Big Data (DS-GA 1004) – Ultimate Finals Preparation Notes

Fully based on Week 1 Slides + Lecture Transcripts

Spring 2025

1 Introduction to Big Data

Big Data is about scaling, not just trendy tools. This course addresses scaling in data storage, computation, and communication across distributed systems.

1.1 Warm-Up Metaphor

Imagine an ant trying to carry a giant **Colocasia gigantea** (Elephant Ear) leaf: too large for one. Solution? **Team effort.** This parallels Big Data: complex tasks distributed among many units.

2 What is Big Data?

- **Scaling** challenges in storage, computation, and processing.
- **Operational Definition**: Data that does not fit comfortably on a laptop.
- **Deeper Definition**: Big Data often requires **coordinated processing across multiple computers**.

2.1 Importance of Scale

- **J.B.S. Haldane analogy**: Small mass → minor effects; Large mass → catastrophic consequences.
- More data (rows/columns) improves:
  - Statistical power
  - Parameter estimation
  - Personalized recommendations
  - Subgroup analysis

  - Leveraging high dimensionality

- But: **too much data unhandled** becomes a showstopper.

3 CS vs DS View of Data

- **DS View**: Data are immutable givens to derive insights.
- **CS View**: Data are bits/bytes needing efficient movement, storage, transformation.

4 The Five V's of Big Data (Laney, 2001; Hurwitz et al., 2013)

- **Volume**: Sheer amount of data.
- **Velocity**: Speed of incoming data.
- **Variety**: Structured vs. unstructured formats.
- **Veracity**: Uncertainty, noise, errors.
- **Value**: Potential actionable insights.

5 Why Study Big Data?

- Machine Learning and statistics perform better with more data.
- However, scaling creates new issues.
- The course covers **underlying principles** to adapt to evolving tools.

6 Class Context

- **DS-GA 1001**: Small datasets, core concepts.
- **DS-GA 1004**: Large datasets, practical scaling.
- Evolution from ideas to massive implementation.

7 Expected Outcomes

- Familiarity with distributed computing and storage.
- Understanding technical scaling challenges.
- Judging the right tool for the right scale.
- Tools: Git, SQL, Hadoop, MapReduce, HDFS, Spark, Dremel, Parquet, Dask, CUDA, etc.

8 How the Class Works

- Platform: Brightspace.
- Homework via GitHub Classroom.
- Weekly assigned readings.
- Grading: 25% HW, 25% Capstone, 25% Final (T/F), 9% Quizzes, 16% Low-stakes work.

9 Key Concepts in Resource Management

9.1 Storage

- Storage costs have dropped exponentially (e.g., 540TB costs $11,292 today).
- Storage rarely the main bottleneck anymore.

9.2 Communication

- **Latency critical**:
  - L1 Cache: 1 ns
  - L2 Cache: 4 ns
  - RAM: 100 ns
  - SSD: 16,000 ns
  - HDD: 2,000,000 ns
- **Main memory access often bottleneck**. Minimize data movement.

9.3 Computation

- **Moore's Law slowing down**. CPU clock speeds plateau.
- **Parallelism** (multi-core, GPUs) is key to future computation.

10 Communication is the Hidden Enemy

- Communication cost grows **super-linearly**.
- **Brooks' Law**: Adding manpower to a late project delays it further.
- Coordination becomes a primary cost in scaling complex systems.

11 Principles of Scaling

11.1 Tasks Easy to Parallelize

- Example: Moving stones for Pyramids (nearly linear scaling).

11.2 Tasks Hard to Parallelize

- Example: Building Cathedrals, software.
- **Communication dominates**.
  "Data Cathedral" metaphor: Big Data requires disciplined coordination.

11.3 Lecture Emphasis

- **Carrying the Leaf**: Distributed effort needed for massive tasks.
- **Brutality of Parallelization**: Expect high effort from students and instructors.
- **Pyramids vs. Operating Systems**: Simple vs. complex parallelization.

12 Big Data Strategy

- Distribute storage and processing.
- Minimize communication overhead.
- Introduce hierarchies where necessary.

13 Critical Quotes

- *"We shouldn't be trying for bigger computers, but for more systems of computers."* – Rear Admiral Grace Hopper
- *"Adding manpower to a late project makes it later."* – Brooks' Law

14 Next Steps

- **Read**: Garcia-Molina, Ullman, & Widom, 2009, Chapter 2.
- Topics: Centralized Systems, File Systems, Relational Databases.

Big Data (DS-GA 1004) – Lecture 2 Finals Preparation Notes

Fully based on Week 2 Slides + Lecture Transcript + Expanded Knowledge

Spring 2025

1 Centralized Systems: File Systems and Relational Databases

2 Announcements

- HW1 due next Monday (02/10).
- Lab 2 focuses on SQL.
- Updated syllabus available on Brightspace.

3 Confusion, Doubt, and Struggle (CDS) Questions

3.1 Is a large random array "data"?

Yes, if it is used for input to models, simulations, or analysis, even if it is synthetic.

3.2 What is a Transistor?

- Basic electronic switch: allows or blocks electrical current.
- Fundamental building block of CPUs.
- Miniaturization led to Moore's Law and modern computing.

3.3 What is a CPU?

- Central Processing Unit: Executes programs by processing data and instructions.
- Contains Arithmetic Logic Unit (ALU), Control Unit, and Registers.
- Modern CPUs have multiple cores (multi-core architecture).

3.4 How does a Computer Work?

- CPU interacts with memory (RAM) and storage (HDD/SSD).
- Data flows at speeds constrained by physical wiring and architecture.
- Minimizing distances (e.g., cache memory near CPU) improves speed.

4 File Systems

4.1 Overview

- Organizes data into hierarchical structures (directories and files).
- Persistent storage across sessions.

4.2 Advantages

- Simplicity.
- Portability.
- Low overhead.

4.3 Limitations

- Poor support for complex queries.
- Limited metadata.
- Difficult to enforce data consistency.

5 Relational Databases

5.1 Why Use Databases?

- Structured querying (SQL).
- Data integrity and schema enforcement.
- Concurrency management.
- Scalability for larger datasets.

5.2 Relational Model Concepts

- Data is stored in **tables** (relations).
- Each table consists of **tuples** (rows) and **attributes** (columns).
- Tuples are unordered and unique.

5.3 Mathematical Basis

- A relation R over sets A1, A2, ..., An is a subset of the Cartesian product $A1 \times A2 \times \cdots \times An$.
- $R \subseteq A1 \times A2 \times \cdots \times An$.

6 Structured Query Language (SQL)

6.1 Overview

- Declarative language: Describe *what* you want, not *how* to get it.
- Supported by almost all modern RDBMS.

6.2 Core SQL Operations

- `SELECT`: Retrieve data.
- `INSERT`: Add new data.
- `UPDATE`: Modify existing data.
- `DELETE`: Remove data.

6.3 Joins

- **INNER JOIN**: Matching rows only.
- **LEFT OUTER JOIN**: All from left table + matched from right.
- **RIGHT OUTER JOIN**: All from right table + matched from left.
- **FULL OUTER JOIN**: All rows from both, matched where possible.
- **CROSS JOIN**: Cartesian product.

6.4 Aggregation Functions

- `AVG`, `SUM`, `COUNT`, `MAX`, `MIN`.
- `GROUP BY` and `HAVING` clauses for grouping and filtering aggregates.

7 Indexing

7.1 Purpose

- Speeds up retrieval operations.
- Organizes data in structures like B-trees or hash maps.

7.2 Trade-offs

- Increases storage space.
- Slows down `INSERT` and `UPDATE` operations.

