Moving Data into Hadoop

MODULE - I Load scenarios.

→ After putting bigdata into hadoop — main question is related to data life cycle & data movement.

→ automate the flow of this humongous amount of data

→ (3) diff scenarios for loading data.
- (1) Data ~~from~~ at rest
- (2) Data in motion
- (2) data from a web server or database log.
- (3) data from data warehouse.

(1) data that is already in a file in some directory
- no additional updates are planned on this data & it can be transferred as it is.
- transfer using - HDFS shell commands, cp, copyFromLocal, put

(2) Data in motion:
- continuously being updated.
- New data might be added regularly to these data sources
  - → Data might be appended to a file.
  - → Discrete or different logs might be getting merged into one log.
- lots of servers generating log files or other data
- ex. → A file that is getting appended
  - → A file that is being update.

- Data from multiple locations that need to be merged into one single source.

ex: Merge all log files from one directory into one file.

- Examples of data in motion include:
  - → Data from webserver such as a websphere application server.
  - → Apache web server data in database server logs or application logs.

(1) Data from a Data Warehouse / RDBMS.

- export the data and then use hadoop commands to import the data
- Using Sqoop.

(2) Data from webserver using flume or streaming tool like storm, chukwa.

⇒ flume - Developed by cloudera

- (3) tiered distributed service for data collection & possibly processing of data that consist of logical nodes.

- 1st tier - agent tier has flume agent installed at the sources of the data.
- This agent send their data to the 2nd tier - collector tier
- 2nd tier aggregates the data & forward it to final 3rd tier - storage tier such as HDFS.

- Each logical node has a Source and a Sink.

tells from where to collect the data	where the data is to be sent

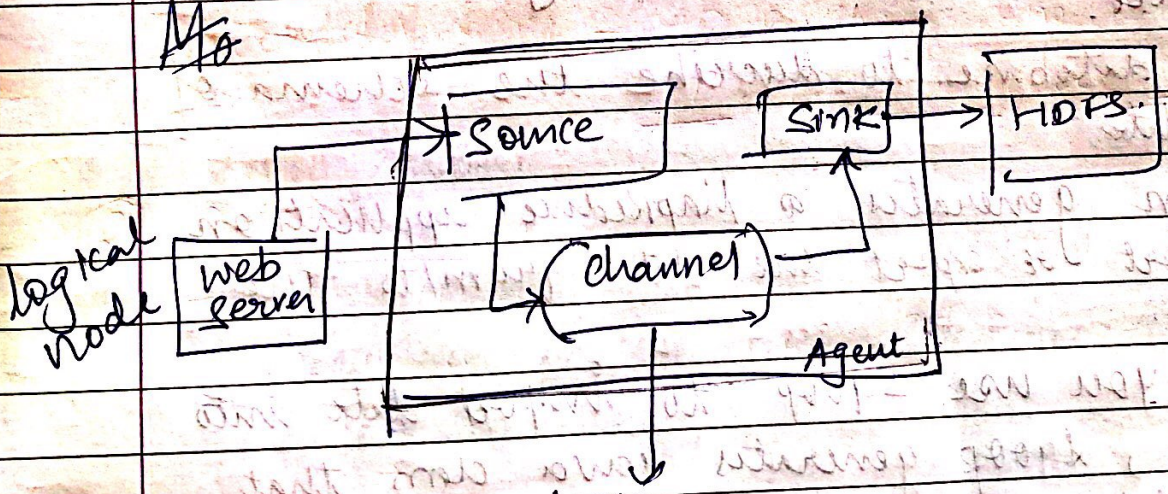
- flume uses the concept of physical node.

Single java process running on one machine in cluster as a single JVM.

(5) data from webserver?

⇒ Alternate approach here would be to use JMX :- java Management Extension commands

Ans



- eg, in general reading speed is faster than the writing speed.
- buffer to match read & write speed differences
- intermediary storage that stores the data being transferred temporarily & therefore prevents data loss.

