2.1 — Storage Formats & Distributed File Systems

Data Processing at Scale

2026-03-27

Data Processing at Scale — Day 2

Course website

https://vbergeron.github.io/data-processing-at-scale/

This presentation

https://vbergeron.github.io/data-processing-at-scale/2.1-storage-formats.pdf

Why do files matter ?

MapReduce (2004)

1. **Load** — read data from files into workers
2. **Map** — apply a function on each partition
3. **Shuffle** — redistribute by key across the network
4. **Reduce** — aggregate partitions into results
5. **Save** — persist results back to files

Files are the interface between every stage.

Why file format is the first optimization

The file format determines the cost of every MapReduce stage:

1. **Map** : Read speed constrained
2. **Shuffle** : Size constrained
3. **Reduce** : Write speed constrained

This stays true for any modern system.
There is not so many ways to persist data.

Format is one of **key design decisions**.
It is not an implementation details.

Distributed file systems

Where do the files live?

- Your laptop: one disk, one filesystem, one failure domain
- A cluster: thousands of disks, across racks and datacenters

The data doesn't fit on one machine. We solved this in session 1.3.


hadoop

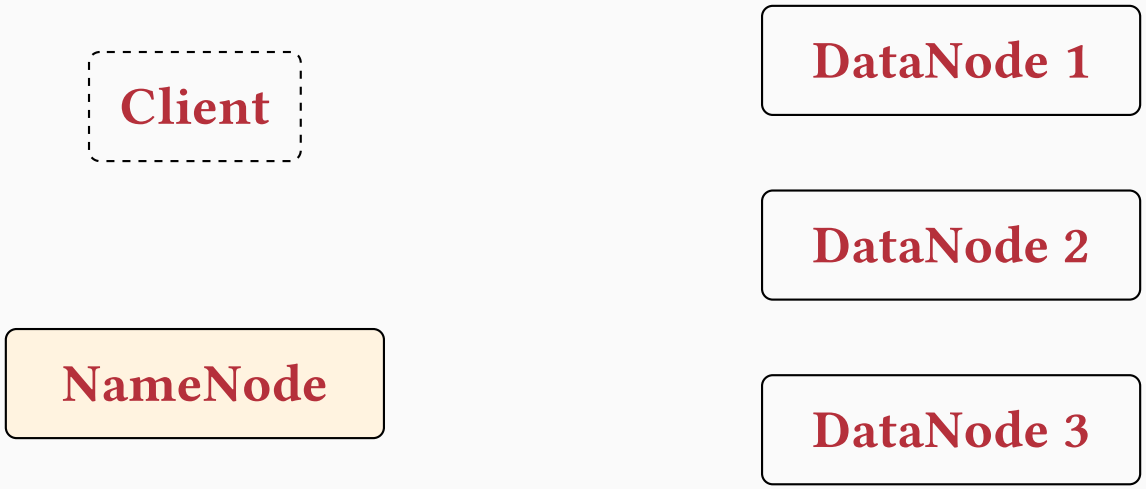
GFS / HDFS

Google File System (2003) and its open-source clone HDFS (2006):

- Files are split into **blocks** (64–256 MB each)
- Each block is replicated (default: **N copies** on different racks)

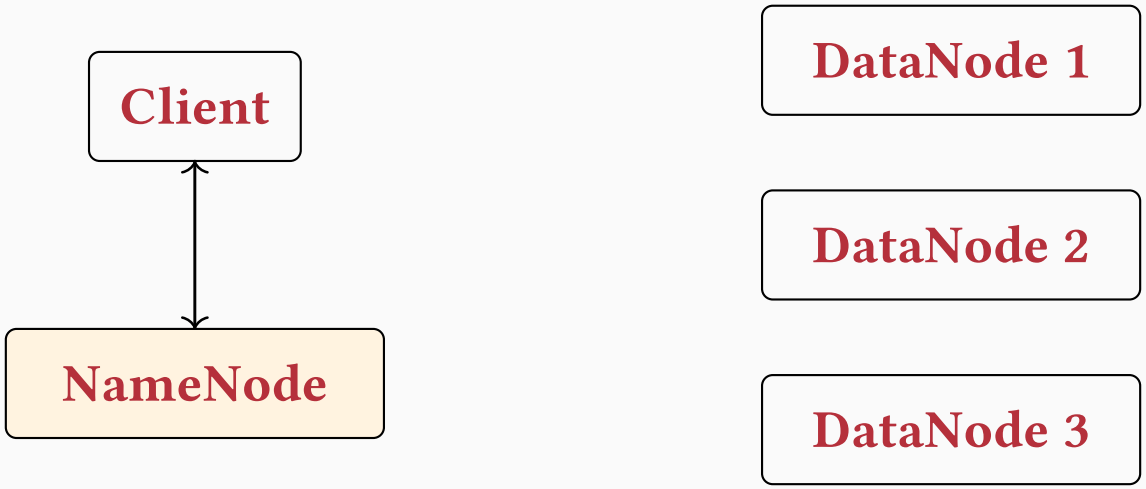
This is the foundation that made MapReduce, and later Spark, possible.

