

# What's the Big Deal with Big Data?

# 30

by Mairi Hallman and Patrick Boily, with contributions from Mian Ahsan Ali and Andrew Macfie

**Big data** is often described as “data that is too large for a single machine” or “data that do not fit on a laptop.” Those descriptions are not wrong *per se*, but they do sweep a lot of practical details under the rug: big data is the regime in which the **workflow** of ordinary data analysis begins to sputter and fail, not because the underlying ideas of modeling and inference have suddenly changed, but because previously negligible **constraints** cannot not be ignored anymore.\*

Perhaps surprisingly, in the big data regime, “more data” does not automatically provide “better evidence.” Large observational datasets can contain **measurement error**, **missingness**, **coding artifacts**, shifts in how variables are recorded, and subtle forms of **selection bias**.

With sufficiently large sample sizes, it can also become easy to obtain tiny *p*-values for effects that are practically irrelevant, while simultaneously missing the bigger story because the data science pipeline is **brittle** and/or the data does not actually measure what we think it does. Big data forces us to think simultaneously about **statistical validity** and **computational feasibility**.

What should the phrase “big data” mean to data scientists? Mostly, we think that it indicates that we should proceed with caution: failure modes in big data settings are usually conceptual, not algorithmic. The **5-V paradigm**, which describes big data *via* five aspects (see Section 30.2.7), is used not as a marketing checklist to distinguish **big data** from **small data**, but as a **practical diagnostic tool**: which of these pressures should be driving data pipeline and analytical choices, and what do they force us to give up?

Once we understand what makes a problem “big,” the question becomes: what needs to be done differently? For a while, the answer has been **distributed computing**, where a dataset is partitioned across machines (or across processes on the same machine) and computations are coordinated so that no single “worker” must store or process **everything**.<sup>1</sup> It also motivates a concrete way to think about **scaling**: when is the bottleneck due to computation, and when is it due **data movement**?

On the **hardware** side, we consider what can be gained at the device level (memory, storage, and parallelism), why cluster computing keeps reappearing as the default solution, and how cost and processing power shape realistic choices for analysis teams.

\* This includes **memory limits**, **disk and network I/O**, the **cost of moving data**, **heterogeneous data formats**, **continuous arrival**, **systems that evolve dynamically** over time, and so on.

|                                    |      |
|------------------------------------|------|
| 30.1 Danish Medical Data . . .     | 1956 |
| 30.2 Motivation . . . . .          | 1960 |
| Big Data in Practice . . . .       | 1960 |
| Big Data Across Domains            | 1960 |
| Value, Cost, Decisions . . .       | 1961 |
| Common Misconceptions .            | 1961 |
| Big Data vs. Small Data .          | 1962 |
| Data Sources . . . . .             | 1962 |
| 5-V Paradigm . . . . .             | 1963 |
| Failure Modes . . . . .            | 1964 |
| 30.3 Proceeding With Caution       | 1965 |
| Regime Change . . . . .            | 1965 |
| Ethics and Values . . . . .        | 1966 |
| Truly Large Numbers . . .          | 1967 |
| Practical Guidance . . . . .       | 1967 |
| 30.4 Distributed Computing .       | 1968 |
| One-Machine Breakdowns             | 1968 |
| Distributed vs. Parallel . .       | 1969 |
| Intuitive Analogies . . . .        | 1970 |
| When It Helps . . . . .            | 1972 |
| 30.5 Hardware Solutions . . . .    | 1973 |
| Device-Level Parallelism . .       | 1974 |
| Scaling . . . . .                  | 1974 |
| Cost vs. Performance . . .         | 1975 |
| Benchmarking . . . . .             | 1976 |
| 30.6 Software Solutions . . . .    | 1979 |
| R Tools . . . . .                  | 1979 |
| Python Tools . . . . .             | 1981 |
| Frameworks . . . . .               | 1982 |
| Selecting a Framework . .          | 1983 |
| 30.7 Practical Considerations .    | 1984 |
| MapReduce and Hadoop .             | 1984 |
| Apache Spark . . . . .             | 1986 |
| Parquet & Column Storage           | 1987 |
| Working With Clusters . .          | 1988 |
| AMS EMR Workflow . . .             | 1992 |
| 30.8 ML at Scale in Spark . . .    | 1997 |
| Naïve Bayes . . . . .              | 1997 |
| <i>k</i> -Means and Initialization | 2004 |
| Streaming <i>k</i> -Means . . . .  | 2009 |
| Regression/Regularization          | 2012 |
| Tree-Based Methods . . .           | 2016 |
| Class Imbalance . . . . .          | 2020 |
| 30.9 Exercises . . . . .           | 2024 |
| Chapter References . . . .         | 2027 |

1: A canonical reference point is MapReduce, which popularized a simple interface for distributed aggregation and transformation on large clusters [10].

On the **software** side, we focus on how the analyst's workflow changes: what it means to do big data work from R, how to select a framework that matches the problem and the team, and how to reason about the trade-offs between **convenience**, **transparency**, and **performance**.

We conclude with a practical discussion of **Spark** and an R-facing workflow, emphasizing the ideas that transfer across platforms and will remain relevant even as specific tools evolve or go out of fashion.

### 30.1 The Danish Medical Data Study

An alarm goes off. A person wakes up, scrolls through a social media feed, and gets ready for work. During a commute, they choose a playlist on Spotify while scrolling through the news.

During work, they consult Google and ChatGPT as needed and check out Reddit during a break. On the way home, they browse Amazon and buy new sneakers. To unwind at the end of the day, they click through a few options on Netflix before selecting a movie they have already seen four times.

In the background, this ordinary day generates an extraordinary amount of data. Queries, searches, clicks, and location history are stored somewhere, typically alongside metadata such as timestamps, device identifiers, and coarse location signals.

In many cases, those records are not isolated: they can be linked across platforms, aggregated over time, and analyzed at scale. Multiply this by millions of people and by years of activity, and the phrase **big data** starts to sound less like a buzzword and more like an on-the-nose descriptor.

But defining big data is often easier than explaining why it matters.<sup>2</sup>

Big data analysis is often associated with advertising, recommendation systems, and behavioural tracking. The same ideas, however, can also be used to study population-scale health outcomes, where the goal is not to optimize clicks, but to identify **risk patterns** and **disease progression** in ways that can support earlier intervention and better treatment.