8 ACID Properties of Transactions

8.1 Atomicity

- All or nothing execution.
- Partial updates are rolled back.

8.2 Consistency

- Enforces database rules.
- Moves from valid state to valid state.

8.3 Isolation

- Transactions operate independently.
- Prevents "dirty reads," "non-repeatable reads," "phantoms."

8.4 Durability

- Once committed, the result persists even after crashes.

9 Expanded Knowledge: Real-World Concepts

9.1 CAP Theorem (Related to Distributed Databases)

- You can have at most two out of three: **Consistency**, **Availability**, **Partition Tolerance**.

9.2 Normalization vs. Denormalization

- **Normalization**: Remove redundancy, maintain integrity.
- **Denormalization**: Add redundancy to optimize read performance.

9.3 Common SQL Pitfalls

- **N+1 Query Problem**: Caused by poor join design.
- **Over-indexing**: Can degrade performance.
- **Non-optimal GROUP BY**: Can lead to memory overuse.

10 Summary

- File systems are simple but limited.
- Relational databases provide structure, consistency, and concurrency.
- SQL empowers data querying.
- Indexes accelerate reads but slow writes.
- ACID ensures transactional reliability.
- Normalization minimizes redundancy but may complicate queries.
- CAP theorem explains distributed system trade-offs.

11 Next Week

- Topic: **Distributed Computation with Map-Reduce**.
- Readings:
  - Dean & Ghemawat (MapReduce)
  - DeWitt & Stonebraker (Parallel Databases)

Big Data (DS-GA 1004) – Lecture 3 Finals Preparation Notes

Fully based on Week 3 Slides + Lecture Transcript + Expanded Knowledge

Spring 2025

1 Distributed Computation: Introduction to MapReduce

2 Announcements

- Everyone has an HPC (High-Performance Computing) account.
- Lab 3: Hands-on with MapReduce.
- HW1 is due tomorrow (02/11).
- Quiz during this week's lab.
- HW2 releases this week (due 02/27).

3 Historical Context: How Data Management Evolved

3.1 Timeline

- **1940s**: Vacuum tubes and plugboards (hardware dominated).
- **1950s**: Punchcards (software programs emerged).
- **1960s**: File systems, custom software per application/query.
- **1970s**: **Relational model** proposed (Codd, 1970), SQL standard.
- **1980s**: Relational DBMS become commodity technology.
- **1990s**: Web explosion; Distributed/parallel computing begins.
- **2000s**: **MapReduce introduced** (Google) for massive scale.
- **2010s**: Cloud computing, cluster management, dataframes.

4 Concept Review: RDBMS

- **Relations** standardize data structure.
- **SQL** standardizes data access.
- **DBMS** abstracts internal complexities.
- **ACID Transactions** guarantee data safety.

5 Confusions Resolved

5.1 Keys in Computer Science

- Think of **key** as a unique **identifier** (ID), unless specified otherwise.

5.2 Normalization

- **1NF**: Each field contains atomic (indivisible) values.
- **Higher NFs**: Ensure no partial or transitive dependencies.
- **Heuristic**: Break into multiple tables linked by primary/foreign keys.

5.3 Indexing

- Lookup tables to accelerate search.
- Variants include clustered vs non-clustered indexes.
- **SQL Syntax**: `CREATE INDEX idx_name ON table(column);`

6 Introduction to MapReduce

6.1 Motivation

- Processing N massive documents $\Rightarrow \Omega(N)$ time sequentially.
- Embarrassingly parallel problem $\Rightarrow$ process documents independently.

6.2 Google's Context

- Text indexing for entire web.
- Need to distribute workload efficiently & aggregate results.

7 MapReduce Framework

7.1 Core Idea

- Program consists of only two stages: **Map** and **Reduce**.
- **Map Phase**: Apply a function independently to data splits to produce (key, value) pairs.
- **Shuffle Phase**: Group all identical keys together.
- **Reduce Phase**: Aggregate values per key.

7.2 Example: Word Count

- **Mapper**: Emit (word, 1) for each word in document.
- **Shuffle**: Group identical words.
- **Reducer**: Sum counts for each word.

8 Functional Programming Basis

8.1 Mapping

- `map(f, [x1, x2, ..., xn])` $\rightarrow [f(x1), f(x2), ..., f(xn)]$

8.2 Reducing

- `reduce(g, [x1, x2, ..., xn])` recursively applies function g to combine values.

9 Important Properties

9.1 Associativity

- $(a \oplus b) \oplus c = a \oplus (b \oplus c) \Rightarrow$ Order of reduction doesn't matter.

9.2 Determinism

- Same input yields the same output.
- No side effects.

9.3 Scalability

- More machines $\Rightarrow$ Linear speed-up $\approx$ possible.

10 Working with MapReduce: Practical Tips

- Prefer integers or strings for keys (floating point issues).
- Design simple `map` and `reduce` functions.
- Use combiners to reduce data transfer when possible.

10.1 Combiner Example

- Local reduction before shuffle phase.
- Word count example: sum local counts.

11 Challenges and Drawbacks

11.1 Key Skew

- Uneven distribution of keys $\Rightarrow$ Some reducers overloaded.

11.2 Not Suitable For

- Iterative algorithms (e.g., gradient descent).
- Interactive applications (e.g., real-time exploration).

12 Critiques of MapReduce (Stonebraker et al.)

- Too low-level $\rightarrow$ No schema awareness.
- Lacks indexes, views, integrity constraints.
- Not novel: Earlier systems had partitioning & aggregation.
- Not compatible with traditional DBMS ecosystem.

13 Why was MapReduce so impactful?

- Simple abstraction; easy to learn and deploy.
- Fits many one-shot batch-processing needs.
- Democratized distributed programming.

14 Expanded Knowledge: Legacy and Beyond

14.1 Hadoop

- Open-source implementation of MapReduce.
- Built atop HDFS (Hadoop Distributed File System).

14.2 Modern Successors

- Apache Spark (supports iterative algorithms, faster in-memory computation).
- Flink, Dask for advanced distributed workflows.

14.3 Design Philosophy

- Divide and Conquer ("Divide et Impera").
- Stateless computation where possible.
- Embrace locality to minimize communication.

15 Summary

- MapReduce enabled scalable parallel computation.
- Great for batch processing, bad for iterative ML.
- Concepts of mapping, shuffling, and reducing remain foundational.

16 Next Week

- Hadoop Distributed File System (HDFS): How to manage distributed storage.
- Hands-on with HDFS.

Big Data (DS-GA 1004) – Lecture 4 Finals Preparation Notes

Fully based on Week 4 Slides + Lecture Transcript + Expanded Knowledge

Spring 2025

1 Distributed Storage: Introduction to HDFS

1.1 Class Culture Reminder

- "I don't believe it unless I implement it in code and understand the algorithm": Computer Science (CS) view.
- "I don't believe it unless I see proof and derivation": Statistics (IDS) view.
- "I need to see experimental design and data": Data Science (DS) view.

2 Last Week vs. This Week

- Last week: Introduction to distributed computation and MapReduce.
- This week:
  - CDS
  - Data storage concepts
  - Distributed data storage
  - The Hadoop Distributed File System (HDFS)

3 Confusion, Doubt, & Struggle: Hash Functions

- Take input (message) → output a fixed-size string.
- Properties:
  - Deterministic
  - Fast computation
  - Small input changes → Large output changes
  - Pre-image resistant
  - Collision resistant

3.1 Examples

- Function $f(x) = x \mod 10$
  - $f(7) = 7$, $f(42) = 2$, $f(420) = 0$, $f(2777) = 7$
  - Not collision resistant (e.g., 7 and 2777 map to 7).
- Function $g(x) = ([ax] + b) \mod p$, with constants $a = 3$, $b = 5$, prime $p = 97$
  - More robust; harder to invert.