HDFS architecture



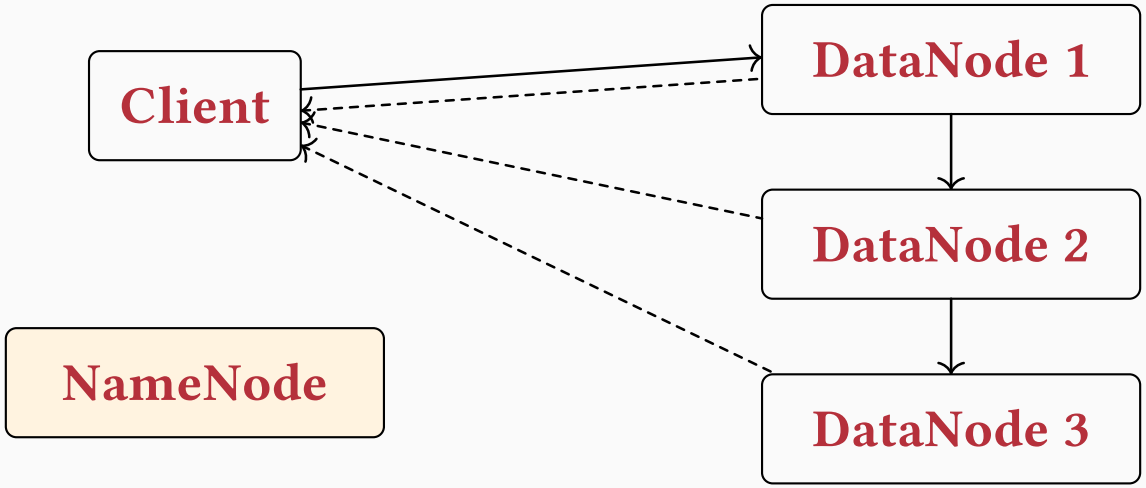
- **NameNode**: stores the metadata — file names, block locations, permissions
- **DataNodes**: store the actual blocks on local disks, send heartbeats to the NameNode

HDFS — the write path



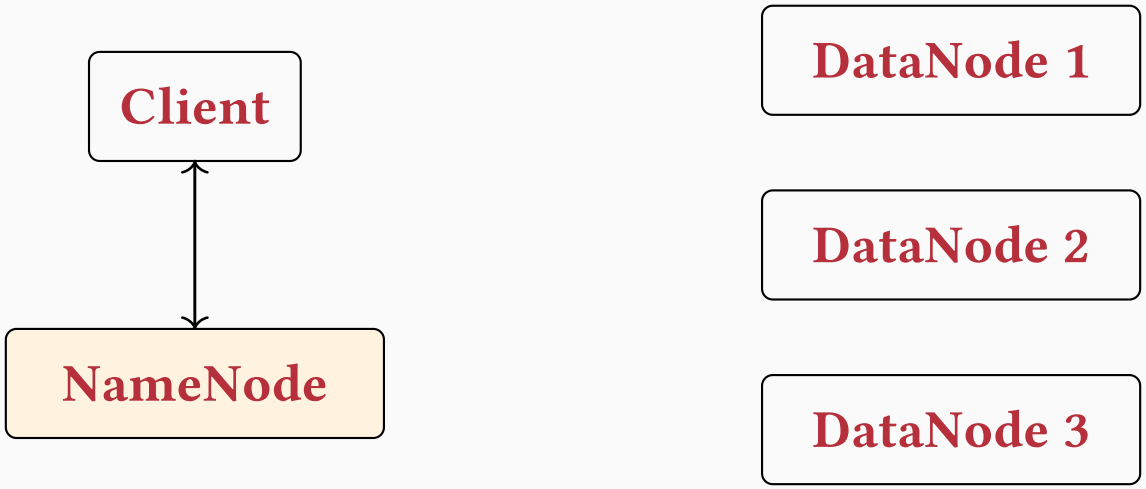
1. **Client** ask the **NameNode** for block allocation

HDFS — the write path



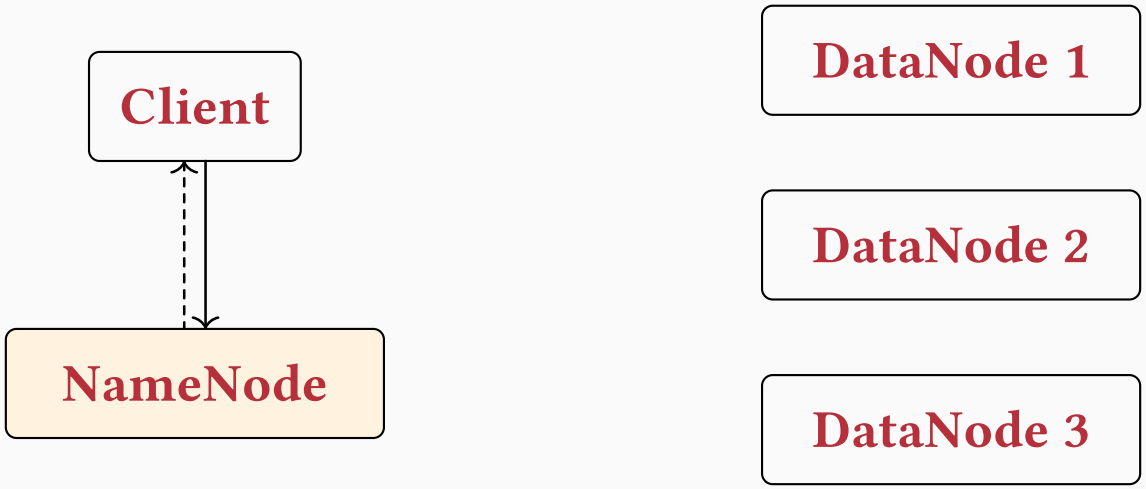
2. **Client** push data on the first **DataNode**
   1. Replication in pipelined to reduce bandwidth
   2. Every **DataNode** acknoledge to the **Client**

