Data Science and Analytics

MapReduce, YARN, Hive and Pig

The notes for the lesson “MapReduce, YARN, Hive and Pig”: chapters 6.3 to 6.4, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

2 chapters | Chapters 6.3 to 6.4

About these notes

These are the notes for the University free n easy video lesson **MapReduce, YARN, Hive and Pig** on Data Science and Analytics: chapters 6.3 to 6.4, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

6.3 MapReduce programming model

MapReduce is a programming model for processing huge data sets in parallel on a cluster. The programmer writes only two functions: **map**, which turns each input record into key–value pairs, and **reduce**, which combines all the values that share a key. The framework does everything else: it splits the input, runs many map tasks next to the data, sorts and moves the intermediate pairs to the reducers, re-runs tasks that fail and writes the result to HDFS. Jeffrey Dean and Sanjay Ghemawat described it at Google in 2004; Hadoop MapReduce is the open-source version. Counting, grouping, joining and indexing large files all fit this pattern.

You need: Chapter 6.2 (HDFS blocks and DataNodes).

Map, shuffle and reduce

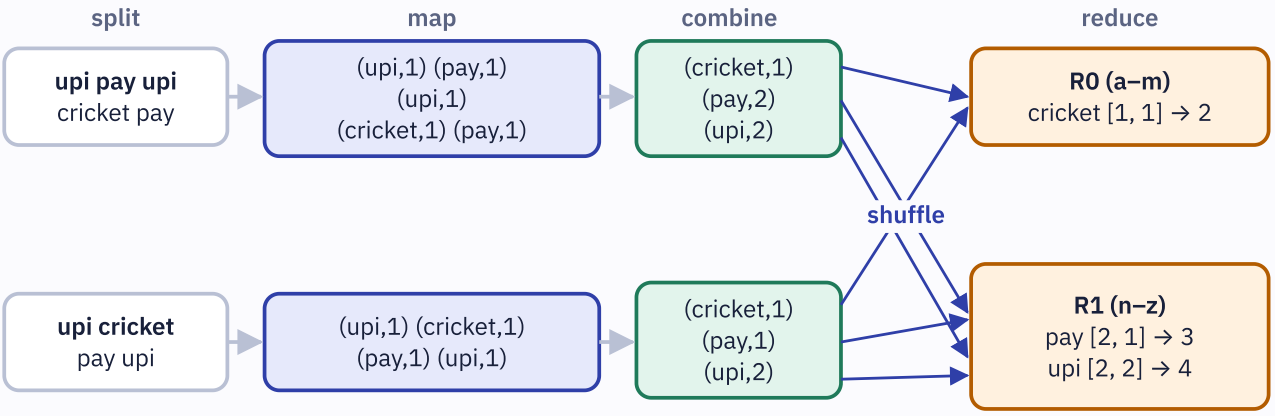
The two functions have fixed shapes:

```
map:    (k1, v1)        → list of (k2, v2)
reduce: (k2, list of v2) → list of (k3, v3)
```

For a text file, Hadoop gives map the byte offset of a line as k1 and the line itself as v1. A job runs in these phases:

1. **Split**: the input is divided into **input splits**, normally one per HDFS block, and one map task is started per split. A RecordReader turns the split into records.
2. **Map**: each map task calls map() once per record and collects the output pairs in a memory buffer.
3. **Partition**: each pair is assigned to one of R reducers, by default partition = hash(key) mod R. All pairs with the same key reach the same reducer.
4. **Combine** (optional): a **combiner**, a mini-reduce run on one mapper's own output, adds up pairs locally to cut network traffic.
5. **Shuffle and sort**: each reducer fetches its partition from every mapper; the framework merges the pieces and sorts them by key, so each key arrives once with the list of all its values. This is the most expensive phase, because data crosses the network.
6. **Reduce**: reduce() is called once per key. Output goes to HDFS as files named part-r-00000, part-r-00001, and so on, one per reducer.

The scheduler tries to run each map task on a node that holds a replica of its split (**data locality**), or failing that in the same rack. Map output is written to the mapper's local disk, not HDFS, because it is temporary. If a task fails it is simply re-run on another node; if one task is unusually slow (a **straggler**), a duplicate copy may be started and the first to finish wins (speculative execution).