MODULE - II Using Sqoop

→ Tool designed for efficiently transferring bulk data between Hadoop & structured data store such as relational databases.

→ Command line interface for transferring a data.

→ Sqoop can be used also to populate tables in Hive or HBase.

→ Sqoop is designed to transfer data between relational database systems and hadoop.

→ uses JDBC to access the relational systems

→ JDBC driver jar files for relational database to be accessed into the $SQOOP_HOME/lib so that driver can used by the sqoop software.

→ uses database to describe the schema of the data

→ It then generates a MapReduce application to import or export the data from/to the database.

→ When you use sqoop to import data into hadoop, sqoop generates java class that encapsulates one row of the imported table.

→ you have ~~actual~~ access to the actual source code for the generated java class.

→ This can allow you to quickly develop other MapReduce applications that use the records that Sqoop stored into HDFS.

→ Database connection requirements are the same for
- import
- export

→ you need to specify:

- JDBC connection string
- Username
- Password

ex: sqoop import/export --connect jdbc:db2://your.db2.com:50000/yourDB --username db2user --password yourpassword

→ connecting to DB2 system that listen to on port 5000 & is running on a system with hostname of

userid of db2 user

password of ~~db2~~ db2password.

* sqoop import:

→ Imports data from relational tables into HDFS.
- each row in the table becomes a separate record in HDFS.

→ Data can be stored as
- Text files
- Binary files
- into HBase
- into hive.

→ Imported data
- Can be all rows of a table
- Can limit the rows & columns
- Can specify your own query to access relational data

→ If you want to specify the location of the imported data,

ref, [--target-dir] argument

otherwise target directory name will be same as the table name.

* examples: Import statement

→ import all rows & columns from a table:

```
sqoop import --connect jdbc:db2://your.db2.com:5
  yourDB \ --username db2user
           --password db2password
           --table db2table \
           --target-dir sqoopdata
```

→ Sqoop imports the data in parallel.
→ You can override the no. of mappers that sqoop is to use.
→ the default is 4.
→ To split the data across multiple mappers, by default, sqoop uses the primary key of the table.
→ It determines the minimum & maximum values for the key & then assumes an even distribution of values.

List - databases , list - tables

→ import - all - tables.

→ you can use the `--split-by` argument to have the distribution work with a different column.

→ If the table does not have an index column, or has a multi-column key, then you must specify the `--split-by` column parameter

→ Additional Parameters

⇒ e.g. --split-by tbl-primary key.

⇒ -- columns "empno, empname, salary".

→ To only import data from subset of columns.

⇒ -- where "salary > 4000"

→ to limit the rows.

⇒ -- query 'SELECT e.empno, e.empname, d.deptname FROM employee e JOIN department d on (e.deptnum = d.deptnum)'.

→ supply your own query.

★ examples: export statement

→ exports a set of files from HDFS to a relational database system

- Table must already exist
- Records are parsed based upon user's specifications.

(1) → Default mode is "insert"
- insert rows into the table.

(2) → update mode.
- generates update statements
- Replaces existing rows in the table.

* Syntax: --update-key argument

which column or comma separated columns.

(where clause) to specify where to update.

→ missing rows are not inserted
→ Not detected as an error.

③ Call mode:
→ makes a stored procedure call for each record.

④ → --export-dir
- specifies the directory in HDFS from which to read the data.