HDFS — the write path



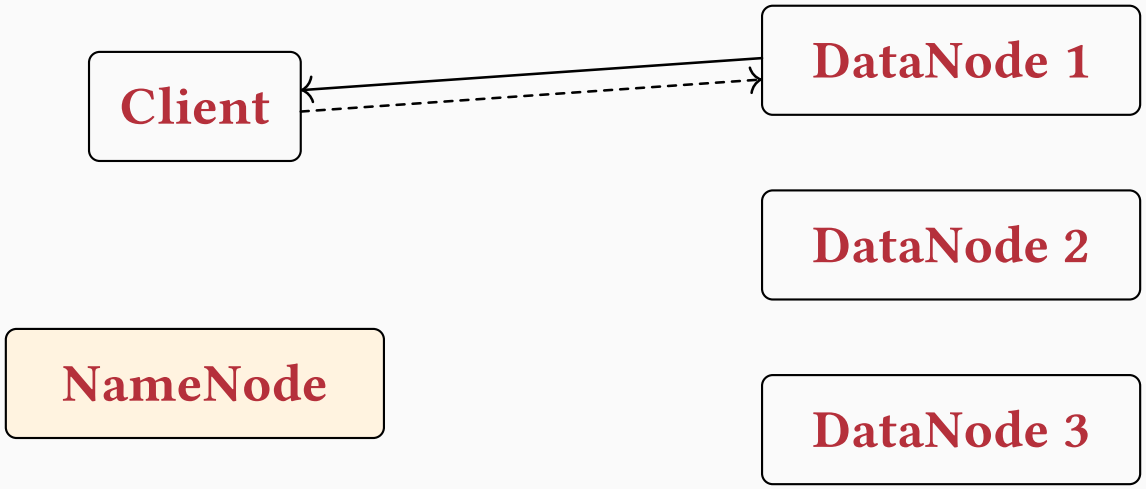
3. **Client** notify the **NameNode** the transfer is complete

HDFS — the read path



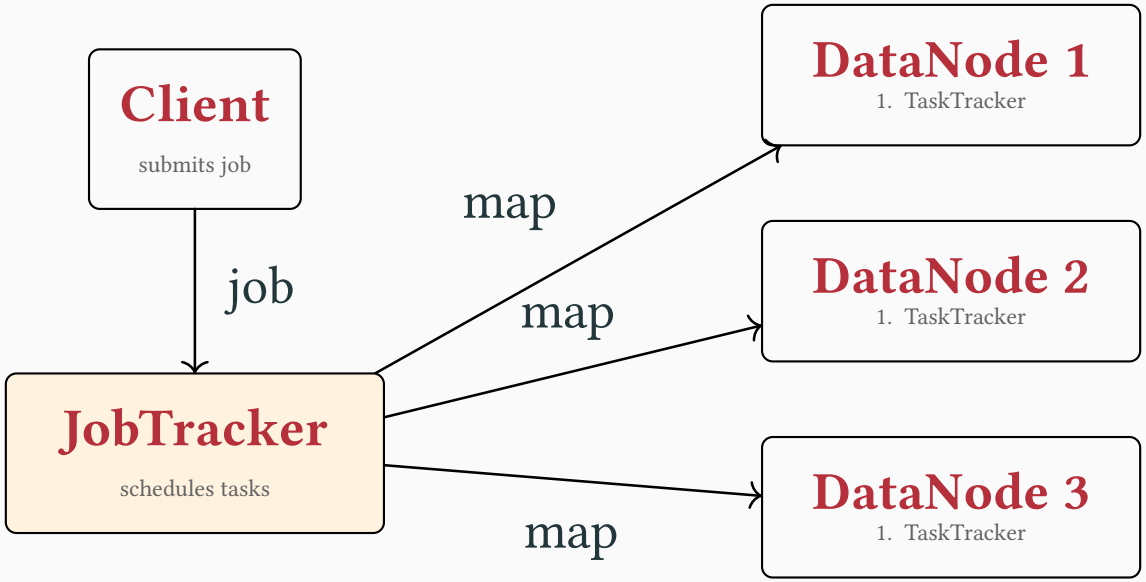
1. **Client** asks the **NameNode** for the block list and their locations

HDFS — the read path



2. **Client** reads directly from the **closest** DataNode

HDFS & MapReduce — data locality



- Each DataNode runs a **TaskTracker** — a worker that executes map and reduce tasks
- The **JobTracker** schedules each map task on the DataNode that holds the blocks it needs
- **Data locality**: input data never crosses the network for the map phase

HDFS — the NameNode problem

The NameNode is a **single point of failure** and a **scalability bottleneck**:

- All metadata in RAM — one NameNode must hold the entire namespace
- 150 bytes per block in memory $\longrightarrow$ 100 million blocks $\approx$ 15 GB of RAM
- If the NameNode dies, the cluster is **down** — data is intact on DataNodes but unreachable

In Day 1 terms: the NameNode is a **leader** with no **failover**. A crash is a total outage.

