Big Data — Full Revision (All Important Topics)

1) The 5 Vs of Big Data (what they really mean in practice)

- **Volume:** Massive datasets (GB → PB) overwhelm a single machine or RDBMS table. You need distributed storage (e.g., HDFS) and scale-out processing (e.g., MapReduce/Spark).
- **Velocity:** Ingest/compute quickly (streams, logs, sensors). Batch = hours/minutes; real-time = milliseconds/seconds.
- **Variety:** Tables (structured), logs/JSON/XML (semi-structured), images/video/text (unstructured). Hadoop's **schema-on-read** fits this (load first, interpret later) .
- **Veracity:** Dirty/incomplete data; handle with validation, filtering, replication, and outlier handling.
- **Value:** Insight/impact (KPIs, cost/time saved). All tools/architectures exist to realize this V.

Why Hadoop fits Vs: Can process un/semi-structured data (schema-on-read), scales linearly, and tolerates failures automatically, so you can run "whole-dataset" jobs in reasonable time .

---

2) BDA Cloud Architectures (how clouds fit analytics)

- **Compute on demand (IaaS)** for clusters, **managed platforms (PaaS)** for Hadoop/Spark, **analytics services (SaaS)** for SQL/lakehouse. Elastic scaling + pay-as-you-go keeps costs sane for spiky jobs.
- Typical layered view: **Storage (object/HDFS) → Processing (MR/Spark/YARN) → Query/ML services → Dashboards**.
- Why it matters: eliminates buying/maintaining hardware; lets you burst big jobs, isolate environments, and pick the right engine per workload (batch, stream, SQL, ML).

(Your slides 1.5/1.6 carry the diagrams; above is the distilled "why/what/how" you need to write clearly.)

---

3) Linux File System & Default Directories (core paths you're expected to know)

- **/** root of everything
- **/bin** essential user binaries; **/sbin** admin binaries
- **/etc** system/app configs; **/var** logs, cache, spool; **/tmp** temporary files
- **/lib, /lib64** shared libraries; **/usr** user apps/tools; **/home** user folders; **/dev** device files
  Why in BDA? You'll script in Bash, tail logs in `/var/log`, mount volumes under `/mnt`/`/data`, and point Hadoop/Docker paths correctly.

---

4) All Virtualization Types (very important)

What is it? Abstract a physical host into logical machines/resources.

- **Server virtualization (hypervisors):** Type-1 (bare-metal: ESXi, Hyper-V, Xen) vs Type-2 (hosted: VirtualBox). Snapshots, migration, HA/failover: core exam talking points.
- **Desktop virtualization (VDI):** Centralize user desktops; remote access.
- **Application virtualization:** Isolate app + deps from OS.
- **Storage virtualization:** Pool disks (SAN/NAS abstractions).
- **Network virtualization / SDN:** Logical networks over physical.
- **Data virtualization:** Unified logical data view across sources.

Containers (OS-level virtualization) vs VMs: Containers share the host kernel via **namespaces** (PID/NET/MNT/UTS/IPC/USER) and **cgroups** for CPU/RAM limits; they're lighter/faster than VMs and are now the default packaging for analytics services .

---

5) Disk & Filesystem Block Sizes (and why HDFS uses huge blocks)

- **Disk block (physical):** 512 B–4 KB.
- **Filesystem block (logical):** typically 4–8 KB.
- **HDFS block: 128 MB default** (sometimes 256 MB). Why huge? To **minimize seeks** and maximize sequential throughput for large scans; seek ~10 ms vs transfer ~100 MB/s means blocks ~100 MB make seek ~1% of total time .
- Files split into block-sized chunks, stored independently; a 1 MB file uses ~1 MB (doesn't "waste" to full block on disk) .
- Big blocks reduce NameNode metadata pressure and favor streaming; smaller blocks increase parallelism but add overhead.