Pairs over the network: 9 without the combiner, 3 + 3 = 6 with it.

Output: cricket 2, pay 3, upi 4 (9 words in, 2 + 3 + 4 = 9 counted).

A 10 GB log: 10,240 ÷ 128 = 80 map tasks; 40 at a time gives 2 waves.

Figure 6.3.1 Each mapper emits (word, 1), its combiner adds locally, and the shuffle sends every pair for a word to one reducer: cricket to R0, pay and upi to R1. The combiner cuts the shuffle from 9 pairs to 6.

Word count in Python with Hadoop Streaming

Hadoop Streaming runs any program that reads standard input and writes standard output as a mapper or reducer, with key and value separated by a tab. The mapper in Python:

```
# mapper.py
import sys
for line in sys.stdin:
    for word in line.lower().split():
        print(f"{word}\t1")
```

The reducer relies on the sort: all lines for one word arrive together, so it adds until the word changes.

```
# reducer.py
import sys
current, count = None, 0
for line in sys.stdin:
    word, n = line.rstrip("\n").split("\t")
    if word == current:
        count += int(n)
    else:
        if current is not None:
            print(f"{current}\t{count}")
        current, count = word, int(n)
if current is not None:
    print(f"{current}\t{count}")
```

You can test the whole model on one machine with a pipe, where `sort` plays the shuffle: `cat input.txt | python3 mapper.py | sort | python3 reducer.py`. On the cluster the same two files are passed to the hadoop-streaming jar with `-mapper`, `-reducer`, `-input` and `-output`. In Java, the mapper extends `Mapper<LongWritable, Text, Text, IntWritable>` and the reducer extends `Reducer<Text, IntWritable, Text, IntWritable>`.

Combiners, and where MapReduce is a poor fit

A combiner is allowed only when the operation gives the same answer however the values are grouped, that is, when it is commutative and associative: sum, count, maximum, minimum. Average is not: avg(avg(1, 2), avg(3)) = avg(1.5, 3) = 2.25, but avg(1, 2, 3) = 2. The fix is to emit (sum, count) pairs, combine those, and divide only in the reducer. The framework may run a combiner zero, one or several times, so the job must be correct without it.

Every MapReduce job reads from HDFS and writes back to HDFS. Algorithms that loop, such as k-means clustering (an earlier lesson) or PageRank, must chain one job per iteration and pay

that disk cost each time. That cost is why Spark (Chapter 6.6) replaced MapReduce for most new work.

WORKED EXAMPLE

Question. The input file holds four lines: “upi pay upi”, “cricket pay”, “upi cricket”, “pay upi”. Lines 1–2 form split 1 and lines 3–4 form split 2. Run word count with a combiner and 2 reducers, where the partitioner sends words starting a–m to reducer 0 and n–z to reducer 1. (a) Show the map output, combiner output, reducer input and final output. (b) How many pairs cross the network with and without the combiner? (c) A 10 GB log is processed with 128 MB splits on a cluster that runs 40 map tasks at once. How many map tasks and how many waves? (8 marks)

STAGE	SPLIT 1 SIDE	SPLIT 2 SIDE
Map output	`(upi,1) (pay,1) (upi,1) (cricket,1) (pay,1)`	`(upi,1) (cricket,1) (pay,1) (upi,1)`
Combiner output	`(cricket,1) (pay,2) (upi,2)`	`(cricket,1) (pay,1) (upi,2)`
Reducer input	`R0: (cricket,[1,1])`	`R1: (pay,[2,1]) (upi,[2,2])`
Output	`part-r-00000: cricket 2`	`part-r-00001: pay 3, upi 4`

1. (a) Map emits (word, 1) for every word: 5 pairs from split 1 and 4 from split 2. Each combiner sums its own pairs. “cricket” starts with c (a–m), so it goes to R0; “pay” and “upi” go to R1. The reducers add the lists: cricket 1 + 1 = 2, pay 2 + 1 = 3, upi 2 + 2 = 4.
2. Check: the file has 3 + 2 + 2 + 2 = 9 words, and 2 + 3 + 4 = 9. ✓
3. (b) Without the combiner all **9 pairs** are shuffled; with it, 3 + 3 = **6 pairs**, a third less. On real text with frequent words the saving is far larger.
4. (c) 10 GB = 10,240 MB; 10,240 ÷ 128 = **80 map tasks**. With 40 running at a time they run in 80 ÷ 40 = **2 waves**.