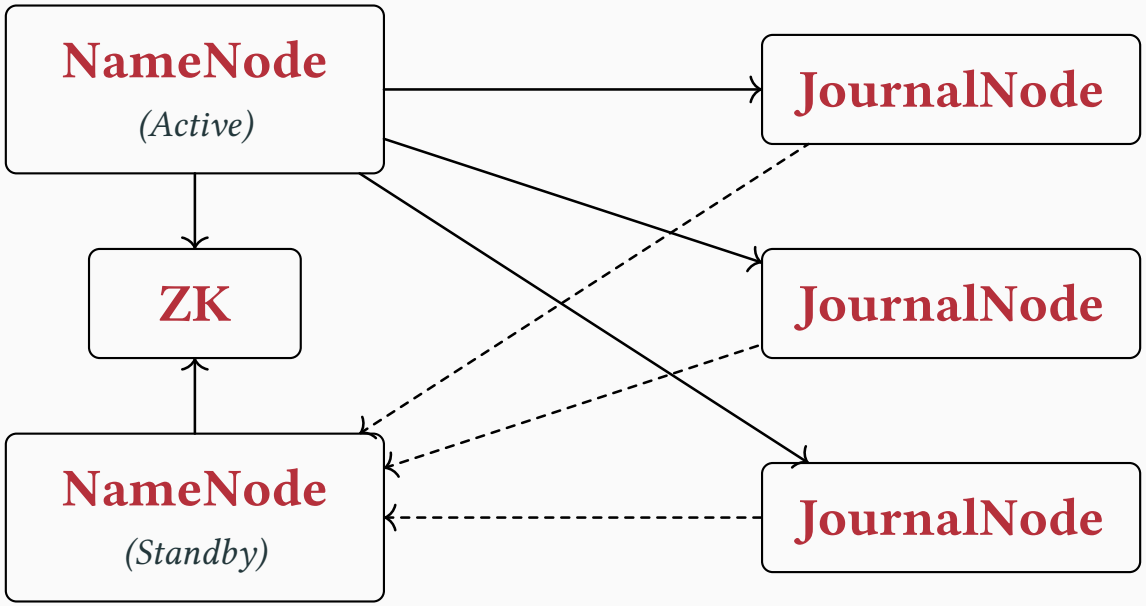
HDFS High Availability

HDFS HA uses a **Standby NameNode** and **JournalNodes**.
NameNodes append and read from a shared edit log stored on **JournalNodes**

- **Active NameNode**: appends to the edit log.
- **Standby NameNode**: replays the edit log.
- Both are using **Quorum** read or writes.

ZooKeeper's role: decide **which** NameNode is active.

HDFS High Availability



APACHE
ZooKeeper™

ZooKeeper — the coordination service

ZooKeeper is a **CP** system (recall Day 1: consistent + partition-tolerant, sacrifices availability):

- A small cluster (3 or 5 nodes) running a **consensus protocol** (ZAB, similar to Paxos/Raft)
- Provides: distributed locks, leader election, configuration, group membership
- **Linearizable writes** — with optional linearizable read: (issue `sync()` first)

ZooKeeper and the split-brain problem

ZooKeeper prevents **split-brain** with **fencing**:

1. Active NameNode holds an **ephemeral lock** (ZooKeeper znode)
2. If the active NameNode crashes, ZooKeeper detects the lost heartbeat and deletes the lock
3. Standby acquires the lock and becomes active
4. A **fencing** mechanism ensures the old active cannot issue writes (e.g., SSH kill, shared storage revocation)

ZooKeeper's quorum ensures the lock is never granted to two nodes at once — making **automatic failover** safe.

HDFS Federation — scaling the namespace

Even with HA, one NameNode holds the **entire** namespace. At petabyte scale, this becomes the bottleneck.

HDFS Federation: multiple independent NameNodes, each owning a **namespace partition** (called a block pool).

- `/data/transactions/` → NameNode A
- `/data/logs/` → NameNode B
- DataNodes serve blocks for **all** NameNodes

In Day 1 terms: the namespace is **partitioned** across NameNodes, while data blocks are **replicated** across DataNodes. Two orthogonal concerns, just like we saw for databases.

HDFS through the Day 1 lens

Day 1 concept	How it shows up in HDFS
Partitioning	Files split into blocks; Federation partitions the namespace
Replication	Each block stored on 3 DataNodes (different racks)
Leader / follower	Active NameNode (leader) + Standby (follower)
Failover	ZooKeeper-based leader election on NameNode crash
Split-brain	Prevented by ZooKeeper fencing
Quorum	JournalNodes (edit log) and ZooKeeper (leader election)
CAP trade-off	NameNode is CP — consistent metadata, unavailable during failover
Data locality	Scheduler places compute where replicas live — avoiding the network

Object storage — the cloud successor

S3, GCS, Azure Blob Storage replaced HDFS in most modern stacks:

- Immutable **objects** (files) in **buckets** (directories)
- No block-level control — you read ranges of bytes
- Virtually unlimited capacity, pay-per-GB

Trade-off: no data locality. Compute and storage are **decoupled**.

Object storage — consistency model

Object storage is now **strongly consistent** for all clients:

- S3: strong consistency for all operations since December 2020
- GCS and Azure Blob: strongly consistent from the start
- After a successful `PUT` or `DELETE`, any client immediately sees the new value

Strong consistency here is **per-object** — there is no multi-object transaction.
Concurrent writes to the same key are last-writer-wins.