---

6) Intel Optane vs SSD vs HDD (exam-style contrasts)

HDD: Magnetic platters; cheap, huge, slow (100–200 MB/s), high latency. **Best** for cold/archival layers (data lakes) .
SSD: NAND flash; fast sequential and random I/O (SATA ~500–600 MB/s; NVMe 3–7 GB/s), ~0.1 ms latency; ideal for hot datasets, ETL, real-time queries .
Optane (persistent memory/NVRAM): bridges RAM and SSD; ~1–3 GB/s, ~300 ns latency; superb for caches, checkpoints, hybrid memory tiers, write-heavy workloads .

Why this matters: Hybrid tiering = HDD (capacity/cold) + SSD (hot/compute nodes) + Optane (memory-adjacent cache). Your answer should tie device traits to BDA job patterns (scan-heavy, random small reads, checkpointing).

---

7) Data Locality (everything from the last two lectures)

- **Principle:** Move compute to data, not data to compute — network is the bottleneck; local disk I/O is faster .

- **Levels:** node-local → rack-local → off-rack (as last resort). Schedulers try node-local first to cut network traffic.
- **Why Hadoop wins for big scans:** Batch engines (MR/YARN) run **whole-dataset** queries by spreading tasks and keeping I/O mostly local, with built-in re-execution if a task/node fails .
- **Small files are bad:** NameNode holds block metadata in RAM (~150 B/block). Millions of tiny files exhaust memory and kill throughput .

---

8) HDFS Fundamentals (blocks, replication, roles)

- **HDFS fits huge files** with streaming access on commodity hardware; it's optimized for **high throughput**, not millisecond latency, and is **write-once, read-many** (append patterns exist but not random updates) .
- **Blocks:** 128 MB default; benefits: files can outgrow any single disk; simpler storage subsystem; easy replication/fault tolerance .
- **Replication (default 3):** typically 1 local, 1 same rack, 1 different rack — for both **performance and rack-level fault isolation**.
- **NameNode vs DataNode:**
  - **NameNode:** namespace + block mapping + replication control; persists FsImage + EditLog (checkpointing) and coordinates clients (but does not serve data) .
  - **DataNode:** stores/serves blocks; heartbeats + block reports; replication/deletion as instructed; participates in write pipelines .

---

9) YARN (how Hadoop 2+ schedules and scales)

- **ResourceManager:** global scheduler; allocates containers (CPU/RAM).
- **NodeManager:** runs on each node; launches/monitors containers; posts heartbeats.

- **ApplicationMaster:** per-job brain; requests containers, launches tasks, tracks progress/failures, reports status.
  This decouples resource management from compute models, allowing MR, Spark, Tez, etc., to share the cluster efficiently .

---

10) MapReduce (concept + pseudocode you can write fast)

- **Why MR:** Scales linearly, hides failures (failed tasks auto-rescheduled), and suits batch analyses of entire datasets .
- **Flow:** Input splits (~block size) → **Map** (emit key,value) → **Shuffle/Sort** (group by key) → **Reduce** (aggregate) → output to HDFS. The **Job** JAR and input/output paths, mappers/reducers, and output key/value classes are set in the driver; **waitForCompletion(true)** runs with verbose logs .
- **Combiner:** optional mini-reduce at mapper side (must be associative/commutative; e.g., sum/max).
- **Classic exam pseudocode (Max Temp):**
  Map: parse (year, temp), filter bad/suspect, emit (year, temp).
  Reduce: output (year, max(temp)) .

---

11) Bash Scripting (with comments)

- **Shebang** `#!/bin/bash`, **chmod +x** to execute.
- **Positional params** `$0, $1, $2` …; quote your vars.
- **Control:** `if … then … fi`, `for/while` loops, functions (`name(){ … }`).
- **Safety:** `set -e` (exit on error), check `$?`, use `IFS= read -r` to avoid whitespace trimming.
- **Redirection:** `>`, `>>`, `<`, and pipes.

- **Typical assessed scripts:** log analyzer (top erroring services → CSV), timestamped backups, simple monitors, small ETL helpers — *write with comments explaining each step*.