* Examples:

① Basic export from files in a directory to a table:

```
sqoop export --connect jdbc:db2:
    //your.db2.com:5000/yourDB \
  --username db2user
  --password db2password
  --table employee \
  --export-dir /employeedata/processed.
```

Page No.:
Date:
YOUVA

(2) Calling a stored procedure:

```
Sqoop export --connect jdbc:db2://
    your.db2.com:50000/yourDB \
    --username db2user
    --password db2password
    --call empproc \
    --export-dir /employeedata/procesed.
```

(3) updating a table:

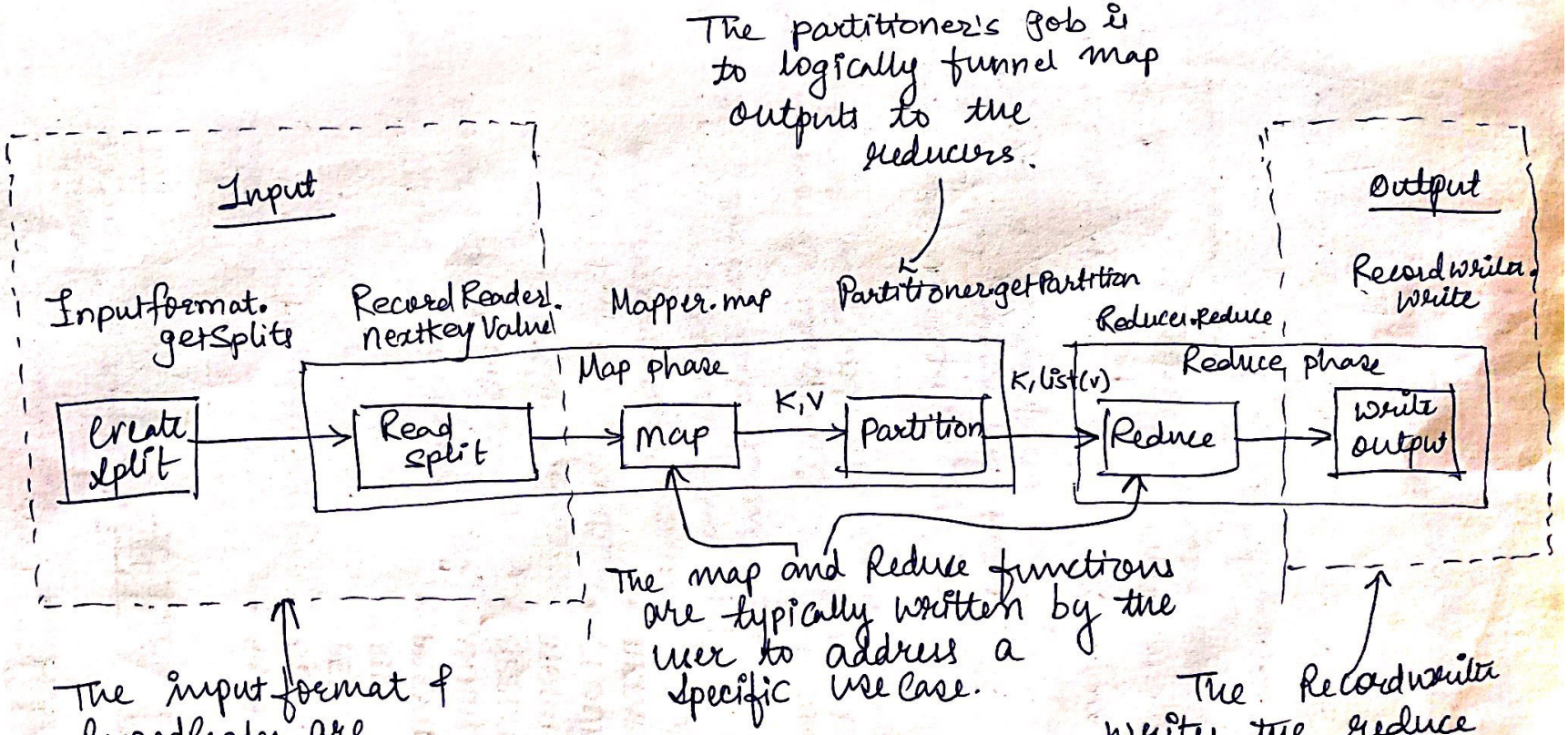
```
Sqoop export --connect jdbc:db2://
    your.db2.com:50000/yourDB \
    --username db2user
    --password db2password
    --table employee \
    --update_key empno
    --export-dir /employeedata/procesed.
```

```
Sqoop export --connect jdbc:mysql://localhost/
    employees. --username ~~hadoop~~ hdus
        --table emp
        --export-dir /user/hdus/employee.
```