HDFS vs object storage — when it matters

Property	HDFS	Object storage (S3)
Data locality	Yes — compute on data node	No — data travels over network
Mutability	Append-only	Immutable objects
Cost	Cluster runs 24/7	Pay per GB stored + read
Scalability	NameNode bottleneck	Virtually unlimited
Ecosystem	Hadoop, on-prem Spark	Everything modern

Most new systems target object storage.

File format choice matters **more** — every byte you read crosses the network.

Let’s start with formats you already know.

Text Formats

CSV — the universal lowest common denominator

```
user_id,name,amount,currency,timestamp
42,Alice,19.99,EUR,2024-03-15T10:30:00Z
43,Bob,1250.00,USD,2024-03-15T10:31:12Z
```

- No standard. RFC 4180 exists — almost nobody follows it exactly.
- No types: is `42` an integer, a string, a float?
- No nested structures
- Delimiter collisions: what if a field contains a comma?

CSV — what it's good at

CSV is often a pragmatic choice for human-interrop.

- **Universal**: every tool reads CSV. Excel, Python, `awk`, databases.
- **Streamable**: you can process line-by-line without loading the whole file
- **Debuggable**: open it in a text editor (or better, with SQLite)

JSON — self-describing and nested

```
{
    "user_id": 42,
    "name": "Alice",
    "amount": 19.99,
    "currency": "EUR",
    "address": {
        "city": "Paris",
        "zip": "75001"
    }
}
```

- **Self-describing**: field names travel with the data
- **Nested**: objects and arrays — richer than flat CSV
- **Typed** (partially): numbers, strings, booleans, null — but no integers vs floats, no dates

JSON — the cost of self-description

```
{"user_id": 42, "name": "Alice", "amount": 19.99}
{"user_id": 43, "name": "Bob", "amount": 1250.00}
{"user_id": 44, "name": "Carol", "amount": 7.50}
```

Field names can account for **more bytes than the data itself**.

NDJSON — JSON for pipelines

Newline-Delimited JSON: one JSON object per line.

```
{"user_id": 42, "name": "Alice", "amount": 19.99}
{"user_id": 43, "name": "Bob", "amount": 1250.00}
```

- **Splittable**: any line boundary is a valid split point
- **Appendable**: just add a line
- **Streamable**: process record by record

This is the default format for GH Archive, log pipelines, and most streaming ingestion.

Text formats trade efficiency for readability.
At scale, that trade stops being worth it.

Binary Row Formats

What “row-oriented binary” means

Each record is serialized as a contiguous byte sequence.
All fields of one row, then all fields of the next.

```
[row 0: user_id | name | amount | currency | timestamp]
[row 1: user_id | name | amount | currency | timestamp]
[row 2: user_id | name | amount | currency | timestamp]
```

Good for: writing whole records, reading whole records, streaming one-by-one.
Bad for: reading a single column across millions of rows.

The design axes

Binary row formats differ on four axes:

Axis	Question
Schema	Is the schema embedded, external, or negotiated?
Evolution	Can readers handle writers with a different schema version?
Zero-copy	Can you access fields without deserializing the whole record?

MessagePack & CBOR — schemaless binary

MessagePack: "JSON but binary." Same data model (maps, arrays, scalars), 30% smaller.

CBOR (RFC 8949): IETF standard, same idea, richer type system (dates, binary blobs, tags).

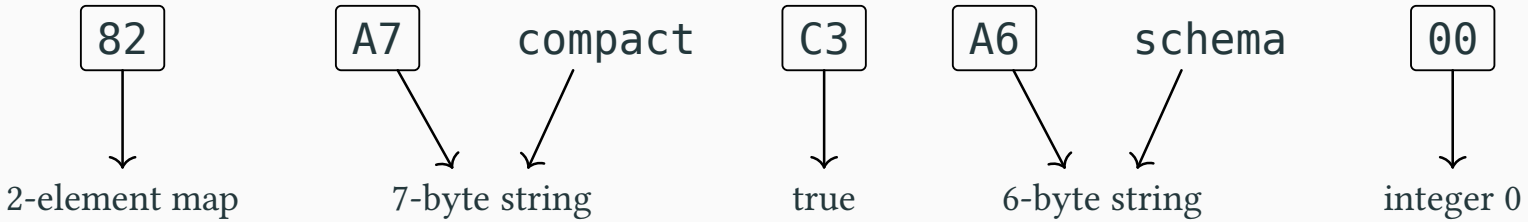
- No schema, no evolution story — same flexibility (and same problems) as JSON
- Good for: caching, inter-service messages, embedded systems
- Not designed for analytical workloads

MessagePack & CBOR — schemaless binary

JSON (27 bytes):

```
{ "compact": true, "schema": 0 }
```

MessagePack (18 bytes):



Protocol Buffers — schema-first, evolution-safe

Google's serialization format. Schema defined in `.proto` files:

```
message Transaction {
  int64 user_id = 1;
  string name = 2;
  double amount = 3;
  string currency = 4;
}
```

- Fields identified by **number**, not name → very compact on the wire
- **Backward compatible**: new code can read old data (unknown fields are ignored)
- **Forward compatible**: old code can read new data (new fields get default values)

Both directions work — as long as you follow the evolution rules.