In 2014, researchers published a paper outlining the results of their analysis of the *Danish National Patient Registry*. This massive dataset, spanning from January 1996 to November 2010, consisted of 65 million hospital records for 6.2 million patients [22]. Of these interactions:

- 16 million were inpatient events;
- 35 million were outpatient events, and
- 14 million were emergency events.

From these encounters, **101 million diagnoses** were recorded using **ICD-10 diagnostic codes**.<sup>3</sup>

2: The key take-away is not only that the datasets are large, but that they are **rich enough** to reveal patterns in behaviour, risk, and outcomes that are difficult to detect in small samples.

3: The *International Classification of Diseases*, 10th revision, is a standardized system of diagnostic codes used in medical records to encode diagnoses consistently across hospitals, clinicians, and time.

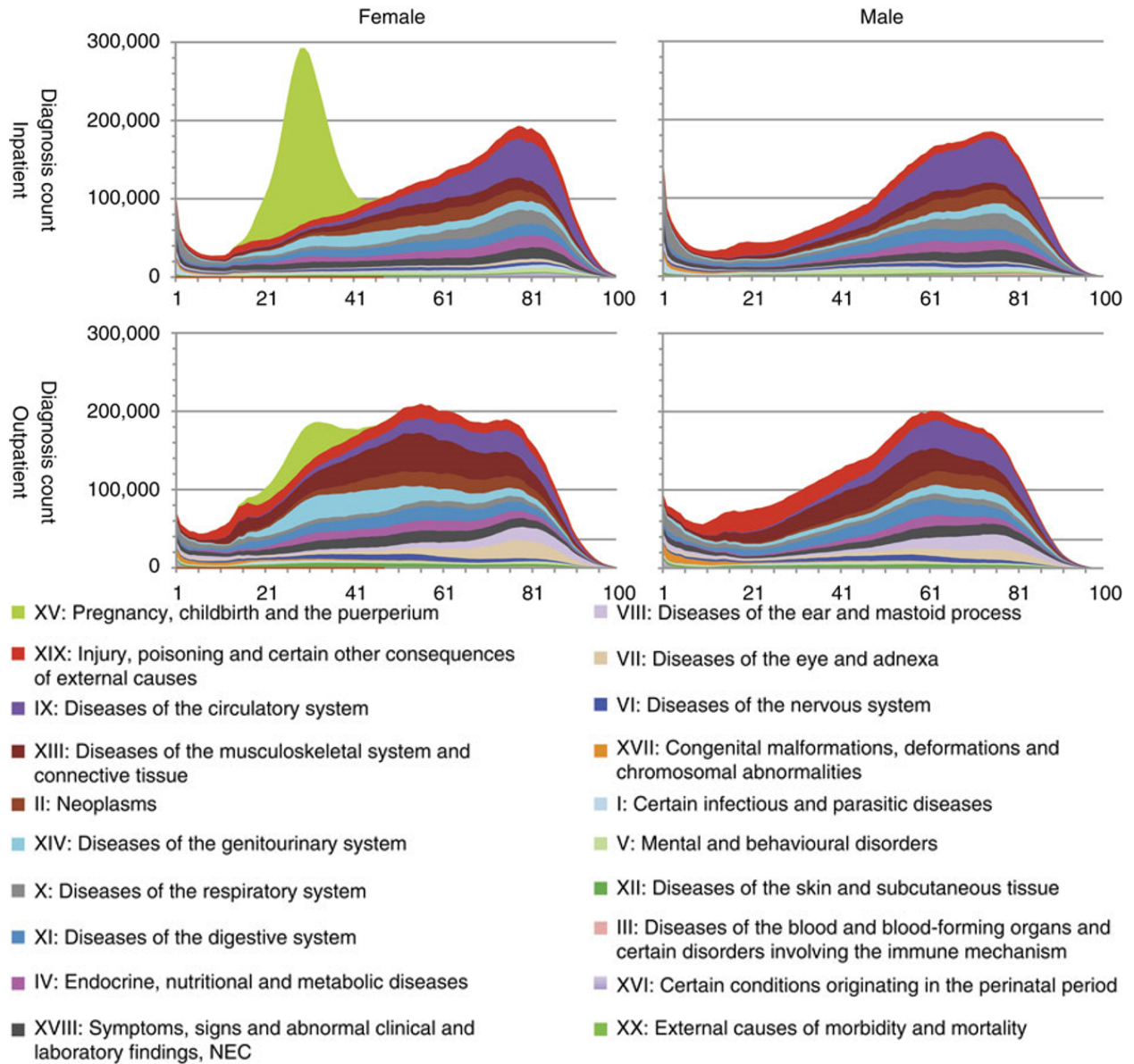


Figure 30.1: Diagnoses in the Danish National Patient Registry, with ICD diagnostic codes[22].

In addition to age and gender, the study found that diagnoses were very strongly correlated with interaction type. For example, compared to other diagnoses, injuries were much more likely to be emergency interactions.

Initial identification of potentially significant correlations was done by comparing patients with two given diagnoses to other patients of the same age and gender who had the same hospital interaction type. The initial **1 194 343 pairs of diagnoses occurring within 5 years of each other** found in the dataset were filtered to 62 821 of interest.

Of the filtered diagnoses pairs, 4014 trajectories were found to have **significant directionality**, meaning that one diagnosis was significantly more likely to occur before another. From here, researchers further identified 5784 trajectories of three diagnoses and **1171 trajectories of four diagnoses**.

The trajectories of four diagnoses were then grouped into 15 clusters. Two of the largest clusters corresponded to chronic obstructive pulmonary disease (COPD) and cerebrovascular disease (ACV). To make these results more tangible, Figure 30.2 shows two example trajectory clusters from the study, one centered on **COPD** and one centered on **ACV**.

In these diagrams, diagnoses play the role of nodes, and statistically supported directional relationships between diagnoses form links that trace common pathways through time, highlighting typical precursors and outcomes.

### Insights

The study found that not only was COPD correlated with cardiovascular diagnoses, but **“all trajectories starting with atherosclerosis are followed by a subsequent COPD diagnosis”**.

Another key finding was that gout was central to the cardiovascular cluster, supporting a hypothesis previously proposed by other researchers. The study also confirmed that COPD was chronically underdiagnosed in the Danish population.

### Why This Matters

Actionable insights from big data studies of this nature can have far-reaching positive impacts. Consider COPD as an example. Researchers noted that the condition was underdiagnosed, but also proposed that the disease trajectories they identified could help medical professionals identify risk factors, leading to earlier diagnosis.