* Inputs & outputs in MapReduce:

→ multiple data formats available.

eg. xml files
text log files



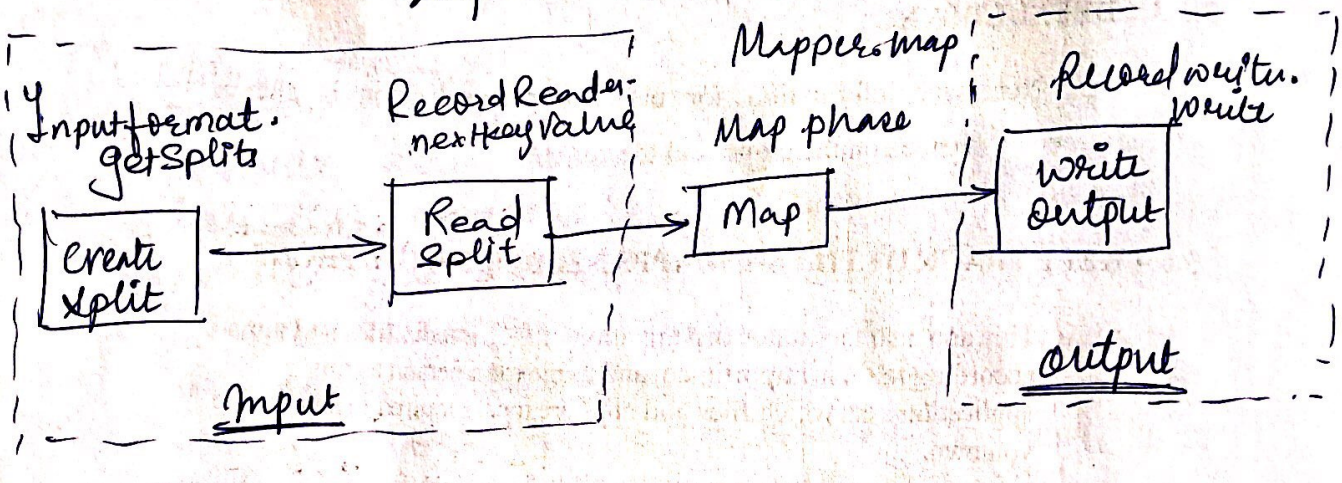
The partitioner's job is to logically funnel map outputs to the reducers.

The input format & RecordReader are responsible for determining what data to feed into the map

The map and Reduce functions are typically written by the user to address a specific use case.

The RecordWriter writes the reduce output to the destination data sink, which is the final resting place of this MapReduce data flow.

* Data inputs : — Input format — This class is consulted to determine how the input data should be partitioned for the map task.
— Record Reader. Reading the data from inputs.



⇒ (1) Input format:

- Every ~~Record~~ job in MapReduce must define its input according to contract specified in the Inputformat abstract class.
- must fulfill (2) contracts
  - (i) describe type information for map input key and values.
  - (ii) how the input data should be partitioned.
  - (iii) indicate the RecordReader instance that should read the data from source.

(2) Record Reader:

- This class is used by MapReduce in the map tasks to read data from an input split and provide each record in the form of a key/value pair for use by mappers.
- A task created for each input split, and each task has a single RecordReader that's responsible for reading the data for that input split.

* Data output:
  - ~~Data output~~ output format → Basic validation of the data sink properties
  - Record writer → write each reducer output to the data sink.

(1) Output format:

- Contracts that implementers must fulfil, including checking the information related to the job output.
- It provides a record writer and specifying an output committer
- which allows writes to be staged and then made "permanent" upon task and/or jobs.

(2) Record writer:

- To write the reducer outputs to the destination data sink.