Avro — the Hadoop ecosystem standard

Apache Avro embeds the **writer's schema** in the file header:

```
{"type": "record", "name": "Transaction",
 "fields": [
   {"name": "user_id", "type": "long"},
   {"name": "name", "type": "string"},
   {"name": "amount", "type": "double"}
 ]}
```

- Reader provides its own schema → **schema resolution** at read time
- No tag numbers — fields matched by **name**
- The default serialization for Hadoop, Kafka

Avro — schema evolution guarantees

Fields are matched by **name**, not number. That changes the compatibility story:

- **Backward compatible** (new code reads old data): safe if new fields have a default value — missing fields use the default.
- **Forward compatible** (old code reads new data): safe if removed fields had a default — old readers skip unknown fields.
- **Both directions**: only guaranteed if every field always carries a default.

Renaming a field **breaks compatibility** unless you declare an `alias`.
Changing a type always breaks.
There is no equivalent of Protobuf's "never reuse tag numbers" — name stability **is** the contract.

The schema travels **with the file**. No external schema registry needed (but one helps at scale).

Zero-copy formats — FlatBuffers & Cap'n Proto

Most serialization formats require full deserialization before accessing any field.

FlatBuffers (Google) and **Cap'n Proto** (author of Protobuf v2):

- Access fields directly from the serialized buffer — no unpacking step
- Memory layout **is** the wire format
- Ideal when read latency matters more than file size

Trade-off: larger on disk (alignment padding), more complex schemas.
Used in: game engines, mobile apps, some high-frequency trading systems.
Rare in data pipelines.

SQLite as a file format

SQLite is **the most deployed database engine in the world** — more instances than all other databases combined.

- One `.db` file, zero server, zero setup, zero dependencies
- Universally readable: every language has a SQLite driver
- Full SQL: filtering, joins, aggregations, window functions
- B-tree indexes: point lookups in O(log n) without scanning the file
- ACID transactions — crash-safe by default

The row format landscape

Format	Schema	Evolution	Zero-copy
MsgPack / CBOR	None	None	No
Protobuf	External	Tag-based	No
Avro	Embedded	Name-based	No
FlatBuffers	External	Tag-based	Yes
Cap'n Proto	External	Tag-based	Yes
SQLite	Embedded	DDL-based	B-tree

Row formats are optimized for

writing whole records.

What if you mostly read single columns?

Columnar Formats

The insight — row-oriented

Array of objects: every record is self-contained.

```
[
  {"id": 1,  "name": "Alice", "amount": 19.99,   "currency": "EUR"},
  {"id": 2,  "name": "Bob",   "amount": 7.50,    "currency": "EUR"},
  {"id": 3,  "name": "Alice", "amount": 340.00,  "currency": "EUR"},
  {"id": 4,  "name": "Carol", "amount": 0.99,    "currency": "EUR"},
  {"id": 5,  "name": "Bob",   "amount": 1250.00, "currency": "USD"},
  {"id": 6,  "name": "Carol", "amount": 88.00,   "currency": "USD"},
  {"id": 7,  "name": "Alice", "amount": 15.00,   "currency": "USD"},
  {"id": 8,  "name": "Bob",   "amount": 4200.00, "currency": "GBP"},
  {"id": 9,  "name": "Carol", "amount": 530.50,  "currency": "GBP"},
  {"id": 10, "name": "Alice", "amount": 2.00,    "currency": "GBP"}
]
```

To read `amount`, you must parse **every byte of every record**.

The insight — column-oriented

Object of arrays: each field is stored as one contiguous block.

```
{
  "id":       [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
  "name":     ["Alice","Bob","Alice","Carol","Bob",
               "Carol","Alice","Bob","Carol","Alice"],
  "amount":   [19.99, 7.50, 340.00, 0.99, 1250.00,
               88.00, 15.00, 4200.00, 530.50, 2.00],
  "currency": ["EUR","EUR","EUR","EUR","USD","USD","USD","GBP","GBP","GBP"]
}
```

To read `amount`, you seek to **one** contiguous region and read only those bytes.

Why columnar wins for analytics

Analytical queries touch few columns across many rows:

```
SELECT currency, SUM(amount) FROM transactions
WHERE date > '2024-01-01'
GROUP BY currency
```

- Only 3 columns used out of potentially dozens
- **Projection pruning**: read only `currency`, `amount`, `date`
- **Predicate pushdown**: skip row groups where `date <= '2024-01-01'`

Why columnar compresses better

Shannon's entropy formula:

$$H = -\sum_{i} p_i \log_2 p_i$$

Fewer distinct values $\longrightarrow$ higher p_i $\longrightarrow$ lower H $\longrightarrow$ fewer bits needed.

- **Column** `["EUR","EUR","EUR","EUR","USD","USD","USD","GBP","GBP","GBP"]` — low H, compresses well
- **Row** `{id, name, amount, currency}` — high H, compresses poorly

Dictionary encoding — low cardinality columns

`currency` has few distinct values — store a lookup table instead of repeating strings.

```
raw:        ["EUR","EUR","EUR","EUR","USD","USD","USD","GBP","GBP","GBP"]
dictionary: {0: "EUR", 1: "USD", 2: "GBP"}
codes:      [0, 0, 0, 0, 1, 1, 1, 2, 2, 2]
```

10 strings ⟶ 10 small integers + 3 strings. Compression ratio grows with repetition.