Big data may require special handling, but it allows us to make connections that could never come from a bivariate relationship or a few data points. This is the sense in which “big data” can be more than a matter of scale: it can change which questions can be asked, and which answers can be supported by evidence.

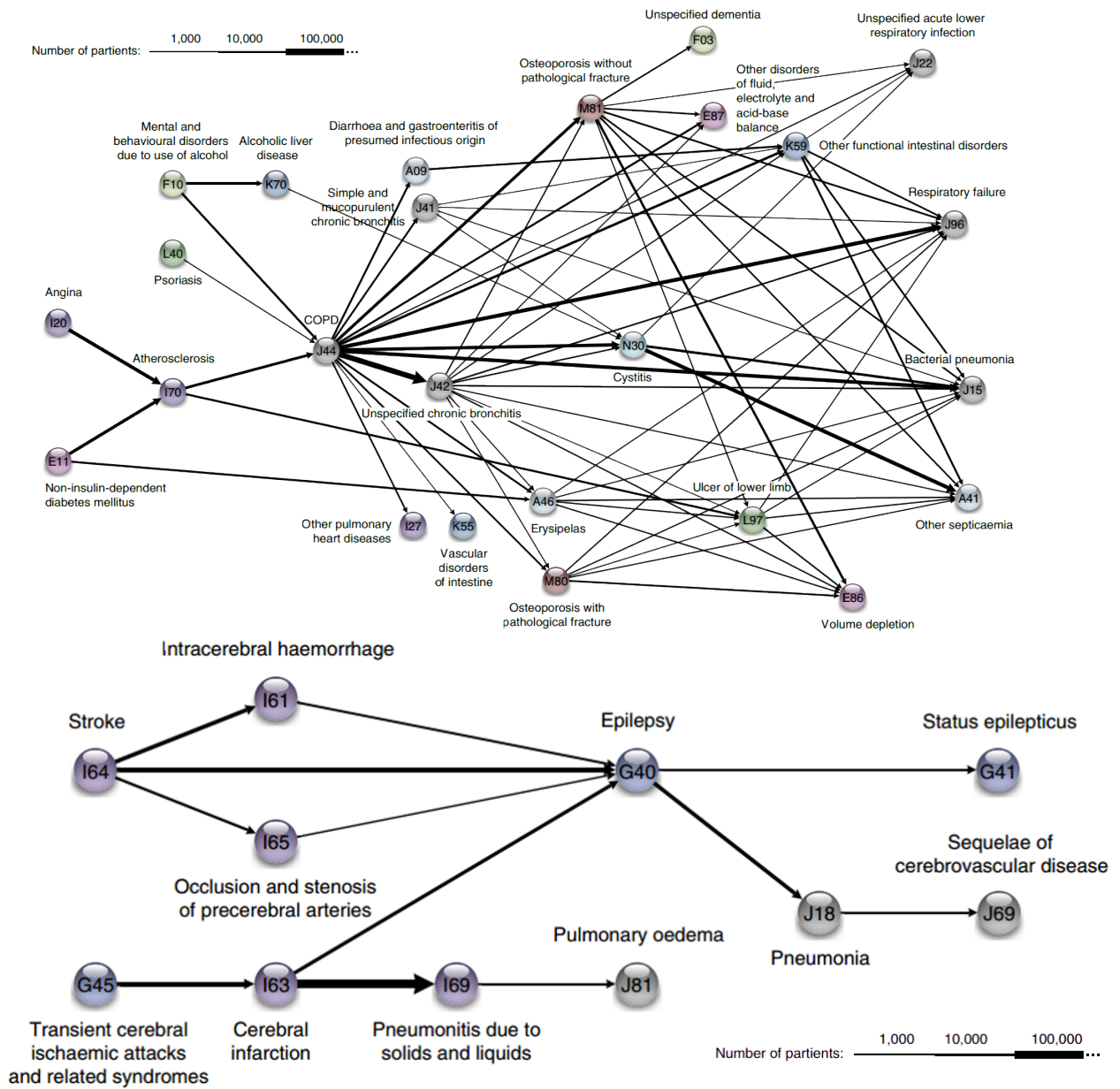


Figure 30.2: Two diagnosis trajectory clusters identified from the Danish National Patient Registry: COPD (top) and cerebrovascular disease (ACV, bottom) [22].

## 30.2 Motivation

4: If not a new type of science, period [33].

Due to the hype surrounding it, it can sometimes feel as though big data is a new type of data science.<sup>4</sup> In reality, its underlying goals are the familiar ones: describing what is happening, detecting anomalies, supporting decisions, building predictive or explanatory models, etc.

What does change, however, is that the practical constraints of computation become crucial. In the big data regime, issues like **memory limits**, **disk and network I/O**, **data movement**, and **coordination overhead** can dominate the timeline of a project. This is why big data is best understood as a shift in **workflow** rather than a shift in the meaning of inference.

### 30.2.1 What “Big Data” Means in Practice

A dataset is not “big” solely because it has many rows; it becomes “big” when ordinary analysis practices begin to break down. Concretely, **big data** is the regime in which one (or several) of the following is true:

- the raw dataset does not fit in RAM, so analysis requires **streaming**, **chunking**, or **distributed** storage;
- simple operations become dominated by **reading** and **writing**, rather than **arithmetic**;<sup>5</sup>
- intermediate results (joins, group-bys, feature tables, model checkpoints) are **too large** to manage comfortably on a single machine;
- the dataset is **heterogeneous** (multiple sources, formats, and update mechanisms), so that data engineering is as important as modeling;
- the data arrives **continuously**, and models must be updated or monitored over time.

5: For example, repeated passes over a large file.

This emphasizes that “big” is relative to a workflow and to a “budget”. A dataset that is big for a laptop might be small for an institution with dedicated servers, and a dataset that is big today might become manageable tomorrow as hardware improves.<sup>6</sup> It is not the **absolute scale** that matters, but whether that scale forces us to change how we **store**, **access**, and **transform** the data.

6: In practice, the most common bottleneck is not floating-point computation but **moving data**: reading from disk, shuffling across a network, serializing objects, and writing results back to storage.

The canonical response to these constraints has been **distributed computing**, where a dataset is **partitioned** across “workers” and computations are organized so that no single worker must store or process everything (we will discuss further in Section 30.4).<sup>7</sup>

7: MapReduce is a standard reference point because it made distributed aggregation and transformation conceptually simple, and because many later systems can be understood as refinements of the same idea [10]. We will revisit this important historical concept.