Expected output of the Python pipe on the same file (sorted by key): `cricket 2`, `pay 3`, `upi 4`.

EXAM TRAPS

- The number of map tasks is set by the number of input splits, not by the programmer. The number of reducers is set by the programmer (default 1).
- A combiner is an optimisation, not a phase you can rely on: it may run zero or many times. Never use it to compute an average directly.
- Reducer input is sorted by key; the values in each list are in no particular order.
- Map output goes to local disk; only reduce output goes to HDFS. A job with zero reducers (map-only, such as filtering) has no shuffle.
- Favourite question: “Explain MapReduce with the word count example and a neat diagram.” Draw Figure 6.3.1 with your own words.

CHECK YOURSELF

– REMEMBER **Write the type signatures of map and reduce.** 2 marks

map: (k1, v1) → list(k2, v2). reduce: (k2, list(v2)) → list(k3, v3).

– UNDERSTAND **What does the partitioner do, and what goes wrong if it sends one key to two reducers?** 3 marks

It decides which reducer receives each intermediate key, by default hash(key) mod R. If one key reached two reducers, each would see only part of its values and output two partial answers for the same key.

– UNDERSTAND **Why is the shuffle the costliest phase of a job?** 2 marks

Every mapper sends data to every reducer over the network, and the data must be written, transferred, merged and sorted. Map and reduce run locally on their own data.

– APPLY **Temperatures per city are (Delhi, 31), (Pune, 28), (Delhi, 35), (Pune, 30), (Delhi, 33). Give the reducer input and output for maximum temperature per city.** 3 marks

Reducer input: (Delhi, [31, 35, 33]), (Pune, [28, 30]). Output: **Delhi 35, Pune 30**. Max is associative, so the reducer can also be the combiner.

– APPLY **Mapper 1 sees marks 60 and 80 for a subject; mapper 2 sees 90. Show how a combiner computes the correct average.** 3 marks

Combiners emit (sum, count): mapper 1 gives (140, 2), mapper 2 gives (90, 1). The reducer adds them to (230, 3) and divides: 230 ÷ 3 = **76.67**. Averaging the averages would give (70 + 90) ÷ 2 = 80, which is wrong.

– ANALYZE **One key holds 60% of all pairs. What happens to the job, and what can you do?** 4 marks

That key's reducer gets 60% of the work and finishes long after the others (data skew), so the job is as slow as that one reducer. Use a combiner to shrink the data first, or split the hot key into sub-keys (salting) and merge the partial results in a second step.

BEYOND THE SYLLABUS

This is not for the exam. Few new jobs are written in raw MapReduce today: Hive runs on Tez or Spark, and Spark runs most batch work. The map–shuffle–reduce idea lives on in every distributed engine, from Spark's stages to Apache Beam and Google Cloud Dataflow. Search for "MapReduce design patterns" and the Dean–Ghemawat paper for joins, inverted indexes and secondary sort.

6.4 Hadoop ecosystem tools: YARN, Hive and Pig

HDFS and MapReduce alone are powerful but hard to use and hard to share. **YARN** (Yet Another Resource Negotiator) is the cluster's operating system: it hands out memory and CPU so that MapReduce, Spark and Hive jobs from many users run side by side. **Hive** lets analysts query files in HDFS with HiveQL, a dialect of SQL. **Pig** lets programmers write a data flow, step by step, in Pig Latin. Both turn a few lines into jobs that YARN runs. Around them sit the rest of the **Hadoop ecosystem**: Sqoop moves tables between relational databases and HDFS, Flume collects log streams, Oozie schedules chains of jobs, ZooKeeper coordinates distributed services, and HBase (Chapter 6.5) stores records.

You need: Chapter 6.2 (NameNode, DataNodes) and Chapter 6.3 (map and reduce tasks); SQL GROUP BY, which works like the Pandas grouping of an earlier lesson (Data wrangling and hierarchical indexing).

YARN: ResourceManager, NodeManagers and containers

- **ResourceManager** (RM), one per cluster, has two parts. The **Scheduler** divides the cluster's resources among applications according to a policy; it does not watch tasks. The **ApplicationsManager** accepts submitted jobs and starts (and if needed restarts) each job's ApplicationMaster.