Delta encoding — monotonic or slowly changing columns

`user_id` increases by small, regular steps — store differences instead of absolute values.

```
raw:     [1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, 1009, 1010]
delta:   [1001,   +1,   +1,   +1,   +1,   +1,   +1,   +1,   +1,   +1]
delta²:  [1001,   +1,    0,    0,    0,    0,    0,    0,    0,    0]
rle:     base=1001, delta=1, count=10   ← 3 values total
```

When deltas are constant, delta-of-delta collapses them to zero — then RLE reduces the whole sequence to 3 numbers. Parquet and Gorilla use this for timestamps: nanosecond precision at fixed cadence compresses to 1 bit per value.

Run-length encoding — repeated values

Dictionary codes are already integers — RLE stacks on top naturally.

```
codes:   [0, 0, 0, 0, 1, 1, 1, 2, 2, 2]   ← currency after dict encoding
encoded: [(0, 4), (1, 3), (2, 3)]           ← (code, count) pairs
```

10 integers ⟶ 3 pairs.

Sorting can have massive impact on compression ratio within row groups.

Using all techniques together

```
{
  "id":       {"enc": "delta2+rle", "base": 1,
                                    "delta": 1,
                                    "runs": [[1,10]]},

  "name":     {"enc": "dict",      "dict": ["Alice","Bob","Carol"],
                                   "codes": [0,1,0,2,1,2,0,1,2,0]},

  "amount":   {"enc": "plain",     "values": [19.99,7.5,340.0,0.99,1250.0,
                                              88.0,15.0,4200.0,530.5,2.0]},

  "currency": {"enc": "dict+rle", "dict": ["EUR","USD","GBP"],
                                   "runs": [[0,4],[1,3],[2,3]]}
}
```

Apache Parquet

The dominant columnar format in the data ecosystem.
Created by Twitter and Cloudera (2013), now an Apache project.

Used by: Spark, Flink, Trino, DuckDB, Snowflake,
BigQuery, Athena, Pandas, Polars, …

Parquet file structure

A Parquet file is organized in three levels:

- **File** → contains one or more **row groups**
- **Row group** → a horizontal slice (typically 128 MB), contains **column chunks**
- **Column chunk** → all values for one column in that row group, split into **pages**

The **footer** contains schema, row group locations, and column statistics.

The reader reads the footer first, then seeks to only the columns and row groups it needs.

Page encodings — where compression happens

Within a column chunk, values are split into pages (1 MB). Each page can use a different encoding:

- **Plain**: raw values, no compression
- **Dictionary**: replace values with integer codes (great for low-cardinality strings)
- **Run-length encoding (RLE)**: collapse repeated values (great for sorted data)
- **Delta encoding**: store differences between consecutive values (great for timestamps, IDs)
- **Bit-packing**: use fewer bits when values are small

Encodings are chosen per-column, per-page. The writer picks the best one automatically.

Column statistics — skip without reading

Each column chunk stores **statistics** in the footer:

- **min / max** value in that chunk
- **null count**
- **distinct count** (optional)

Query: `WHERE amount > 1000`

If a column chunk's max is 500, the entire row group is **skipped**. No bytes read.

This is **predicate pushdown** — the format does filtering for free.

Nested data in Parquet

Parquet supports nested structures using Dremel's **repetition** and **definition levels**:

```
{"name": "Alice", "orders": [{"item": "book", "price": 9.99},
                             {"item": "pen", "price": 1.50}]}
{"name": "Bob",   "orders": []}
{"name": "Carol", "orders": null}
{"name": "Dave",  "orders": [{"item": "cup",  "price": 3.50}]}
{"name": "Eve",   "orders": [{"item": "hat",  "price": 12.00},
                             {"item": "bag", "price": 25.00}]}
```

- **Definition level** (d): how many optional/repeated ancestors are actually present
- **Repetition level** (r): which repeated ancestor restarted (0 = new record)

Dremel encoding — decoded

Schema: `name` (required), `orders` (repeated group), `item` and `price` (both optional). Siblings share the same r/d levels. Max: $d_{max}=2$, $r_{max}=1$.

Record	orders.item	orders.price	r	d	Meaning
Alice	`"book"`	`9.99`	0	2	new record, full order
Alice	`"pen"`	`1.50`	1	2	continued list
Bob	—	—	0	1	orders present but empty
Carol	—	—	0	0	orders is null
Dave	`"cup"`	`3.50`	0	2	new record, single order
Eve	`"hat"`	`12.00`	0	2	new record, first order
Eve	`"bag"`	`25.00`	1	2	continued list

r=0 always marks a new top-level record. d encodes how deep the value actually exists.

Apache Arrow — the in-memory columnar standard

Arrow is **not a file format** — it's a **memory layout specification**:

- Defines how columnar data lives **in RAM** across languages
- Zero-copy sharing between processes — no serialization step
- Thriving ecosystem of libraries and tools

Apache Arrow — IPC and the processing model

Arrow IPC / Feather: on-disk serialization of Arrow buffers.

- Memory-mappable: the file *is* the in-memory layout — open and use directly
- No deserialization cost
- Feather v2 supports per-column compression (LZ4 or ZSTD)
- No column statistics ⟶ no predicate pushdown

Think of it as: **Parquet for persistence, Arrow for processing**.