3.2 Hashing Use Cases

- Error detection (checksum).
- Password storage (cryptographic hashes).
- Hash tables (for fast lookups).

4 Data Storage Systems

4.1 File Systems and Hard Disks

- Files $\rightarrow$ broken into **blocks** $\rightarrow$ mapped onto **sectors**.
- Sectors: Smallest unit (512 to 4096 bytes).
- Moving read head limits throughput $\rightarrow$ Files can fragment.

4.2 RAID (Redundant Array of Inexpensive Disks)

- Multiple disks appear as one to OS.
- Goals:
  - Capacity
  - Reliability
  - Throughput
- Different RAID levels trade off these goals differently.

4.3 Common RAID Levels

- Striping only (no redundancy, capacity scales linearly).
- Full redundancy (RAID 1: mirrored disks).
- RAID 5: Distributed parity (balance between reliability and capacity).

4.4 RAID 5 and Parity Bits

- XOR based parity: Recover lost data blocks.
- Example: Disk0 $\oplus$ Disk1 = Parity.

5 Distributed Data Storage

5.1 Why Not Just RAID?

- RAID is for single machines.
- For **distributed computation**, we need distributed storage.
- Communication over network $\rightarrow$ Bottleneck.

5.2 Simple (Bad) MapReduce Implementation

- Central node stores all data.
- Data transfer becomes network bottleneck.

5.3 Extreme Local Storage

- Replicate all data on all nodes $\rightarrow$ Wasteful and expensive.

5.4 Distributed File Systems

- Reasonable trade-off between redundancy and communication.
- Key Design Factors:
  - Minimize communication.
  - Redundancy control.
  - Data locality.
  - Access patterns (small programs, large datasets).

6 HDFS (Hadoop Distributed File System)

6.1 Key Features

- Distributed, redundant storage.
- Optimized for write-once, read-many patterns.

6.2 Hadoop Framework

- **MapReduce**: Processing engine.
- **YARN**: Resource manager.
- **HDFS**: Storage layer.

6.3 Using HDFS

- Files stored through HDFS commands.
- Sits *above* the native file system.
- Accessed using commands like `hadoopfs -command`.

6.4 HDFS Node Types

- **Name Node**: Metadata manager (file → blocks → data nodes).
- **Data Nodes**: Actually store the blocks.

6.5 Name Node Details

- Maintains the namespace.
- Stores metadata only.
- Failure of name node → Catastrophic.

6.6 Data Node Details

- Store blocks as files.
- Maintain checksum and generation stamp for each block.
- Send periodic heartbeats to name node.

6.7 Writing to HDFS: Block Addition

1. Client asks name node for block allocation.
2. Name node returns list of data nodes.
3. Client sends block to first data node (DN1).
4. DN1 stores and forwards to DN2.
5. DN2 stores and forwards to DN3.
6. DN3 stores and acknowledges → Client finalizes.

6.8 Fault Recovery

- Checkpoints capture name node's state.
- Data node failures tolerated up to replication factor.

6.9 POSIX Non-Compliance

- Append-only writes.
- Simplifies replication and consistency.
- Not designed for small frequent updates.

7 HDFS and CAP Theorem

- Consistency: Name node maintains global consistency.
- Availability: Guaranteed if name node is alive.
- Partition Tolerance: Depends on replication factor.

8 Wrap-up on HDFS

- Distributed, redundant file system optimized for large, immutable datasets.
- Critical component of scalable data infrastructure.

9 Next Week

- Big data infrastructure: General principles and NYU specifics.

Big Data (DS-GA 1004) – Lecture 5 Finals Preparation Notes

Fully based on Week 5 Slides + Lecture Transcript + Expanded Details

Spring 2025

Lecture 5: NYU HPC Infrastructure and Big Data Processing

NYU HPC Research Technology Services

- Team under NYU IT responsible for research computing.
- Services provided:
  - High-Performance Computing (HPC) clusters for research and courses.
  - Big Data support, Machine Learning (ML), Deep Learning (DL), Artificial Intelligence (AI).
  - Cloud Computing support: Google Cloud Platform (GCP), Amazon Web Services (AWS).
  - JupyterHub access for courses.
  - Security Data Research Environment (SDRE) for handling sensitive data.
  - High-Speed Research Network (HSRN) for fast data transfer.
  - General research IT support (hardware advice, cloud recommendations).
- Website: `https://hpc.nyu.edu`
- Contact: `hpc@nyu.edu`

Google Dataproc Cluster (for Courses)

What is Dataproc?

- A managed Hadoop and Spark service running on Google Cloud.
- Supports HDFS, MapReduce, Spark, Hive, Trino, Pig.
- Block Size: 128 MB (optimized for large files, poor for many small files).
- Resource Management: YARN for resource allocation, job scheduling, and monitoring.
- Compared to HPC systems: YARN is simpler and more lightweight than SLURM.

Dataproc System Architecture

- Master Node: Login node (accepts SSH connections).
- Two Primary Worker Nodes: Persistent HDFS storage and compute.
- Secondary Worker Nodes (Auto-scaling): Compute-only, added/removed based on workload.

Usage for Students

- Web access: `https://dataproc.hpc.nyu.edu`
- Home Filesystem: `/home/<netid>_nyu_edu`
- HDFS Filesystem: `hdfs dfs -ls, hdfs dfs -put`
- Quota: 500 GB per user in HDFS.
- Application lifetime limit: 5 hours (Spark/YARN jobs).
- Default Spark deploy mode: `cluster` (for production).
- Debugging: `spark-shell --deploy-mode client`
- Login Node:
  - **ONLY** for job submission and debugging.
  - **NOT** for heavy computation.
  - 3 GB memory limit per user.
  - User sessions forcibly killed after 48 hours.
- Containerization: Jobs run inside containers managed by YARN.
- Logs: Output logs should go to HDFS **(not to stdout!)**

Dataproc Back-end Details

- Start with 1 TB HDFS storage, dynamically scales to 8 TB if needed.
- Ingress storage buckets available to upload very large files.
- All files deleted at semester end **(temporary cluster)**.
- Costs controlled by auto-scaling based on student usage.

Big Data on Greene HPC Cluster

Overview of Greene Cluster

- Greene: General-purpose HPC Cluster.
- Available for NYU researchers (except Langone and Abu Dhabi campuses).
- Specs:
  - 38,000 CPU cores, 220 TB memory.
  - 768 GPUs, totaling 13 TB GPU memory.
  - 12 PB parallel storage.
  - HDR 200Gb/s Infiniband network.
- Location: NYU Research Computing Center (RCDC) in Secaucus, NJ.
- Liquid-cooled NVIDIA H100 GPU servers (SD650N-V3).

Access to Greene

- Host: `greene.hpc.nyu.edu`
- 3 Load-balanced login nodes (via NYU network or VPN).
- Login methods: Terminal (Mac/Linux/Windows WSL), PuTTY, MobaXterm, VSCode Remote.
- Web access: Open OnDemand (OOD) server `https://ood.hpc.nyu.edu`

Open OnDemand (OOD)

- Browser-based access to HPC services.
- GUIs for JupyterLab, Matlab, Spark standalone cluster, Dask with Jupyter, Remote Desktop.
- Enables launching interactive jobs graphically.

Running Spark and Dask

- Spark available in both batch and standalone modes.
- Example Spark batch scripts located at:
  texttt/scratch/work/public/apps/pyspark/3.5.0/examples/spark
- Dask clusters launchable via OOD interface.

