Install chef package on node & copy file chef-validator.pem from server. Fetch certificate from server through knife

*knife ssl fetch -s https://chefserver.example.com.*

It gets fetched into trusted\_certs folder on node

* 1. Join this node to server now

*chef-client -S https://chefserver.example.com -K /etc/chef/chef-validator.epm*

* 1. Running knife command from workstation lists nodes that have joined server

*knife client list*

This can also be checked from **web interface to chef server**

1. Creating a cookbook and recipe
   1. On workstation execute command

*knife cookbook create <cookbook\_name>*

It creates entire cookbook structure on workstation

* 1. In recipes folder under cookbook folder structure open/edit an existing recipe file or create new one
  2. Test recipe using knife command

*knife cookbook test <cookbook\_name>*

* 1. Upload cookbook on server

*knife cookbook upload <cb-name>*

Cookbook presence on server can be verified through **server** **web interface**

* 1. To apply cookbook on a node, from Server Interface add cookbook to **run list on node**
  2. This cookbook will now get periodically pulled onto node by chef client. It can also be explicitly applied from node through **chef-client** command

1. Chef **provides built in knife plugins** for cloud hosting platforms i.e. knife rackspace, knife azure etc. These plugins can be used to create & configure nodes with chef in cloud

knife rackspace sever create -r 'role[webserver]' -N chef.myserver.org -S chef.myserver.org -f 3 -l 125

Knife contacts hosting provider to create a server node, **chef bootstrap** installs chef on this server, giving it configuration information to talk to chef server. Then it installs chef client that talks to chef server, pulls cookbooks and installs them

1. To work with **Hosted version of chef server**, an account is required with opscode (manage.chef.io). This allows using chef server copy that is hosted with opscode rather than having a local server. Up to 5 nodes can be configured to work with hosted server copy for free. Steps to work with hosted chef server are as below
   1. Download **chef dk** and install on Linux system that will be used as chef Workstation
   2. On **manage.chef.io site** create a new organization that comprises of a cluster of machines having similar policies. Policies are run lists to be implemented on a node. On Organizations tab there is a starter kit that should be downloaded and set up on work station
   3. Cookbooks available at **chef supermarket** can be downloaded & installed on chef server. Cookbook is downloaded as a tarball, extracted & put it in cookbooks folder on work station and then **uploaded to chef managed account.** Below commands are executed from chef workstation to download from supermarket and upload to server

*knife cookbook site download hadoop\_cluster*

*tar –zxvf hadoop\_cluster.tar.gz –C cookbooks*

*knife cookbook upload* hadoop\_cluster

* 1. Now this Cookbook should be applied on to a node. Node doesn’t know where chef server is and **so it needs to be bootstrapped**, which is to be done from work station

*knife bootstrap <node ip address> --ssh-user <username> --ssh-password <password> --sudo --use-sudo-password --node-name cnode1 --run-list 'recipe[hadoop\_cluster]'*

1. One of problems reported with using chef is **environment is not always consistent**, that may result in **errors while applying cookbooks on nodes** i.e. version change of software on repository could lead to conflict with commands “yum update” or “sudo apt get”. To handle this recipes may need to be modified to get specific versions for some local repository or one that is within control

**Docker**

1. Docker is a tool designed to make deploying & running applications through containers. Containers have apps running in them, with all dependent bins/libs packaged in them as well. Docker containers are light weight alternatives to VMs. Unlike VM there is no need to pre allocate any ram or disk space. Docker containers work on Linux systems only. On windows system, a Linux VM is required to run docker. Docker ensures identical environment in development and production systems.
2. A Docker file produces **docker images**, which will contain dependencies required by an application. Docker container is **a runtime instance of image**. Image is uploaded to **docker hub**, from where they can be pulled from public repository by various teams ie production & QA who prepare their own containers. a Images
3. Docker hub is cloud hosted service provided by docker. It has a Git repository for images and contains public /private repositories. Images are huge in size and require lot of network bandwidth. Alternatively a **CI server such as Jenkins** is used to build an environment that contains all **dependencies for a particular application** or Micro service. That build environment is deployed on to various teams i.e. testing, staging and production. Docker file written by developer and code is pushed onto git repository. CI server pulls hat code from git and builds environment for that particular Micro service, which is then deployed on testing, staging, production.
4. A user can sign in to docker hub with login credentials: Front page has options for **creating repositories and explore repositories** that contains all repositories available publicly. Docker images are read only templates used to create containers. They contain dependencies for particular applications. It is possible to build a container from more than one image
5. **Docker registry** is storage for docker images; they can be stored in public/private repositories. Repositories can be **present either locally or on cloud** in Docker hub. A container can be prepared by pulling image from pub repository. User can also write an image and upload to docker hub in its repositories.
6. Installing docker container on Ubuntu box
   1. before installing docker, there is need to install recommended packages

*sudo apt-get install linux-image-extra-$(uname -r) linux-image-extra-virtual*

* 1. After installing packages, install docker

*sudo apt-get install docker-engine*

* 1. Start docker service

*sudo service docker start*

* 1. Pull an image from docker hub i.e. centos image

*sudo docker pull centos*

* 1. Start centos container

*sudo docker run -it centos*

* 1. **Docker compose** enables running multiple applications on different containers with a single command. It requires pre-setting an environment

*sudo apt-get install python-pip*

* 1. Install docker compose now

*sudo pip install docker-compose*

* 1. Edit docker **compose.yml file** as per requirement

*sudo gedit docker-compose.yml*

* 1. Pull images & build containers specified in yml file

*sudo docker-compose up -d*