- **NodeManager** (NM), one per worker node, launches containers, watches their memory and CPU use and reports to the RM by heartbeat.
- **ApplicationMaster** (AM), one per application, asks the RM for containers, tells NodeManagers to start tasks in them and tracks progress and failures. For a MapReduce job it is the MRAppMaster.
- **Container**: a slice of one node's resources, such as 4 GB of memory and 1 virtual core, in which one task runs.

1. The client submits the application to the ResourceManager.
2. The ApplicationsManager picks a node and has its NodeManager launch the ApplicationMaster in a container.
3. The AM registers with the RM and requests containers, naming the nodes that hold its input blocks.
4. The Scheduler grants containers on particular nodes, and the AM asks those NodeManagers to start the tasks.
5. Tasks report to the AM; when all finish, the AM unregisters and its containers are freed.

The Scheduler follows one of three policies: **FIFO** (first come, first served), **Capacity** (queues with guaranteed shares, say 70% for production and 30% for research, which may borrow idle capacity) or **Fair** (resources shared equally among running jobs over time).

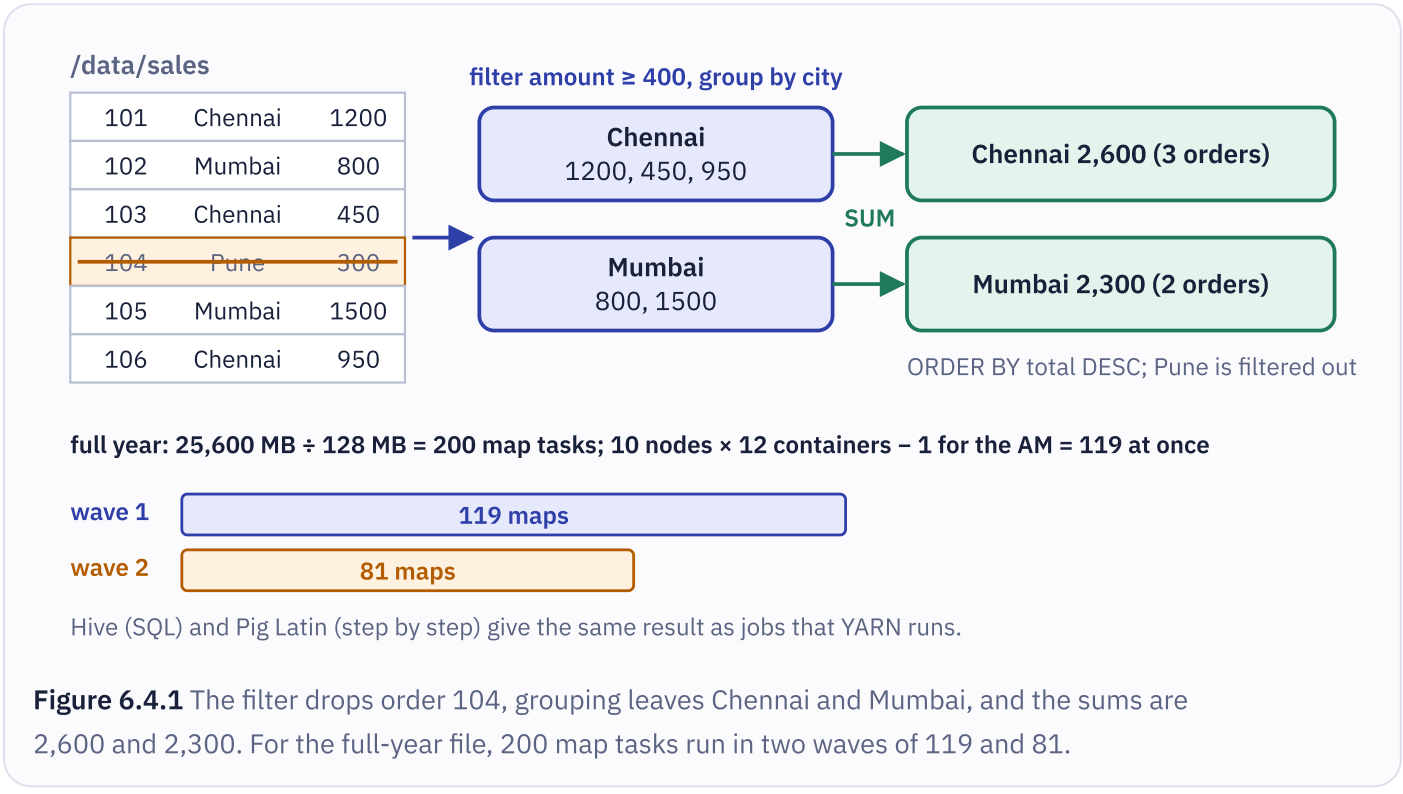


Figure 6.4.1 The filter drops order 104, grouping leaves Chennai and Mumbai, and the sums are 2,600 and 2,300. For the full-year file, 200 map tasks run in two waves of 119 and 81.

Hive: SQL on files in HDFS

Hive was built at Facebook so that analysts who knew SQL could query Hadoop data without writing Java. A Hive **table** is a directory of files in HDFS; its schema (column names and types, file format, location) is kept in the **metastore**, a small relational database. Hive uses **schema-on-read**: files are copied in as they are and checked against the schema only when queried. A query passes from the client (Beeline or a JDBC/ODBC connection to HiveServer2) to the **driver**, the **compiler** (which parses it and checks names against the metastore), the **optimizer**, and finally the **execution engine**, which runs it as MapReduce, Tez or Spark jobs on YARN.

Two table kinds matter. Dropping a **managed** (internal) table deletes its data; dropping an **external** table deletes only the metastore entry. Large tables are **partitioned** (one subdirectory per value, such as `dt=2026-10-01`, so a query for one day reads one directory) and may be **bucketed** (rows hashed into a fixed number of files). Hive is for batch analysis over large tables, with answers in seconds to minutes; it is not for single-row transactions.

Pig: data flows in Pig Latin

Pig was built at Yahoo for programmers who think in steps rather than in one SQL statement. Each Pig Latin statement takes a **relation** (a bag of tuples) and produces a new one: LOAD, FILTER, FOREACH ... GENERATE, GROUP, JOIN, ORDER, DISTINCT, then STORE (write to HDFS) or DUMP (print). Pig is **lazy**: it only builds a plan until a STORE or DUMP, then compiles the whole flow into jobs. Its data model is atom, tuple, bag and map, and it handles semi-structured data without a declared schema. You run it in the Grunt shell or as a script, in local mode or on the cluster.

	MAPREDUCE (JAVA)	PIG	HIVE
Style	Program map and reduce by hand	Procedural data flow	Declarative SQL
Schema	None	Optional	Required, in the metastore
Typical user	Java developer	Programmer building ETL pipelines	Analyst writing reports
Word count	Tens of lines	About 5 lines	About 3 lines