Hardware Configurations

Compute Nodes:

- Standard memory (524 nodes): 48 CPU cores, 180 GB RAM.
- Medium memory (40 nodes): 48 CPU cores, 369 GB RAM.
- Large memory (4 nodes): 96 CPU cores, 3014 GB RAM.

GPU Nodes:

- NVIDIA V100, RTX8000, A100, H100 GPUs.
- AMD MI100, MI250 GPUs.
- Multiple configurations of cores, memory, and GPU types.

Important Greene Policies

- **No compute-heavy jobs on login nodes.**
- Submit batch jobs or start interactive sessions via `srun`.
- Filesystems:
  - Home: 50 GB quota, 30,000 inode limit, backed up daily.
  - Scratch: Larger, non-backed up space for big data.
- VPN required for off-campus login (up-to-date).
- Security is strict: gateway servers exist, daily audits.

Containerization on Greene

Containers

- Singularity used instead of Docker (security reasons).
- Portable, reproducible environments for computation.
- Pack libraries, binaries, dependencies into container images.

Advantages of Containers

- Improved reproducibility (e.g., scientific papers).
- Easier compatibility across upgrades.
- Facilitates hardware/software portability.

Spark, TensorFlow, PyTorch on Greene

- Jobs can be containerized.
- Use Conda environment + Singularity + overlay filesystem.
- Manage resource requests carefully (e.g., GPUs, CPU cores, memory).
- Setup distributed training carefully (use DDP, backend specifications).

Cluster Resilience and Storage

- **Disk failures occur weekly:** Cluster tolerates with redundancy (RAID + replication).
- **Disaster recovery:** Greene storage snapshots backed up on AWS S3.
- **Redundancy:** Filesystems designed for single-point disk failure tolerance.
- **Cooling:** Liquid cooling and advanced ventilation deployed.

Wrap-up

- Big Data clusters (Dataproc) designed for course workloads **(temporary, reset each semester)**.
- Greene HPC cluster supports real research **(persistent, highly redundant)**.
- Future lectures: Submission scripts, Spark cluster configurations, GPU job handling.

Reminder: Save your work from Dataproc before semester end. All files will be erased.

DS-GA 1004: Big Data Lecture 6

Big Data Infrastructure and Introduction to Spark

1. Big Data Infrastructure

Hadoop Framework Overview

- **MapReduce**: Processing engine
- **YARN**: Resource manager
- **HDFS**: Storage layer

Evolution of Hadoop

- Hadoop 1.x: MapReduce + HDFS
- Hadoop 2.x: MapReduce + YARN + HDFS + Other engines (Spark, Flink, Hive, Pig)
- Hadoop 3.x: Kubernetes, YARN, HDFS, Cloud storage

YARN Architecture

- Components:
  - Resource Manager (Resource Scheduler + Application Manager)
  - Node Manager
  - Application Master
  - Containers
- YARN acts as the "Operating System" of Hadoop.

Cluster Resource Terminology

- **Container** (not Docker): Abstraction bundling storage, cores, and RAM.

Importance of Data Locality

- **Goal**: Bring computation to where data resides (cheaper than moving data).
- MapReduce splits input into splits; each split maps to HDFS blocks.
- Scheduler uses HDFS block locations to optimize split assignments.

Network Topology Considerations

- **Best**: Node-local execution (data and compute on same node)
- **OK**: Rack-local execution (same rack, low latency)
- **Worst**: Cross-rack execution (higher latency)

HDFS Replication

- Default replication factor: 3
- Placement: Two replicas in same rack, third in different rack.

2. Spark Introduction

Why Spark?

- **Strengths of MapReduce**: Scalability, Fault Tolerance, Commodity Hardware Friendly.
- **Limitations of MapReduce**: Unsuitable for iterative, interactive computations.

Problems with MapReduce for Iterative Algorithms

- Each iteration (e.g., gradient descent) requires full MapReduce cycle.
- High latency due to blocking between map and reduce stages.

Spark's Key Idea: Resilient Distributed Datasets (RDDs)

- **RDD** = Lineage graph of transformations + Data source + Partitioning interfaces.
- **Deferred computation**: Transformations are lazy.

RDD Operations

- **Transformations** (lazy): map, filter, join
- **Actions** (trigger execution): collect, count, reduce, save

RDD Example (Log Processing)

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.filter(_.contains("MySQL")).map(_.split('\t')(3)).collect()
```

Spark Execution Model

- Builds a dependency DAG from transformations.
- Works **backwards** from action to source.
- Caches intermediate RDDs if needed.
- Loss recovery using lineage information.

Pipelining

- Transformations can be pipelined without materializing intermediates.

Partitions and Dependencies

- **Narrow dependency**: Parent partition maps to one child partition (easy recovery, fast).
- **Wide dependency**: Parent partition maps to multiple children (shuffle, slow).

Co-Partitioning

- Preemptively partition datasets by a key to avoid costly shuffles.
- Use `repartition()` or `bucketBy()` in Spark.

Spark DataFrames

- High-level abstraction over RDDs.
- Tables with schemas, columns = RDDs.
- Queryable with SparkSQL.

Important Practices

- Always aggregate (e.g., `reduce`, `count`) before `collect()` to avoid crashing login nodes.
- Understand Spark `explain()` plans to optimize.

Week 07: Column-Oriented Storage
(Full Detailed Notes)

Introduction

Classroom Instructions:

- Please silence and put away cell phones.
- The slides are for note-taking support only, not a substitute for attending lectures.

Focus of today's lecture:

- Using Apache Spark (for distributed computation)
- Understanding column-oriented storage in big data
- Learning about Dremel and Parquet (two systems for handling structured/nested data at scale)

Core Message:
Even in a world where parallelism is important, data structures matter a lot! Organizing your data wisely is crucial for efficiency.

Part 1: Spark Review — Partitions and Dependencies

Narrow Dependencies

Definition:
Each partition of the parent RDD (Resilient Distributed Dataset) is used by at most one child RDD partition.

Properties:

- Minimal data movement (localized).
- Low communication overhead.
- Easy to pipeline operations.
- Easy failure recovery because no dependency on multiple partitions.

Result:
Fast operations because no expensive shuffle phase.

Wide Dependencies

Definition:
A partition of the parent RDD contributes to multiple child RDD partitions.

Properties:

- Requires key shuffle (expensive, time-consuming).
- High communication cost across the cluster.

- High latency.
- Difficult to pipeline.
- Harder to recover from failures.

Result:
Slow operations because data has to move between machines.

Part 2: The Problem of Wide Dependencies (and Co-Partitioning)

Example Scenario:

Three datasets:

Students (s)	Majors (m)	Academic Records (ar)
sID	mID	sID
Name	Name	Course
Major	School	Grade
Full-Time (FT) status		Credits

Goal:
Suppose we want to check if full-time students take enough credits. We need to join the Students and Academic Records tables on sID.

Problem:
A join creates a wide dependency:

- Students data is spread across partitions.
- Academic Records data is spread differently.
- Need a shuffle to match them = slow.

Solution: Co-Partitioning

Idea:
Pre-arrange (pre-shuffle) data so that records with the same join key (sID) land in the same partition. Done before the expensive operation.

In Spark:

```
s.repartition(3, "sID")
ar.repartition(3, "sID")
```

This way, same student IDs are co-located.

Example:

- Student with sID=1 goes to partition 1
- Student with sID=2 goes to partition 2
- and so on...

Important Notes:

One-time shuffle:
Co-partitioning does require a shuffle up-front, but it avoids repeated future shuffles for frequent access patterns (e.g., GPA calculation).

Key-specific Optimization:
If you co-partition by sID, operations based on sID become fast. But if you need to group by Major later, you need to repartition again.

How Spark Knows?
Spark tracks partitioning info using internal metadata and Catalyst Optimizer (written in Scala).

Types of Co-partitioning:

- `repartition()` — temporary, in-memory.
- `bucketBy()` — persistent, stored on disk.

Part 3: Using Apache Spark

Quick History:

- Developed at UC Berkeley in 2009.
- Open-sourced in 2010.
- Published (paper) in 2012.
- Released 1.0 in 2013.

Key Components:

- RDDs (Resilient Distributed Datasets) — distributed memory objects.
- Integration with Hadoop:
  - HDFS for storage.
  - YARN for scheduling jobs.
- Written in Scala, but APIs exist for:
  - Python (PySpark)
  - R
  - Java
  - MATLAB (via MDCS)

Architecture:

Driver and Sessions:

- Driver process:
  - Runs on head/login node.
- Session:
  - Connects user code to the Spark cluster.

Illustration (in words):

- Driver starts a session.
- Session coordinates distributed workers (executors).
- Cluster Manager (like YARN) allocates resources.

Spark DataFrame API

Problem with RDDs:

- Flexible but cumbersome for ad-hoc tasks.

Solution:

- DataFrames offer a structured, easy-to-use abstraction.
- Inspired by pandas (Python) and R DataFrames.

Fun Fact:
Internally, DataFrames are built on top of RDDs.

DataFrames vs RDDs:

Aspect	DataFrames	RDDs
Structure	Schema-defined columns	Raw distributed objects
Optimized?	Yes, through Catalyst	No
User Experience	Easier (pandas-like)	Harder (manual transforms)

In Spark 2.x and beyond, DataFrames are primary interface.

Spark SQL

You can query DataFrames using SQL!

Example:

```
df.createOrReplaceTempView('students')
spark.sql("SELECT Major, AVG(Credits) FROM students GROUP BY Major").show()
```

Equivalent without SQL:

```
df.groupBy('Major').avg('Credits').show()
```

Best Practice:
Always run `.explain()` to see the execution plan.

Caution:

- Avoid `.collect()` on huge data — dangerous (might crash the driver).
- Use `.take(n)`, `.save()`, `.show()` instead.

Part 4: Column-Oriented Storage

Why Care About Storage Layout?

Problem:
Storage bandwidth and latency have become bottlenecks compared to CPU speeds.

Goals:

- Reduce number of bytes transferred.
- Optimize memory access patterns.

Row-Oriented Storage (e.g., CSV)

Example:

```
id, name, mass
1, T.Rex, 8000
2, Stegosaurus, 4000
3, Ankylosaurus, 4000
```

Easy for humans.
Terrible if you want:

- Only the mass column.
- Only the 1000th record.
- Variable-length rows → no easy indexing → full scan needed.

Problem:

- Time Complexity: O(n)
- Slow random access.

Column-Oriented Storage

Key Change:
Store each column separately.

Example:

```
id: [1, 2, 3]
name: ["T.Rex", "Stegosaurus", "Ankylosaurus"]
mass: [8000, 4000, 4000]
```

Benefits:

- Accessing a single column is fast.
- Easier to compress (same data types).
- Sequential disk reads → better cache utilization.

Compression in Column Stores

Why compression matters:

- Saves space.
- Saves time (smaller data transferred).
- Sometimes enables operations on compressed data directly.

Part 5: Compression Techniques

Compression	Description	Example
Dictionary Encoding	Map frequent values to small integers	"Pop" → 0, "Rock" → 1
Bit Packing	Store small integers tightly without gaps	[0,1,0,2,1] packed
Run-Length Encoding (RLE)	Store value + repetition count	[8000, 1], [4000, 4]
Frame of Reference	Store base + offsets	1004,1005,1006 → base 1000, offsets [4,5,6]
Delta Encoding	Store first value + successive differences	1004,1005 → 1004, +1
Huffman Coding	Variable length coding based on frequency	Common values = shorter codes

Choosing compression depends on data patterns and access goals.

Part 6: Structured Data (Trees) — Dremel

Challenge with Nested Data:

Real-world data (e.g., web pages, logs) is hierarchical and nested, not tabular.

Example:

- Document ID
- Links (backward, forward)
- Names (with Language info)

Dremel's Core Ideas:

Flatten tree-structured records into a table.

Keep track of:

- Repetition Level (r): Where a repeated field occurred.
- Definition Level (d): How deeply nested the field is, counting missing optional fields.

Result:

Hierarchical data becomes queryable like columns!

Part 7: From Dremel to Parquet

Parquet (2013) = Open-source implementation of Dremel ideas.

Stores structured, nested data efficiently in columnar format.

Parquet File Layout:

- File
  - Row Group 0
    - Column Pages
  - Row Group 1

        * Column Pages
    - Footer (Metadata)

Metadata tells Spark:

- Which row groups to scan or skip.
- How data is compressed.

Part 8: Benefits and Costs of Parquet

Pros:

- Cross-language (Python, Java, C++) support.
- Only necessary columns decoded.
- Built for large distributed file systems (like HDFS).

Cons:

- Complex format (binary, not human-readable).
- Requires proper tooling to read/write.

Final Wrap-up

- Column stores are better for attribute-heavy analysis.
- Dremel lets us flatten structured/nested records.
- Parquet is the de facto standard for big data columnar storage.
- Even in big data, good data structure choices are critical.

Week 08: Dask
DS-GA 1004: Big Data
Detailed Notes for Final Exams

Introduction to Dask

Dask is an open-source Python library for parallel computing. Designed to scale from single machines to clusters, it integrates with the Python scientific stack (NumPy, Pandas, Scikit-Learn) and supports out-of-core computation.

Key Features

- **Delayed Computation**: Builds task graphs for lazy evaluation (similar to Spark RDDs).
- **Collections**: Provides distributed versions of common data structures:
  - **Bags**: Unstructured data (parallel Python lists).
  - **DataFrames**: Tabular data (Pandas-like, partitioned).
  - **Arrays**: N-dimensional arrays (NumPy-like, chunked).
- **Out-of-Core Processing**: Handles datasets larger than RAM by chunking data.

Comparison of Big Data Frameworks

Method	**Strengths**	**Weaknesses**
Collections of Files	Flexible, unstructured	No built-in parallelism
Relational Databases	SQL interface, structured	Complex parallelism
Map-Reduce + HDFS	Parallel, scalable	Restricted to map/reduce
Spark	Mature, SQL/-DataFrames, cluster-ready	Poor Python integration, rigid data model
Dask	Python-native, out-of-core, flexible	Less mature, requires manual optimization

Dask vs. Spark

Similarities

- Lazy evaluation via task graphs.
- Distributed DataFrames and collections.
- Fault tolerance through lineage.

Differences

Aspect	Dask	Spark
Language	Python-centric	JVM-based (Scala/Java)
Data Model	Prioritizes arrays (NumPy)	Prioritizes tabular (SQL)
Scaling	Single-machine out-of-core + clusters	Cluster-first
Ecosystem	SciPy stack (sklearn, PyTorch)	Hadoop ecosystem (HDFS, YARN)

Dask Collections

Bags

- Unordered collections of Python objects (analogous to Spark RDDs).
- Operations: `map`, `filter`, `foldby`.
- Example:

```
import dask.bag as db
b = db.from_sequence(range(5))
c = b.map(lambda x: x**2)
c.compute()  # [0, 1, 4, 9, 16]
```

- **Optimization Tip**: Avoid `groupby` (high shuffle); use `foldby` for associative/commutative operations.

DataFrames

- Partitioned Pandas DataFrames. Read from CSV/Parquet:

```
import dask.dataframe as dd
df = dd.read_csv('s3://bucket/*.csv')
df.groupby('column').mean().compute()
```

- **Partition Management**:
  - Repartition after filtering to avoid empty partitions.
  - Use `repartition()` and `persist()` for balanced workloads.

Arrays

- Chunked NumPy arrays for large datasets:

```
import dask.array as da
x = da.from_array(np.random.randn(2000, 6000), chunks
    =(1000, 1000))
```

- Supports most NumPy operations (e.g., slicing, reductions).

Schedulers and Execution

- **Single Machine**:
  - Threads: Shared memory, lightweight.
  - Processes: Avoid GIL, true parallelism.
- **Clusters**: Deploy on YARN, Kubernetes, or HPC systems (e.g., Greene).

Case Study: Machine Listening Evaluation

- **Problem**: Evaluate 20,000 model outputs across 10 models and 2,000 audio files.
- **Solution with Dask**:
  1. Store outputs as {model_id}/{recording_id}.txt.
  2. Use delayed functions for parallel scoring:

```
from dask import delayed
@delayed
def evaluate(file):
    # Load data, compute metrics
    return metrics
results = [evaluate(f) for f in glob('*/*.txt')]
df = dd.from_delayed(results).compute()
```

  3. Convert to DataFrame and save as Parquet.
- **Why Dask?**: Embarrassingly parallel, minimal code changes.

Best Practices

- **Minimize Shuffling**: Use combiners (`foldby`) and avoid wide dependencies.
- **Chunk Sizing**: Balance chunk size (too small → overhead; too large → memory issues).

- **Cluster vs. Single Machine**: Use clusters only when data exceeds single-node resources.

Alternatives to Dask

- **Polars**: Rust-based parallel DataFrame library (columnar, multi-threaded).
- **Modin**: Distributed Pandas replacement.

Exam Tips

- Understand when to use Dask vs. Spark (Python integration vs. mature ecosystem).
- Know how Dask handles out-of-core computation (chunking + task graphs).
- Be able to contrast Bags, DataFrames, and Arrays.

Week 09: Similarity-Based Search
DS-GA 1004: Big Data
Detailed Notes for Final Exams

Introduction to Similarity Search

The challenge of finding similar items in large datasets efficiently. Traditional brute-force approaches are computationally infeasible at scale. Key techniques include hashing, approximation, and locality-sensitive methods.

Core Concepts

Jaccard Similarity

For sets A and B, Jaccard similarity $J(A, B)$ measures overlap:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Distance Metric: $D(A, B) = 1 - J(A, B)$.

MinHash (Broder, 1997)

Approximates Jaccard similarity using hash functions. For a permutation π of all elements:

$$h(S|\pi) = \min\{\pi(k) \mid \pi(k) \in S\}$$

Key Property:

$$P[h(S_1) = h(S_2)] = J(S_1, S_2)$$

Locality-Sensitive Hashing (LSH)

Improves MinHash by grouping signatures into blocks. Parameters b (bands) and r (rows per band) control precision/recall:

$$P[\text{Collision in LSH}] = 1 - (1 - J^r)^b$$

Algorithms and Implementation

MinHash Signature Calculation

1. Generate m hash functions $H_1, H_2, \ldots, H_m$.
2. For each set S, compute $h_i(S) = \min_{x \in S} H_i(x)$.
3. Similarity is approximated by collision frequency.

LSH Workflow

1. Divide MinHash signatures into b bands of r rows.
2. Hash each band separately.
3. Candidate pairs are those sharing at least one band hash.

Extensions Beyond Sets

Ruzicka Similarity for Bags

Extends Jaccard to multisets by treating duplicates as unique elements:

$$R(A, B) = \frac{\sum \min(A_i, B_i)}{\sum \max(A_i, B_i)}$$

Cosine Similarity for Vectors

For vectors $\mathbf{u}, \mathbf{v}$:

$$\text{sim}(\mathbf{u}, \mathbf{v}) = \cos\theta = \frac{\mathbf{u}^T\mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|}$$

LSH via random hyperplanes: $h_w(\mathbf{x}) = \text{sign}(\mathbf{w}^T\mathbf{x})$.

Tradeoffs and Optimization

Method	**Advantages**	**Limitations**
Brute Force	Exact results	$O(N^2)$ complexity
MinHash	Sub-linear time, scalable	Approximate, sensitive to hash collisions
LSH	Reduces candidate set size	Requires tuning b and r

Case Study: Plagiarism Detection

- Represent documents as sets of shingles (word n-grams).
- Compute MinHash signatures for all documents.
- Use LSH to identify candidate pairs.
- Verify with exact Jaccard similarity on candidates.

Failure Modes and Mitigations

- **Stop Words:** Common words (e.g., "the") cause spurious collisions. Mitigation: Remove stop words before hashing.
- **High Dimensionality:** Curse of dimensionality affects spatial methods. Mitigation: Dimensionality reduction (e.g., PCA).

Exam Tips

- Understand the relationship between Jaccard similarity and MinHash collision probability.
- Know how LSH parameters b and r affect precision/recall tradeoffs.
- Be able to contrast MinHash, LSH, and cosine similarity techniques.
- Practice calculating Jaccard/Ruzicka similarities and interpreting hash collisions.

Appendix: Key Formulas

$$\begin{aligned}
\text{Jaccard:} \quad & J(A,B) = \frac{|A \cap B|}{|A \cup B|} \\
\text{MinHash Collision:} \quad & P[h(S_1) = h(S_2)] = J(S_1, S_2) \\
\text{LSH Collision:} \quad & P = 1 - (1 - J^r)^b \\
\text{Ruzicka:} \quad & R(A,B) = \frac{\sum \min(A_i, B_i)}{\sum \max(A_i, B_i)} \\
\text{Cosine:} \quad & \cos\theta = \frac{\mathbf{u}^T \mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|}
\end{aligned}$$

L10: Recommender Systems – Detailed Textbook Notes

1 Introduction to Recommender Systems

Recommender systems are among the most important and widespread applications of big data and machine learning today. These systems are responsible for personalized recommendations on platforms like Amazon, Netflix, Spotify, and YouTube.

Fundamentally, a recommender system aims to solve a personalized search problem: while traditional search returns the same results for everyone, a recommender system tailors results based on the user's history, preferences, and behavior.

Recommender systems thrive with big data — the more user-item interactions are recorded, the more accurate and meaningful the recommendations become.

1.1 Historical Context

- Early Internet: Users had to explicitly search for information (e.g., Yahoo in 1997).
- Today: Platforms proactively push content to users through recommendations.
- Need for personalization arises due to the diversity of human preferences: the same stimulus (e.g., a song) can evoke dramatically different reactions in different people.

2 Popularity-Based Baseline Models

Before building personalized models, it is critical to establish a simple baseline: the popularity-based model.

2.1 Definition

The popularity-based model ranks items by their average rating across all users. Mathematically:

$$p(i) = \frac{\sum_u r(u,i)}{|\{u : r(u,i) \text{ is defined}\}|} \tag{1}$$

where:

- $p(i)$ is the popularity score of item i,
- $r(u,i)$ is the rating of user u for item i,
- the denominator counts the number of users who rated i.

This is simply the mean rating per item.
Important points:

- The same ranking is shown to all users.
- This model works surprisingly well for some domains (e.g., "Top 100 Movies", "Bestsellers").

Examples:

- IMDb Top Movies: Shawshank Redemption, Godfather, Dark Knight, etc.

- Billboard Hot 100 Songs
- Barnes and Noble Bestsellers

3 Improving Popularity Models: Shrinkage