Towards new formats

Parquet was built in the early 2010s for the Hadoop era. Since then, both hardware and workloads have changed:

- **Hardware**: NVMe SSDs, wide SIMD (AVX-512), GPUs
- **Storage**: S3-first stacks where every read is a network call
- **Workloads**: point lookups, embeddings, images, ML features

Lance — random access without compromise

Built for AI pipelines on NVMe storage (LanceDB, 2023):

- **Repetition index**: random access in **1–2 IOPS** per lookup, independent of nesting depth (Parquet: hundreds of IOPS)
- **Dual encoding**: "full zip" for fast scans, "miniblock" for random access — same file, two read paths
- **Versioned updates**: append-only log with fast deletes — no rewriting files
- **Native vector search**: built-in ANN index for embedding lookups

Matches Parquet on full scans; dramatically outperforms it on point lookups.

Apache Vortex (incubating) — a framework, not just a format

Less a single format, more a **framework for composable columnar encodings** (SpiralDB → Apache, 2024):

- **Composable encodings**: FSST for strings, ALP for floats, dictionary for categoricals — mix and match per column
- **SIMD-native decompression**: designed to saturate modern memory bandwidth
- **Compressed execution**: keeps data compressed in Arrow arrays — no decompress-then-process step
- **2–10× faster scans** than Parquet, **100–200× faster random access**

Philosophy: no single compression scheme is best for all data types. Let the format adapt.

F3 — the academic successor (SIGMOD 2026)

Future File Format (F3) — a CMU/Tsinghua project with Wes McKinney (creator of Pandas and Arrow) as co-author:

- **Embedded WASM decoders**: every file carries its own decoding logic — any reader can decode any encoding, forever
- **Decoupled I/O units**: independent of row group size, tuned per storage medium (8 MB default for S3)
- **Flexible dictionary scope**: per-column dictionary granularity instead of Parquet's one-per-row-group
- Published at SIGMOD 2026 — the premier database systems venue

Key insight: the format is **extensible by design** — new encodings are Wasm plugins, not spec changes. The interoperability problem that plagues Parquet v2 adoption is solved architecturally.

Common themes across new formats

Theme	What it means
SIMD & GPU-first design	Encodings built to exploit vector instructions and parallel hardware
Sub-row-group granularity	Decompress 1K values at a time, not 1 MB pages
Composable lightweight codecs	Chains of simple encodings (RLE, bit-pack, dictionary) beat one heavyweight codec
AI/ML-native	First-class support for embeddings, vectors, random access
S3/NVMe-aware I/O	Tuned for object storage round trips and NVMe parallelism

Research formats to also watch: **BtrBlocks** (TUM — cascaded lightweight compression), **FastLanes** (CWI — compressed execution for DuckDB/Velox).

Parquet manage the data.
Who manages the **metadata**?

Lakehouse Table Formats

The problem Parquet doesn't solve

You have a data lake: thousands of Parquet files in S3.

- How do you know which files belong to the `transactions` table?
- You add a column — how do old files get read with the new schema?
- A write fails halfway — some files are written, some aren't. Is the table consistent?
- You need to delete GDPR-regulated rows. Parquet files are immutable.

Parquet is a **file** format. You need a **table** format.

What a table format does

A metadata layer on top of immutable Parquet (or ORC) files:

- **Catalog**: which files constitute the current table
- **Schema evolution**: add, rename, reorder columns across file versions
- **ACID transactions**: atomic writes — readers never see partial results
- **Time travel**: query the table as it was at any point in the past
- **Partition evolution**: change partitioning without rewriting data

Apache Iceberg

Created by Netflix (2017), now the most widely adopted table format.

- **Snapshot isolation**: readers see a consistent snapshot; writers commit atomically
- **Hidden partitioning**: partition strategy is metadata, not directory structure — `WHERE date = '2024-03-15'` works without knowing the partition scheme
- **Schema evolution**: add, drop, rename, reorder columns — all via metadata updates
- **Time travel**: `SELECT * FROM t AS OF '2024-01-01'`

Adopted by: Spark, Flink, Trino, Snowflake, BigQuery, Dremio, AWS Athena.

Delta Lake

Created by Databricks (2019). Built on a **transaction log** (`_delta_log/`):

- JSON log files record every change: add file, remove file, schema change
- **Optimistic concurrency**: writers commit by appending to the log
- **MERGE / UPDATE / DELETE**: SQL DML on immutable Parquet files
- **Z-ordering**: sort data across multiple columns for better predicate pushdown

Tightly integrated with Spark and the Databricks platform.
Open-sourced, but the ecosystem is narrower than Iceberg's.

Table format comparison

Feature	Iceberg	Delta Lake
ACID transactions	Yes	Yes
Time travel	Yes	Yes
Schema evolution	Full	Full
Hidden partitioning	Yes	No
Partition evolution	Yes	No
Upsert optimization	Merge-on-read (v2)	Via MERGE

The industry is converging toward **Iceberg** as the default.

One sentence to remember

The file format is a design-time bet
on your query pattern.
Get it right and every query benefits for free.

Vocabulary recap

Term	Definition
Row-oriented	Fields of a record stored contiguously
Column-oriented	All values of a field stored contiguously
Self-describing	Format carries its own schema (JSON, Avro files)
Schema evolution	Readers and writers can use different schema versions safely
Predicate pushdown	Skipping data based on column statistics without reading it
Projection pruning	Reading only the columns needed by a query
Table format	Metadata layer (Iceberg, Delta, Hudi) that gives ACID and schema to file collections
Time travel	Querying a table as it existed at a past point in time via snapshot metadata

Lab

https://vbergeron.github.io/data-processing-at-scale/lab-2.1-file-formats.pdf

Vocabulary recap

Next

https://vbergeron.github.io/data-processing-at-scale/2.2-spark.pdf