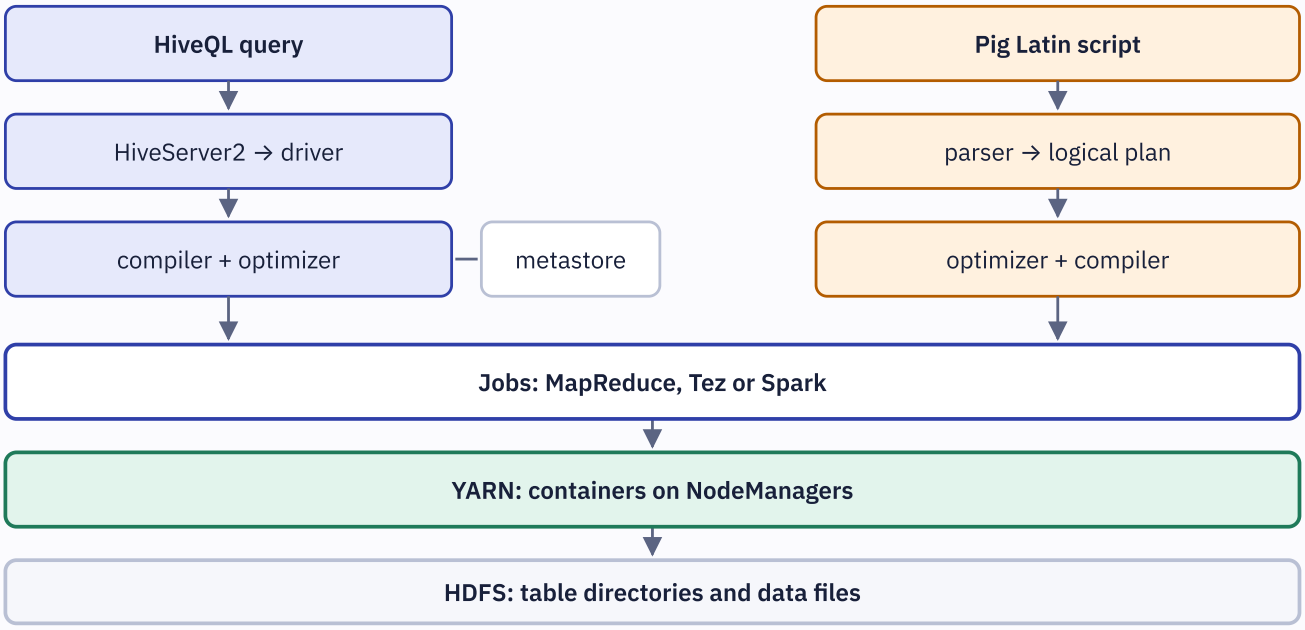


Figure 6.4.2 Hive (blue, declarative) and Pig (orange, procedural) are front ends. Both compile what you write into ordinary jobs; YARN runs them; HDFS holds the data. Only Hive needs the metastore, because only Hive insists on a schema.

WORKED EXAMPLE

Question. The HDFS directory `/data/sales` holds comma-separated rows (order_id, city, amount in ₹): 101,Chennai,1200; 102,Mumbai,800; 103,Chennai,450; 104,Pune,300; 105,Mumbai,1500; 106,Chennai,950. (a) Write HiveQL and Pig Latin for the total and number of orders per city, counting only orders of ₹400 or more, highest total first, and give the output. (b) The full year's file is 25.6 GB with 128 MB splits. Each of 10 worker nodes gives YARN 48 GB and 12 virtual cores, and a map container needs 4 GB and 1 core. How many map tasks run, and in how many waves? (10 marks)

HiveQL:

```
CREATE EXTERNAL TABLE sales (order_id INT, city STRING, amount INT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/data/sales';

SELECT city, SUM(amount) AS total, COUNT(*) AS orders
FROM sales WHERE amount >= 400
GROUP BY city ORDER BY total DESC;
```

Pig Latin:

```
sales  = LOAD '/data/sales' USING PigStorage(',')
         AS (order_id:int, city:chararray, amount:int);
big    = FILTER sales BY amount >= 400;
bycity = GROUP big BY city;
totals = FOREACH bycity GENERATE group AS city,
         SUM(big.amount) AS total, COUNT(big) AS orders;
sorted = ORDER totals BY total DESC;
DUMP sorted;
```

1. FILTER drops order 104 (₹300 < ₹400), leaving 5 rows; Pune disappears.
2. GROUP: Chennai {1200, 450, 950}, Mumbai {800, 1500}.
3. Totals: Chennai 1200 + 450 + 950 = 2,600 from 3 orders; Mumbai 800 + 1500 = 2,300 from 2 orders. Sort by total, descending.
4. Output. Hive prints `Chennai 2600 3` and `Mumbai 2300 2`; Pig prints `(Chennai,2600,3)` and `(Mumbai,2300,2)`.
5. (b) Map tasks = 25,600 MB ÷ 128 MB = **200**. Containers per node = min(48 ÷ 4, 12 ÷ 1) = 12, so 10 × 12 = 120 in the cluster, minus 1 for the ApplicationMaster = 119 at once. Waves: 119 in the first, 200 − 119 = 81 in the second: **2 waves**.

EXAM TRAPS

- YARN is not a processing engine; it only allots resources. MapReduce and Spark are engines that run on it.
- Scheduler and ApplicationsManager are both inside the ResourceManager; the ApplicationMaster is per job and runs on a worker node.
- Dropping an external Hive table keeps the data; dropping a managed table deletes it.
- Pig Latin is procedural and lazy; HiveQL is declarative. Neither is a database: both read files and run batch jobs.
- Favourite question: “Explain YARN architecture with a neat diagram” and “Compare Hive and Pig”.