Popularity-based models can be biased if an item has very few ratings. To correct for this, we introduce a shrinkage (damping) term.

3.1 Shrinkage Formula:

$$p(i) = \frac{\sum_u r(u,i)}{|\{u : r(u,i) \text{ is defined}\}| + \beta} \tag{2}$$

where:

- β is a shrinkage hyperparameter (e.g., 5, 10, 20).

Purpose:

- Items with fewer ratings are penalized (lowered in popularity) automatically.
- Large β values strongly discount items with few ratings.
- Key idea: Ratings with small sample sizes are unreliable and should be treated cautiously.

3.2 How to Choose β

- Use cross-validation on a holdout set to select the best β.
- β controls the trade-off between trusting high-rated but rarely rated items and established, frequently rated items.

4 Bias Model (Decomposed Popularity)

To improve interpretability, the bias model decomposes a rating into three components:

$$\hat{r}(u,i) = \mu + b_i + b_u \tag{3}$$

where:

- μ = global mean rating,
- b_i = item bias (item-specific offset),
- b_u = user bias (user-specific offset).

4.1 Interpretation

- Item bias: Some items are consistently rated higher/lower across users.
- User bias: Some users tend to give higher/lower ratings across items.

4.2 Important Note

This decomposition does not change the overall recommendation ranking compared to popularity-based models. It only makes the model more interpretable.

It does not improve prediction accuracy directly.

5 Collaborative Filtering

Now moving to true personalization: Collaborative Filtering (CF).

5.1 Basic Idea

Collaborative Filtering predicts a user's interests based on the interests of similar users or similar items. There are two main types:

- User-based Collaborative Filtering
- Item-based Collaborative Filtering

5.2 User-Based Collaborative Filtering

Steps:

- Find users similar to the target user.
- Recommend items that similar users liked.
- Similarity metrics: Jaccard similarity, cosine similarity, correlation, etc.

Challenges:

- Sparse data: users have rated/interacted with only a small subset of items.
- Aggregating feedback when multiple similar users give conflicting signals.

5.3 Item-Based Collaborative Filtering

Steps:

- Find items similar to the items the user has interacted with.
- Recommend similar items.
- This is more stable than user-based filtering, because items are relatively stable over time compared to users.

Example: Amazon's "Customers who bought this also bought..." system.

6 Moving Beyond Neighborhood Methods: Latent Factor Models

Neighborhood models were dominant until the early 2000s. Today, latent factor models dominate because they:

- Scale better,
- Generalize better with sparse data,
- Capture hidden structure in preferences.

6.1 Matrix Factorization

We model the ratings matrix R as:

$$R \approx U \times V^T \tag{4}$$

where:

- U is the user feature matrix (users $\times$ factors),
- V is the item feature matrix (items $\times$ factors).

Each user and item is represented as a low-dimensional vector.
The predicted rating is the dot product:

$$\hat{r}(u, i) = U_u \cdot V_i \tag{5}$$

6.2 Alternating Least Squares (ALS)

ALS Algorithm:

- Initialize U and V randomly.
- Alternately fix V and solve for U, then fix U and solve for V via Ordinary Least Squares (OLS).
- Repeat until convergence.

Advantages:

- Works well even with missing data (only observed ratings needed).
- Highly parallelizable (solving for different users/items independently).

Key: ALS solves the minimization of squared error between predicted and actual ratings.

7 Implicit Feedback Handling

In real-world systems, explicit feedback (like star ratings) is rare. Most data is implicit: clicks, play counts, browsing time, etc.

7.1 Problem with Implicit Feedback

- A click does not necessarily mean "like".
- A lack of click may not mean "dislike".

7.2 Solution: Weighted Loss

Instead of predicting the exact number of interactions, predict the existence of interaction (yes/no) and weight the loss function by the number of interactions.

Modified loss function:

$$\sum_{u,i} c(u, i)(p(u, i) - U_u \cdot V_i)^2 \tag{6}$$

where:

- $p(u, i) = 1$ if an interaction happened, 0 otherwise,
- $c(u, i) = 1 + \alpha \times$ (number of interactions),
- α is a hyperparameter controlling confidence in feedback.

8 Cold Start Problem

Cold start refers to the challenge when:

- A new item has no interactions.
- A new user has no interaction history.

Solutions:

- Use content-based models (metadata like genres, categories, etc.).
- Request initial preferences during signup (e.g., favorite genres).
- Actively promote new items to collect interactions.

9 Evaluation Metrics for Recommender Systems

Evaluating recommenders is tricky because traditional metrics like RMSE are not sufficient. Instead, ranking metrics are used.

9.1 Key Metrics

Metric	Use case
Mean Squared Error (MSE)	Regression settings (early recommenders)
Area Under the Curve (AUC)	Ranking positive interactions ahead of negatives
Mean Average Precision (MAP)	Quality of top-ranked items
Mean Reciprocal Rank (MRR)	First relevant item matters most (e.g., autoplay)
Normalized Discounted Cumulative Gain (NDCG)	Graded relevance + ranking order

10 Normalized Discounted Cumulative Gain (NDCG)

10.1 Purpose

- Takes into account not just whether items are relevant, but how relevant they are.
- Higher scores are given for placing highly relevant items higher in the ranking.

10.2 Formula

The Discounted Cumulative Gain (DCG) at rank k:

$$DCG_k = rel_1 + \sum_{i=2}^{k} \frac{rel_i}{\log_2(i)} \tag{7}$$

The Normalized DCG (NDCG):

$$NDCG_k = \frac{DCG_k}{IDCG_k} \tag{8}$$

where $IDCG_k$ is the maximum possible DCG (ideal ranking).
An exponential version $(2^{rel} - 1)$ can be used to further reward highly relevant items.

Big Data L11: Beyond Set Similarity, Spatial Search, and Graph Algorithms

1 Beyond Simple Sets: Multi-sets and Ruzicka Similarity

In many cases, it is not sufficient to know whether an item appears in a set; we must know how often it appears. These are bags or multi-sets.

1.1 Ruzicka Similarity

An extension of Jaccard similarity for bags (multi-sets). Instead of treating each element as binary (present/absent), Ruzicka compares counts.

Formula:

$$R(A, B) = \frac{\sum_i \min(A[i], B[i])}{\sum_j \max(A[j], B[j])} \tag{1}$$

where:

- $A[i]$, $B[i]$ are the counts of item i in bags A and B.

Example: Given

$$X = [1, 3, 5, 7], Y = [2, 3, 1, 6] \tag{2}$$

Compute:

$$\sum \min(X, Y) = 1 + 3 + 1 + 6 = 11 \tag{3}$$

$$\sum \max(X, Y) = 2 + 3 + 5 + 7 = 17 \tag{4}$$

Thus:

$$R(X, Y) = \frac{11}{17} \approx 0.65 \tag{5}$$

2 Beyond Sets: Spatial Similarity Search (Vector Databases)

In modern applications (e.g., document embeddings), objects are represented as high-dimensional vectors. Cosine similarity is often used to measure closeness.

Cosine Similarity Formula:

$$\text{cosine similarity}(a, b) = \frac{a^T b}{\|a\|\|b\|} \tag{6}$$

Important Cases:

- $\cos(0°) = 1$ (same direction, highly similar)
- $\cos(90°) = 0$ (orthogonal, unrelated)
- $\cos(180°) = -1$ (opposite directions)

3 Locality Sensitive Hashing (LSH) for Cosine Similarity

Motivation: Computing cosine similarity between billions of vectors ($O(Nd)$) is expensive.