### 30.2.2 Why Big Data Matters Across Domains

Big data techniques appear across domains because the underlying structure of modern information systems is similar. Many organizations have:

- **digital traces of behaviour** (clicks, views, purchases, searches, location pings);
- **operational logs** (events, sensor readings, errors, transactions);
- **high-dimensional records** (text, images, genomic sequences, medical codes);
- **large-scale networks** (users, devices, suppliers, interactions), etc.

In retail and media, this may mean **recommender systems, forecasting, and inventory optimization**; in finance, **fraud detection, risk monitoring, and stress testing**; in transportation and logistics, **prediction, routing, and capacity planning**; in public health and medicine, population-scale **outcome tracking and disease progression analysis**, as in the Danish registry example of Section 30.1.<sup>8</sup>

Across all of these settings, the value of big data is often tied to **granularity** and **timeliness**: being able to detect small shifts early, segment populations sensibly, and monitor systems as they evolve. This is one reason why many big data projects are closer to **measurement and monitoring** than to one-off modeling exercises.

### 30.2.3 Value, Cost, and Decision-Making

Big data methods are adopted when they can change the balance between **value** and **cost**.

On the value side, large datasets can **reduce** uncertainty, support **finer segmentation**, and make **rare events** observable. On the cost side, they introduce **expenses** that are easy to underestimate: engineering time, infrastructure, storage, compute, opportunity cost of slow iteration, etc.

In many projects, the right question is therefore not “how do we fit/run this model?” but “**what is the cheapest workflow that produces good-enough evidence to support decisions?**”

A simple way to think about this is to separate two kinds of cost:

- **computational cost** refers to the time and resources required to run transformations and models;
- **organizational cost**, to the complexity introduced by distributed systems (configuration, monitoring, failure modes, debugging, access control, and reproducibility).

Distributed frameworks can reduce computational cost dramatically, but they also increase organizational cost; thus, we should adopt big data tools only when the computational savings **outweigh** the added complexity.<sup>9</sup>

### 30.2.4 Common Big Data Misconceptions

A frequent misconception is that big data tools should be used whenever data is “large” in an informal sense. In reality, many successful projects do not require Hadoop, Spark, or cluster computing at all.<sup>10</sup> Big data methods are often unnecessary when:

- the dataset **fits in memory** and the analysis is not computationally heavy;
- a **well-chosen sample** answers the question with essentially the same conclusions;
- the bottleneck is **data quality** (measurement error, missing values, shifting definitions), not scale;
- the task is **exploratory** and benefits most from fast iteration rather than maximum throughput.

8: The key point is not that big data is automatically better, but that it can support questions that require both **breadth** (many people or events) and **depth** (rich records over time).

9: This trade-off is also why many teams start with a small-scale prototype on a sample of the data, then scale out once the pipeline is stable and the analysis objective is clear.

10: Managers are sometimes eager to use such products simply because the infrastructure has already been purchased. When tool choice becomes a post-hoc justification for prior spending, it is easy to fall into a sunk cost fallacy and to scale a pipeline that the decision problem does not actually require.

11: This connects to the **law of truly large numbers** discussed in Section 30.3.3: with enough opportunities, surprising coincidences become inevitable [12].

There is also a statistical version of the same warning: large datasets can make it easy to detect tiny effects that are practically irrelevant, and they can also encourage overconfident pattern-finding.<sup>11</sup> In other words, **big data is not a guarantee of good evidence**; rather, it is an invitation to be more disciplined about what the data **measures**, how the **pipeline** is constructed, and which claims are actually **supported**.

A good rule of thumb is to treat big data tooling as a **means**, not an identity: we use it when it improves the quality and timeliness of decisions, and avoid it when it mainly adds complexity.

### 30.2.5 Big Data vs Small Data

In **small data** settings, it is often reasonable to assume that the dataset is **easy to load** into memory, relatively **stable** over the lifetime of a project, and **sufficiently curated** that the main difficulty lies in **analysis** rather than **data handling**.

In **big data** settings, those assumptions fail in routine ways. As we have mentioned, the main conceptual shift is not that inference has changed, but that the **pipeline** becomes an crucial object of study: how data is produced, stored, accessed, cleaned, joined, and transformed often determines what kinds of conclusions are **possible**, and how much those conclusions **cost** to obtain.

### 30.2.6 Data Sources and Modern Data Generation

A useful way to distinguish **big data** from **small data** is to look not only at how much data we have, but at **how** it is produced. In classical settings, data is usually collected **on purpose**: a study is designed, variables are defined in advance, and a dataset is assembled to answer a specific question (see [3, ch. 10, 14, 16]).

In contrast, many modern large datasets are **system by-products**: they are produced because a platform needs to run, a service needs to be monitored, or a pipeline logs what it is doing. The resulting data is often informative, but it is typically **untidy**, **incomplete**, or **unstable**.

Modern data generation is often **event-based**. Rather than recording one row per individual in a fixed table, systems emit streams of **events** with timestamps, identifiers, and context (see [3, ch. 28] for more information). These streams are typically **high-frequency** and **granular**, which is partly why they are valuable, but that also makes them difficult to manage: a single user interaction might generate multiple logs across multiple services, each with slightly different **semantics** and **latency**.<sup>12</sup>

This is also why, in big data work, **data engineering** (see [3, ch. 17]) is often inseparable from analysis: the analyst rarely receives “the dataset.” Instead, there are multiple raw sources that must be **linked**, **filtered**, and **reconciled**. The key practical ingredients are **identifiers** (so that events can be joined), **time** (so that sequences can be reconstructed), and **provenance** (so that we can track how a variable was created and whether its meaning has drifted).

12: That is, the precise meaning of a log field or event (what exactly is being measured, under what definition, and with which edge cases) and the delay between an interaction and when its corresponding log entry becomes available for analysis, which can vary across services due to batching, queuing, retries, and network or processing overhead.

When identifiers are missing or inconsistent, the problem can change from modeling to **record linkage**, **de-duplication**,<sup>13</sup> and careful definition of what constitutes an “entity” in the first place.

A further difference from many small-data settings is **heterogeneity**. Real systems produce a mixture of **structured** records (transactions, inventories), **semi-structured** records (JSON logs, nested fields), and **unstructured** data (text, images, audio).<sup>14</sup> Even when everything is “structured,” definitions can shift over time through software updates, policy changes, or new instrumentation. This kind of **schema drift** is not an anomaly: it is an **expected feature** of long-running systems.