CHECK YOURSELF

– REMEMBER **Name the two components of the YARN ResourceManager.** 1 mark

The Scheduler and the ApplicationsManager.

– REMEMBER **Name four tools of the Hadoop ecosystem besides HDFS and MapReduce, with one job each.** 2 marks

Hive (SQL queries), Pig (data-flow scripts), HBase (record store), Sqoop (database ↔ HDFS transfer), Flume (log collection), Oozie (workflow scheduling), ZooKeeper (coordination). Any four.

– UNDERSTAND **Why did Hadoop 2 introduce YARN?** 3 marks

Hadoop 1’s JobTracker did both resource management and job tracking for every job, which limited cluster size and allowed only MapReduce. YARN moves resource management to the RM and job tracking to one AM per application, so the cluster scales further and runs other engines such as Spark.

– UNDERSTAND **What is the Hive metastore and what does schema-on-read mean?** 3 marks

The metastore is a relational database holding each table’s columns, types, format, location and partitions. Schema-on-read: data is loaded as raw files and checked against the schema only when a query reads it.

– APPLY **Write a HiveQL query for the average marks per department from students(roll, dept, marks), for departments averaging above 70.** 3 marks

```
SELECT dept, AVG(marks) AS avg_marks FROM students GROUP BY dept HAVING AVG(marks) > 70;
```

– APPLY **Nodes give YARN 64 GB and 16 cores; containers need 6 GB and 1 core. How many containers fit on one node?** 2 marks

Memory allows 64 ÷ 6 = 10 (rounded down); cores allow 16. The smaller wins: **10 containers**, memory-bound.

– ANALYZE **A team cleans messy JSON logs in many steps, and another writes monthly sales reports. Which tool suits each?** 4 marks

Cleaning: Pig (or Spark), because the job is a multi-step flow over semi-structured data without a fixed schema. Reports: Hive, because the data is tabular, the users know SQL, and partitioned tables make monthly queries cheap.

BEYOND THE SYLLABUS

This is not for the exam. Pig is rarely chosen for new projects now; Spark SQL and DataFrames do the same work faster. Hive's metastore lives on as the catalogue that Spark, Trino and Presto read, and Kubernetes now competes with YARN as the cluster manager. Search for "Trino" and "Hive metastore with Spark" to see how SQL on data lakes works today.

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
map and reduce	(k1, v1) → list(k2, v2); (k2, list(v2)) → list(k3, v3)	6.3
Map tasks and waves	Tasks = input ÷ split size; waves = ⌈tasks ÷ tasks that fit at once⌉	6.3
Partitioner	Reducer = hash(key) mod R	6.3
Combiner	Only for commutative, associative operations; average via (sum, count)	6.3
Containers per node	min(node memory ÷ container memory, node cores ÷ container cores)	6.4
Hive tables	Managed: DROP deletes data. External: DROP keeps data	6.4

Exam question bank

Long-answer exam questions on these chapters, each with an outline of the points an examiner looks for. Answer one in full on paper before you read its outline.

Remember and understand

– UNDERSTAND **Explain YARN architecture and the steps of running an application on it. Why did Hadoop 2 introduce YARN?** 13 marks

ResourceManager (Scheduler, ApplicationsManager), NodeManager, ApplicationMaster, container; the five steps; FIFO, Capacity and Fair schedulers; Hadoop 1 JobTracker bottleneck and MapReduce-only limit.

Apply

– APPLY **Trace word count in MapReduce on “big data big ideas”, “data moves to code” split across two mappers, with a combiner and two reducers (a–m, n–z). How many map tasks does a 20 GB input need with 128 MB splits?** 13 marks

Map output per split; combiner output (big 2, data 1, ideas 1 | data 1, moves 1, to 1, code 1); R0 (a–m) gets big, code, data, ideas, moves; R1 (n–z) gets to; final: big 2, code 1, data 2, ideas 1, moves 1, to 1 (8 words ✓). 20 GB = 20,480 MB ÷ 128 = **160 map tasks**. Diagram in the style of Figure 6.3.1.