---

12) Docker — meanings of the commands (don't write them, explain them)

- **Architecture (how it stacks):** CLI → **dockerd** → **containerd** (lifecycle) → **runc** (sets up namespaces/cgroups) → Linux kernel; this is why containers feel "instant" and portable .
- **Images/layers:** Dockerfile instructions create cached, shareable layers; rebuilds reuse unchanged layers for speed and space efficiency .
- **Run/exec/stop/rm:** start a container from an image; execute in a running container; stop/remove containers. `-d` = detached; `--name` = readable name; `-p host:container` = port mapping; `--net` = network. You should be able to explain "what happens" when we `docker run` a service container (e.g., NameNode) and why those flags are needed.
- **Volumes (important):** externalized persistence outside the writable container layer; survive restarts/removals; shareable across containers; managed by Docker; essential for DBs/logs/stateful apps .
- **Storage types to mention:** volumes, bind mounts, and plugins (EBS, etc.) — know the **purpose**: persistent, shareable, backup-friendly .

---

13) RAID (mirroring) vs HDFS replication (what to write in a compare/contrast)

- **RAID:** within one server; disk-level redundancy; quick failover to mirror, but **not** rack/server failure safe. Good for NameNode metadata disks.
- **HDFS replication:** cluster-level; **HDFS blocks replicated across nodes/racks**; automatic re-replication and parallel reads; protects against node/rack failures — exactly

what big data durability needs .

---

14) Storage Blues → Why we needed Hadoop

- Capacity exploded but **seek/access speeds lag**; a single drive reading TBs takes hours. Parallelizing across **many disks/nodes** reduces total time **drastically**, but brings failure risk (hence replication) and the need to **combine** distributed data safely — enter **HDFS + MapReduce** .

---

15) HPC (MPI) vs Hadoop (data-intensive analytics)

- **HPC/MPI:** compute-intensive tasks, shared SAN, explicit message passing (send/recv), fragile on failure.

- **Hadoop:** data-intensive; **co-locates data and compute**, hides failures, simple key-value programming model. Use this contrast to justify "why Hadoop for logs/clicks" vs "why MPI for simulations" .

---

16) Put it together: how a Hadoop job actually runs (you can narrate this)

1. Client submits job JAR + I/O paths.

2. YARN starts **ApplicationMaster**.

3. AM asks RM for containers near the data (locality).

4. **Map** tasks process splits; bad records filtered early; intermediate pairs staged.

5. **Shuffle/Sort** groups by key; **Reduce** aggregates; output written to HDFS; existing output dir must not exist (safety) .

(That "output dir must not exist" line is a classic sanity point to mention.)

---

17) Short, high-yield numerics you can drop in answers

- **Default HDFS block:** 128 MB; **default replication:** 3 copies (often across racks).
- **NameNode metadata per block: ~150 bytes** (why small files are poison) .
- **Optane vs SSD vs HDD latency:** ~**300 ns** vs **~100 µs** vs **~10 ms** (orders-of-magnitude story).
- **NVMe speeds: 3–7 GB/s**; **SATA SSD: ~500–600 MB/s**; **HDD: ~100–200 MB/s** .

---

18) What to write if asked to "explain" a Docker run line

Example:
`docker run -d --name namenode --net hadoop-net -p 9870:9870 hadoop-nn`

- **Run a container** in background (**-d**), give it a **name**, attach it to a **user network** so DataNodes/others can reach it, **publish the web UI port** 9870, and **use the namenode image** so it boots the NN process with its config.
  (Always explain effect of each flag and why it's needed in a cluster.)

---

19) Bash: write with comments and safety

If a scripting question appears, **comment every step**, use `set -e`, handle missing inputs, print clear status, and echo where outputs are saved. If reading lines use:

```
while IFS= read -r line; do
  # process "$line"
done < file.txt
```

Explain **why** you used `IFS= read -r` (preserve spaces/backslashes) and **why** you avoided `cat ... | while` (subshell scope).

Quick exam advice (based on sir's cues)

- He likes **conceptual synthesis** + **practical tie-ins**. When you define, **pair it** with why it matters (performance/fault-tolerance/operability).
- Sprinkle **block/latency numbers** where relevant.
- For Docker and Bash, **explain intent** (what the command/flag is achieving), not rote syntax.
- For MR/HDFS, **always mention locality + failure recovery** — that's the heart of Hadoop.