The point of this subsection is not to re-list applications across domains (as in Section 30.2.2).<sup>15</sup> Rather, it is to highlight a workflow implication: when data is produced as a stream of operational traces, the primary challenges are often **ingestion**, **storage layout**, **joining and aggregation**, and **quality control** before we ever reach the modeling stage.

This is precisely where big data tools becomes relevant: the statistical goal is not new, but the data arrives in forms and volumes that force different choices about how we store, transform, and compute.

### 30.2.7 The 5-V Paradigm

A useful way to articulate the practical difference between big data and small data is the **5-V paradigm**.<sup>16</sup> The objective is not to treat the Vs as a checklist, but to identify **which pressures** are actually driving engineering and analysis decisions.

**Volume:** in big data settings, **storage** and **memory limits** cease to be background details; even when raw data fits on disk, **intermediate objects** (joins, feature tables, model checkpoints, etc.) can quickly exceed what may comfortably be managed on a single machine.

**Velocity:** data may be generated **continuously**, sometimes at a rate that forces a trade-off between **latency** and **sophistication**,<sup>17</sup> in such cases, it is often more important to have a **stable** pipeline that can be monitored than it is to have the most complex model.

**Variety:** data may come in **many formats** and from **many sources**,<sup>18</sup> variety increases not only storage complexity, but also the risk that variables with **similar names** encode **different meanings** across systems.

**Veracity:** as scale increases, it becomes easier for **data quality problems** to hide in plain sight: **missing value patterns** change over time, definitions shift, and small encoding quirks upstream<sup>19</sup> can get copied into every derived table, biasing aggregates and features until the artifact looks like a **real pattern** rather than a **data issue**.

**Value:** big data tools are justified when their use can change what **decisions** can be supported, or the cost and timeliness of making those decisions: a practical litmus test is whether the additional complexity buys **better evidence** or merely **higher throughput**.<sup>20</sup>

Some authors also talk of the **7-V paradigm** [30], by adding two additional Vs (these may change from authors to authors).

13: The elimination of duplicate observations.

14: See [3, ch. 14].

15: There we emphasized *where* big data is used; here we emphasize **how** it is generated, because generation mechanisms determine storage, access patterns, and failure modes.

16: Originally, the 3-V paradigm [24].

17: The latter term is linked to how elaborate the processing pipeline is (richer feature engineering, heavier data validation/cleaning, and/or more complex models). Higher sophistication can improve accuracy or reliability, but it usually increases compute and coordination time, which can force a trade-off with low **latency** when data arrives quickly.

18: This includes structured tables, semi-structured event logs, and unstructured modalities like text and images.

19: Placeholders like 999, default categories, unit mix-ups, join duplication, etc.

20: That is, the analysis improves decision quality (for example, by reducing uncertainty, reducing bias, or measuring the right quantity), as opposed to only providing more computations per unit time. Throughput is useful, but it is not a justification on its own if it only accelerates an unreliable or irrelevant pipeline.

21: This can occur because user behaviour evolves, sensors drift, products change, logging conventions are updated, or all (or some) of the above.

22: For example, in **clickstream analysis** (page views, carts, purchases), a slow join or group-by could lead data scientists to:

- switch to faster inner join,
- de-duplicate records aggressively,
- cache intermediate tables,
- use sample/approximate counts, etc.

These changes can make the job **faster**, but they also change the data's ability to **represent the system**:

- an inner join silently drops users who browsed but did not purchase (biasing the dataset toward purchasers),
- de-duplication can collapse legitimate repeated events,
- cached intermediates can mix pipeline versions across days, and
- sampling can erase rare but important patterns, say.

The result is a pipeline that **runs faster** while answering a **subtly different** question:

- at **best**, the **mismatch** is eventually detected **downstream** (for example, when metrics become inconsistent or a production model fails on the "missing" cases) and we need to go **back to square one**, wasting time and other resources;
- at **worst**, the mismatch goes **unnoticed** and decisions are made on the basis of **erroneous data**, with potentially **costly consequences**.

**Variability:** the data-generating process may **change over time**,<sup>21</sup> in practice, variability is a reminder to treat models and pipelines as objects that require **monitoring**.

**Visualization:** human interpretation does not disappear in big data settings, nor does our need for storytelling (see [4]); even when the dataset is too large to plot directly, analysts still rely on **summaries**, **subsampling**, and **diagnostic graphics** to validate assumptions, detect anomalies, communicate results, and so on.

### 30.2.8 Typical Failure Modes

When big data is treated like small data, **failure** often first shows up as a **performance** problem, only later becoming a **correctness** problem.<sup>22</sup>

**Memory-first thinking:** a common early failure mode is assuming that data can be loaded and manipulated as a single in-memory object; when this assumption fails, workarounds like **swapping** (using disk storage as temporary "extra memory" when the RAM is full) or repeated **copying** can make even simple tasks unusably slow.

**I/O-dominated pipelines:** in big data regimes, the dominant cost is often **reading**, **writing**, and **moving data**, not arithmetic; workflows that repeatedly scan large files, save many intermediate CSVs, or shuffle large tables across a network can spend most of their time **waiting**.

**Naïve joins and group-bys:** operations that are routine on small data can become **expensive** and **fragile** when scaled up; joins can trigger large shuffles, and group-bys can become **bottlenecked** by skew, where a small number of keys account for a large fraction of the rows.

**Schema drift and silent meaning changes:** when data sources evolve, variables may keep the same name but **change meaning**; this is often noticed quickly in small data projects, but it can silently **contaminate** downstream tables for months in big data projects without **explicit monitoring**.

**Statistical overconfidence:** big data analysis can produce **tiny  $p$ -values** (see [3, ch. 7]) for effects that are **practically irrelevant**, and they can also make **spurious patterns** easy to discover and **difficult to falsify**; this is one reason why we will later revisit the **law of truly large numbers** [12].

**Premature scaling:** another common failure mode is **scaling up** before the question is **clear** and the pipeline is **stable**; when a workflow is still evolving, distributed systems can amplify **debugging costs** and make it harder to identify what is going/went wrong.

At the risk of repeating ourselves (we think it's really important!), the main takeaway is that "big data" is not a label that attaches to a dataset once and for all; it is a **description of a regime** in which **computational constraints**, **data engineering**, and **statistics** interact tightly, and in which the **simplest working pipeline** is often the **best starting point**.

## 30.3 Proceeding With Caution

In the previous section, we focused on what makes a problem “big” in practice, and on the kinds of workflow failures that appear when scale and complexity are underestimated.