– APPLY **Table marks(roll, dept, subject, score) is in HDFS. Write HiveQL and Pig Latin for the average score per department, only for scores of 40 or more, highest average first.** 10 marks

Hive: CREATE EXTERNAL TABLE with delimiter and LOCATION; `SELECT dept, AVG(score) AS a FROM marks WHERE score >= 40 GROUP BY dept ORDER BY a DESC;` Pig: LOAD with PigStorage and schema; FILTER BY score >= 40; GROUP BY dept; FOREACH … GENERATE group, AVG(…); ORDER … DESC; DUMP. Mention that both compile to jobs on YARN.

Glossary (terms used in this lesson)

ApplicationMaster Per-job YARN process that requests containers and tracks the job’s tasks.

BASE	Basically available, soft state, eventually consistent: the relaxed guarantee of many NoSQL stores.
Combiner	A local reducer run on one mapper’s output to cut network traffic.
Container	A slice of a node’s memory and CPU in which one YARN task runs.
DataNode	HDFS worker that stores blocks and serves reads and writes.
Data lake	Low-cost store of raw data of every type, structured only when read.
Metastore	Hive’s database of table schemas, locations and partitions.
NameNode	HDFS master that keeps the namespace and block locations in memory.
Partitioner	The function that sends each intermediate key to a reducer, by default hash(key) mod R.
Schema-on-read	Applying structure to stored data only when it is queried.
Shuffle	Moving intermediate data across the network so all values of a key meet in one place.
YARN	Yet Another Resource Negotiator: Hadoop’s resource manager and job scheduler.

Sources and further reading

Textbooks

- Tom White, *Hadoop: The Definitive Guide*, 4th edition, O’Reilly, 2015: HDFS, MapReduce, YARN, Hive, Pig and HBase.
- Seema Acharya and Subhashini Chellappan, *Big Data and Analytics*, Wiley India: Hadoop, NoSQL, MongoDB, Cassandra, Hive and Pig, widely prescribed in Indian universities.
- Davy Cielen, Arno D. B. Meysman and Mohamed Ali, *Introducing Data Science*, Manning, 2016: handling large data, Hadoop and Spark, NoSQL.
- Jure Leskovec, Anand Rajaraman and Jeffrey D. Ullman, *Mining of Massive Datasets*: the chapter on MapReduce and distributed file systems.
- Bill Chambers and Matei Zaharia, *Spark: The Definitive Guide*, O’Reilly, 2018.
- Pramod J. Sadalage and Martin Fowler, *NoSQL Distilled*, Addison-Wesley, 2012.
- Martin Kleppmann, *Designing Data-Intensive Applications*, O’Reilly, 2017: replication, partitioning, consistency and stream processing.
- Ralph Kimball and Margy Ross, *The Data Warehouse Toolkit*, 3rd edition, Wiley: dimensional modelling.

- Jiawei Han, Micheline Kamber and Jian Pei, *Data Mining: Concepts and Techniques*, 3rd edition, Morgan Kaufmann: the chapter on data warehousing and OLAP.

Open web resources

> The open web could not be reached while this book was written, so these links were not opened for it. They are well-known free resources listed from memory; check that each still opens. The DPDP Act penalty ceiling (₹250 crore for failing to take reasonable security safeguards) was confirmed with a web search that returned summaries only.

- Apache Hadoop, "HDFS Architecture" guide: hadoop.apache.org, HdfsDesign.
- Apache Spark documentation, including the RDD programming guide and the Structured Streaming guide: spark.apache.org/docs/latest.
- Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", OSDI 2004: research.google.com archive, mapreduce-osdi04.pdf.
- Leskovec, Rajaraman and Ullman, *Mining of Massive Datasets*, free online edition and lecture slides: mmds.org.
- Apache Cassandra documentation, on data distribution, replication and consistency levels: cassandra.apache.org/doc/latest.
- Ministry of Electronics and Information Technology, Government of India, for the text of the Digital Personal Data Protection Act, 2023: meity.gov.in.

Syllabus: the University free n easy universal syllabus, built from top universities' syllabi.

Built for the University free n easy YouTube channel from the universal syllabus of Data Science and Analytics (Oct 2026). Written with top universities' syllabi and standard textbooks; check a formula against your prescribed book before an exam, and report mistakes on the channel.