LSH reduces the search space by hashing similar vectors into the same bucket.

3.1 Charikar's LSH (2002):

Idea:

- Pick a random hyperplane (random vector w).
- Define hash function:

$$h_w(x) = \begin{cases} 1, & \text{if } w^T x \geq 0 \\ -1, & \text{otherwise} \end{cases} \tag{7}$$

Collision Probability:

- Probability that two vectors u, v hash to the same value:

$$\Pr[h(u) = h(v)] = 1 - \frac{\theta}{\pi} \tag{8}$$

where θ is the angle between u and v.
More aligned vectors (small θ) are more likely to collide.
Multiple Projections:

- Use m random hyperplanes to improve precision.
- Probability of at least one collision:

$$1 - \left(\frac{\theta}{\pi}\right)^m \tag{9}$$

4 Relevance from Graph Structure: PageRank

Early search engines ranked documents based on text match alone, vulnerable to spam attacks.
PageRank introduced network structure into search relevance:

- Trusted pages are those with many incoming links from other trusted pages.

5 PageRank Model: The Random Surfer

Treat the web as a directed graph:

- Nodes = web pages
- Edges = hyperlinks

Model user behavior as a random walk:

$$P[\text{next page } v \mid \text{current page } u] = \frac{1}{\text{out-degree}(u)} \tag{10}$$

Goal: Find the steady-state distribution over pages.

6 Mathematical Foundation: Markov Chains

Matrix M:

$$M[v, u] = P[\text{go to } v \mid \text{at } u] \tag{11}$$

M is stochastic: non-negative, columns sum to 1.
Steady-state vector p:

$$p = Mp \tag{12}$$

p is the eigenvector of M with eigenvalue 1.

7 Computing PageRank: Power Iteration

For large graphs:

- Initialize $p^{(0)}$ to uniform.
- Iterate:

$$p^{(k)} = Mp^{(k-1)} \tag{13}$$

until convergence.
This is efficient and parallelizable.

8 Handling Problems in Graphs

8.1 Disconnected Graphs

- No unique steady-state.
- Solution: Ensure graph is strongly connected.

8.2 Sinks (no outgoing links)

- Sink nodes cause probability "leaks."
- Fix: Add self-loops to sinks.

8.3 Spider Traps

- A group of pages that only link to each other traps the surfer.
- Solution: Teleportation.

9 Teleportation / Random Restart (Google's Trick)

Adjust transition matrix:

$$M' = aM + (1 - a)\frac{1}{N}ee^T \tag{14}$$

where:

- $a \approx 0.85$ (follow links with 85% probability),
- With probability $1 - a$, jump randomly to any page.

Prevents sinks and spider traps from breaking PageRank.

10 Personalized PageRank

Instead of jumping uniformly:

- Jump according to a user preference distribution q.
- Modified iteration:

$$p = aMp + (1 - a)q \tag{15}$$

Applications:

- Personalizing search results.
- Recommender systems tailored to user interests.

11 Distributed PageRank with Spark

PageRank computations at web scale use Spark:

- Leverage matrix multiplication via MapReduce.
- GraphX and GraphFrames simplify distributed graph operations.

12 Summary

Concept	**Key Idea**
Ruzicka Similarity	Extend Jaccard to bags by comparing counts
Cosine Similarity	Measure angle between vectors
LSH	Approximate nearest neighbors via random hyperplanes
PageRank	Importance from network links, not just content
Power Iteration	Efficiently find steady-state distribution
Teleportation	Fix sinks and spider traps
Personalized PageRank	Incorporate user preferences into ranking

Big Data L12: Socio-cultural Impact

Topics Covered:

- Socio-cultural impact of recommender and information retrieval systems
- Privacy issues and de-anonymization
- Introduction to Differential Privacy

1 Socio-cultural Impact of Recommender Systems

Claim:
Recommender systems might have contributed to the downfall of sites like Buzzfeed.

Issues:

- **Echo Chambers and Filter Bubbles**:
  - Recommendations rely on **similarity**.
  - Over time, diversity decreases; users herd around similar content.

Pariser, 2011 coined "filter bubble."

    - Best case: users get what they like.
    - Typical case: users get bored and leave.
    - Worst case: users become **isolated** and **polarized**.

- **Targeted Advertising**:
  - Personalization often based on user features (Age, Gender, Zip code).
  - Can lead to discrimination issues (e.g., lawsuits against Facebook).
- **Ethical Challenges in Representation**:
  - Recommenders are shaped by **biased past behaviors**.
  - Important questions:
    - How is the model biased?
    - How are atypical users treated?
    - Who truly benefits from personalization?

2 Privacy and De-anonymization

Problems with Anonymization:

- Simply **removing identifiers** (names, IDs) isn't enough.
- **Common but flawed methods**:
  - **Obfuscate identifiers**: Replacing names with random numbers.
  - **Perturb observations**: Adding random noise.
  - **k-anonymity** [Sweeney, 2002]: each attribute shared by at least k people.
  - **Only publishing summary statistics**: Still leaky!

3 De-anonymization Attacks

Netflix Prize Attack ([Narayanan & Shmatikov, 2008]):

- Netflix released "anonymized" movie rating data.
- Attack:
  1. Define **similarity** between users.
  2. Given **partial ratings**, compute similarity to users.
  3. If the match is strong, re-identify the user.
- **Result**:
  - With just **8 ratings** (allowing 2 mistakes) and 14 days timestamp fuzziness, **99%** of users could be uniquely identified!
  - Even **without timestamps**, rare movie ratings leak identity.

Why it matters:

- Preferences (movies, music) correlate with sensitive personal attributes (politics, religion, sexual orientation).
- **Privacy breaches are irreversible.**

4 Broader Examples of Privacy Violations

- **2010 US Census Attack** ([Abowd, 2019]):
  - Reconstruction attacks using public census summaries.
- **Target Pregnancy Prediction** ([Duhigg, 2012]):
  - Target inferred a teenager's pregnancy based on shopping patterns, disclosed it accidentally.

5 Differential Privacy (DP)

Concept:
If one individual's data is removed, the result of any computation should not substantially change.

- **DP is a property of algorithms, not datasets.**
- **Randomization happens at the algorithm level**, not by modifying the raw data.

Formal Definition:
For datasets D and D' differing by one record:

$$\Pr[A(D) \in S] \leq e^{\epsilon} \times \Pr[A(D') \in S] \tag{1}$$

where:

- ϵ (epsilon) controls **privacy loss**.
  - Smaller ϵ: Stronger privacy.
  - Larger ϵ: Weaker privacy but more accuracy.

Mechanism: Adding Laplace Noise

- **Sensitivity** (Δf): Maximum change one row can cause in the output.

- **Laplace mechanism**: Add noise drawn from Laplace$(0, \Delta f/\epsilon)$.
- **Why Laplace noise?**
  - Heavy tails $\rightarrow$ better protection compared to Gaussian noise.

Trade-off:

Noise Level	**Privacy**	**Accuracy**
High Noise (small ϵ)	High	Low
Low Noise (large ϵ)	Low	High

- **Larger datasets** $\rightarrow$ **easier privacy** (sensitivity decreases).

Differential Privacy in Action:

- **Sum queries** (e.g., total clicks) are less sensitive than **Max queries** (e.g., maximum income).
- Privacy loss **accumulates** over multiple queries!

6 Summary

- **Simply de-identifying data is not enough** — high-dimensional data is easy to re-identify.
- **Differential Privacy** offers a mathematically sound way to **release data while protecting individuals**.
- **Laplace noise** carefully balances **privacy and reproducibility**.