In this section, we shift from tooling and pipelines to a different kind of warning: even when a big data pipeline is fast and stable, the **inferences** and **decisions** it supports can still be **fragile**.

The central theme is that big data tends to increase the **stakes**. It becomes easier to:

- **overfit**,
- **confuse correlation with causation**,
- mistakenly optimize a **proxy** as if it were the **true target**, and
- **cause harm** at scale when a system is optimized in the wrong direction.

Proceeding with caution means treating big data work as an **interaction** between:

1. a data-generating system that can change,
2. a decision process that embeds values, and
3. statistical reasoning that must remain honest about uncertainty.

### 30.3.1 Regime Change and the Limits of Historical Data

A persistent temptation is to treat the past as a faithful rehearsal for the future. In big data settings,<sup>23</sup> this approach can fail for a simple reason: the data is often generated by a **living system** (a platform, a hospital network, a supply chain, a policy environment, etc.), and living systems are rarely constant.

A **regime change** occurs when the rules of the game shift enough that **historical patterns no longer summarize current behaviour**, either in the form of a policy change, a software update, a new measurement instrument, a product redesign, a change in user incentives, an external shock, etc.<sup>24</sup> Even if the data remains well-structured through the change, it might still become partially **out of date** for the task at hand.<sup>25</sup>

This shows up in practice in several common ways.

**Covariate shift:** the input distribution changes (who shows up, what is measured, what behaviours are represented, etc.) even if the underlying relationship between inputs and outcomes remains similar.

**Label shift:** the prevalence of outcome classes changes (for example, the base rate of fraud), which can break probability calibration even when ranking quality seems stable.

**Concept drift:** the meaning of the target itself changes, often because the system adapts to the model or because the operational definition of an outcome is revised, etc.

23: In the general data science setting too, if we’re honest.

24: The COVID-19 pandemic and resultant lockdown and upheaval might be the standard bearer for data regime changes for years to come.

25: In statistical language, we lose some form of **stationarity** (see [3, ch. 9]): the joint distribution of variables and outcomes can drift over time, so a model trained on “yesterday” data can be mis-calibrated “today”.

In a sense, the specific regime change mechanisms are less important than the workflow lesson: big data analysis is often an analysis of a **moving target**.

This is also where big data connects back to the earlier discussion of **veracity** and **variability**. A pipeline can be perfectly reproducible while faithfully reproducing an outdated, mismeasured, or drifting quantity. In that sense, the limit of historical data is not primarily a limit on computation power, but an **epistemic** one: it constrains what we can **claim about the present**, and especially **about the future**.

### 30.3.2 Ethics and Values: Data-Informed vs. Data-Dictated Decisions

**Decision-making** has been influenced by the advent of data science and big data analytics, and not necessarily all for the positive. Ideally, stakeholders would make **data-informed decisions** (using data evidence to support judgment) over **data-dictated decisions** (turning data into an authority that replaces judgment, often by turning a complex human or institutional question into a single score, threshold, or ranking). But this is not what always happens.

The risk is not merely philosophical: when decisions are **dictated** by a model, several negative outcomes can result (more negative consequences are discussed in [3, ch. 13, 14]).

**Proxy substitution:** the system optimizes what is **easy to measure** rather than **what matters**.

**Automation bias:** humans defer to a model even when the model is operating **outside its intended regime**.<sup>26</sup>

**Distributional harm:** a model that is accurate **on average** can still be systematically harmful for **subpopulations**, especially when measurement quality differs across groups.

**Surveillance creep:** a dataset assembled for one operational purpose can be repurposed to monitor behaviour, enforce policies, or infer sensitive attributes, etc.

Proceeding with caution requires acknowledging that “**values**” are attached to big data projects even before a model is fitted.<sup>27</sup> The **technical** pipeline and the **ethical** stance are not separable, because the pipeline determines what becomes **legible** and **actionable**.<sup>28</sup>

A practical way to keep this grounded is to ask, **early and explicitly**:

- What decision(s) will this analysis influence, and who bears the cost of errors?
- What is being measured/targeted, and what is being used as a proxy?
- What would be a reasonable argument *against* deploying the resulting model or metric?

These are not statistical questions, but tackling/answering them is essential to **statistical responsibility** at scale.

26: This often interacts with Section 30.3.1: if a regime changes, the model may **silently** become less reliable precisely when the environment becomes harder to reason about informally.

27: Which outcomes count as “**success**,” which **errors** are acceptable, which **trade-offs** are permitted, which **populations** are protected by design, etc.

28: In this context, “legible” means “measurable in the data, in a form the system can reliably see and optimize.” Logging, encoding, and aggregation choices make some outcomes visible as **metrics** (and therefore actionable), while other important effects remain **hidden** or only **weakly proxied**.

### 30.3.3 The Law of Truly Large Numbers

As datasets grow, it becomes easier to find patterns. It also becomes easier to find **spurious correlations**, that is to say, patterns that are not real or data relationships that do not actually carry inferential or descriptive meaning. This is the intuition behind the **law of truly large numbers**: with enough opportunities for coincidence, surprising events become inevitable [12].

A simple version of the idea comes from **multiple testing** (see [3, ch. 8]). Suppose we test  $m$  unrelated hypotheses, each at significance level  $\alpha$ . Even if **none** of the hypotheses reflect a real effect, the probability of seeing **at least one** “significant” result is

$$1 - (1 - \alpha)^m,$$

which can become quite large when  $m$  is itself large; for instance, when we use the classical value  $\alpha = 0.05$  and test  $m = 10$  independent hypotheses about the data, the probability of one of those showing a spurious correlation is already at  $\approx 40.1\%$ !<sup>29</sup>

Big data settings naturally invite large  $m$ : in a dataset with 1,000 numerical features, for instance, we could compute up to  $m = 500,500$  correlations. The **statistical failure mode** is not subtle: we mistake a coincidence for a mechanism, then optimize around it; the **organizational failure mode** (*p-hacking*) is equally common: we only build false confidence when we repeatedly check the data until something interesting appears.

The core caution is therefore not “do not look for patterns,” but “**treat found patterns as potential false positives until they are validated.**”

### 30.3.4 Validation, Sanity Checks, and Uncertainty

But in big data work, validation is a **state of mind**, not just a single step near the end of a project: we assume that things can (and will!) go wrong, and we deliberately design checks that will surface problems **early** and **cheaply**.<sup>30</sup>

A few practical strategies are especially useful because they reduce the chance we will fool ourselves while also improving “**debuggability.**”

**Start with invariants:** before sophisticated modeling, verify facts that must be true by definition (counts that reconcile, units that match, ranges that make sense, join keys that do not explode row counts).

**Track meaning, not only code:** a pipeline can be reproducible while a variable silently changes meaning; maintain a minimal data dictionary and record when definitions or logging conventions change.<sup>31</sup>

**Validate on time, not only on rows:** if the system evolves, a random training split could be misleading; whenever possible, test whether conclusions remain stable across time windows.

29: This should not be read as a knock against hypothesis testing itself. It is merely a reminder that, at scale, the number of chances we have to fool ourselves can be enormous, especially when we search across many variables, many segments, many time windows, and many model variations.

30: The motivation is practical:

- big data pipelines are **expensive** to run **end-to-end**; if a join key drifted, a unit changed, or a logging field went missing, we want to catch it before burning hours (and dollars!) on a full cluster job;
- many **failures compound**, so that small upstream issue can quietly **propagate** into downstream tables, features, and dashboards; early failure prevents “bad data” from becoming **institutionalized**, and
- fast checks also enable **rapid iteration**; if every run takes hours, users stop validating, which is a breeding ground for subtle bugs to survive.

31: This is one reason why “schema drift” and “semantic drift” are often more dangerous than missing values: missingness is visible; semantic changes may not be.

32: A **narrow** confidence interval (see [3, ch. 7, 8]) is only meaningful when the statistical model matches the data-generating process well-enough that the uncertainty being quantified is the uncertainty that matters.

33: An “I/O job” is a computation whose runtime is dominated by **input/output** rather than arithmetic: most of the time is spent reading from disk/object storage, writing results, shuffling data over the network, and serializing/de-serializing records. The **bottleneck** (to be discussed shortly) is moving bytes rather than doing math.

34: In practice, what looks like a “modeling” job often becomes an “I/O job”: the machine spends most of its time waiting for data to be read, written, or copied.

35: That is, the **slowest constrained step** in a pipeline (a machine, worker, network link, disk, or stage) that **limits** the end-to-end throughput.

**Quantify sensitivity:** re-run key conclusions under small, defensible perturbations: alternative joins, alternative handling of missing values, alternative segment definitions, alternative reasonable pre-processing choices, etc.

We end with a caution about **uncertainty**. In large samples, classical standard errors can become tiny even when the main uncertainty is not sampling error but **measurement error, dependence, selection, or nonstationarity** (see [3, ch. 9, 10]). Big data can therefore produce **extremely precise estimates of the wrong quantity**.<sup>32</sup>

The overall takeaway is consistent with the chapter’s theme so far: (with apologies to J. Irving,) in the world according to big data, sometimes **more is less**; the promise of big data may be real, but the conceptual failure modes often get in the way. Proceeding with caution means building pipelines and analyses that remain **honest under drift** and about what the data (big or small) **can and cannot justify**.

## 30.4 Distributed Computing Concepts

A recurring theme of this chapter is that **big data** is about entering a regime where **constraints** that were once negligible (memory, I/O,<sup>33</sup> coordination, failures) begin to have an impact on determining what is feasible, analysis-wise.

**Distributed computing** is the broad family of ideas and systems we use to **respond** to those constraints: we split the data across multiple **workers** and **coordinate them** so that no single machine must store or process **everything**.

### 30.4.1 When Single-Machine Analysis Breaks Down

In a **single-machine** workflow, we think of analysis as a **sequence of transformations** applied to an **in-memory object**. But once datasets and intermediate objects become too large, this view breaks down in several routine ways.

First, **memory limits** are absolute. When a dataset (or a join, or a feature table) does not fit in RAM, the system must repeatedly move chunks between RAM and disk. This **swapping** can turn a task that should take seconds into one that takes hours (or days, or ...) because disk access is **orders of magnitude slower** than memory access.<sup>34</sup>

Second, even when the data fits on disk, the pipeline can become **I/O-dominated**. Many common operations involve **repeated passes** over **large files or tables** (filters, group-bys, sorts, merges, etc.). The arithmetic may be simple, but reading and writing becomes a **bottleneck**.<sup>35</sup>

Third, big data pipelines often demand **throughput** and **reliability** at the same time. When a workflow runs for hours, failures cease to be rare edge cases: a single machine can **crash** (often!), a process can **run out of memory**, a disk can **fill unexpectedly**, and so on. Distributed systems are typically designed so that a task can be **re-tried on another**

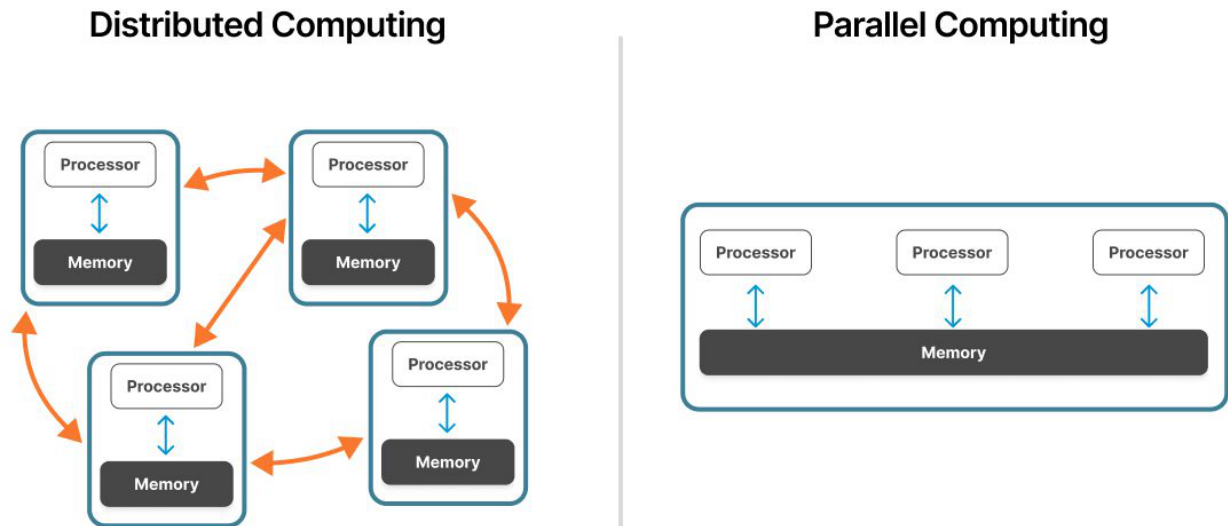


Figure 30.3: The conceptual differences between parallel and distributed computing [34].

**worker** if necessary, making failure a **manageable event** rather than a project-ending disaster.<sup>36</sup>

Finally, even if the analysis fits on one machine in principle, **time-to-iteration** can become unacceptable. Many projects fail not because the final computation is impossible, but because it takes too long to **iterate**, **validate**, and **debug** the pipeline.

### 30.4.2 Distributed vs. Parallel Computing

The words **parallel** and **distributed** are sometimes used interchangeably, but it is helpful to separate them.

**Parallel computing** refers to multiple computations taking place **simultaneously**; this can happen on one machine (multiple cores, multiple threads, a GPU) or across multiple machines. The key idea is **concurrency**, which is to say that independent pieces of work proceed **at the same time**.

**Distributed computing** is a particular kind of parallelism in which work is split across multiple machines (or multiple processes) that have **separate memory**. Because the workers do not share RAM, they must communicate over a network, which introduces **overhead** and **failure modes** that are not present in shared-memory parallelism.

In short: **parallel** means “many things at once,” while **distributed** means “many things at once, across separate machines, with explicit coordination and communication” (see Figure 30.3).

In practice, analysts also use **grid computing**, where a large collection of machines contributes compute capacity to a common objective, sometimes across administrative boundaries.<sup>37</sup>

Grid computing is distinct from the tightly coordinated **cluster computing** used by systems like Hadoop or Spark, but it highlights the same core idea: **distribute work to match scale**.

36: This is one reason that “distributed” is not only about speed. It is also about making large jobs robust to routine failures.

37: A well-known scientific example is large-scale distributed compute for experiments at CERN, where the *World LHC Computing Grid* shares analysis workloads across participating sites [6, 5]; *SETI@home*, a volunteer computing project, was a popular public example [43].

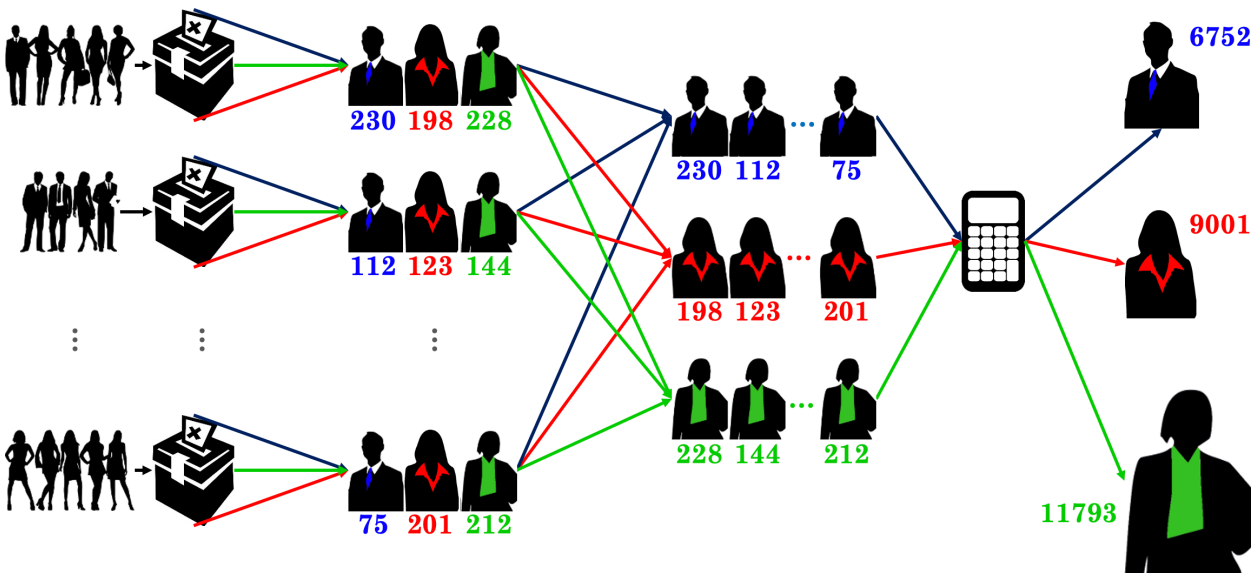


Figure 30.4: A simple analogy for parallelism: counting ballots by splitting work across many local counters and then aggregating results.

### 30.4.3 Intuitive Analogies

Analogies are imperfect, but they help emphasize what changes when we distribute work: we may gain **capacity** and **speed**, but we must also think about **coordination**, **communication**, and **bottlenecks**.

#### Election Counting

Election counting provides a clean picture of **divide-and-aggregate**. Each **ballot box** can be counted independently, producing a **local summary**. Those summaries are then combined to produce **riding-level totals**, then **province-wide totals**, and so on (see Figure 30.4).<sup>38</sup>

This conceptual pattern is found behind many distributed data operations: a **map** step applies a local computation to each partition, and a **reduce** step aggregates the partial results.<sup>39</sup> The appeal is that local work is easy to scale, and aggregation is often much smaller than the raw data.

#### Pizzeria Bottlenecks

A busy pizzeria illustrates a different lesson: **parallelism** does not automatically **eliminate waiting**. We can add more preparation cooks to take orders, prep toppings, and assemble pizzas, but if there is only one oven, it becomes a **bottleneck**; the entire system's throughput is capped by the slowest essential component.<sup>40</sup>

Distributed data pipelines have the same structure. Some steps scale almost **linearly** with more workers (independent parsing, simple filtering, embarrassingly parallel feature extraction); other steps create **global contention** (large shuffles, global sorts, wide joins). When the bottleneck is network traffic or a skewed key, "adding workers" can resemble hiring more staff in a kitchen with a single oven: **more movement**, but not necessarily **more throughput**.<sup>41</sup>

38: In the Canadian context (and other first-past-the-post jurisdictions), candidates are elected at the riding-level and totals at higher hierarchical level have less importance; in proportional representation systems (and their variants), national-level totals are more important than local-level totals.

39: MapReduce popularized this mental model for large-scale processing [10]. In the ballot analogy, "map" is counting each box and "reduce" is summing the counts. We will have more to say on the topic in Section 30.7.1.

40: The oven need not be the culprit, as can be seen in Figure 30.5.

41: This is one reason distributed systems force us to reason about **where time is spent**: CPU, disk, network, and coordination overhead can each become the effective "oven."

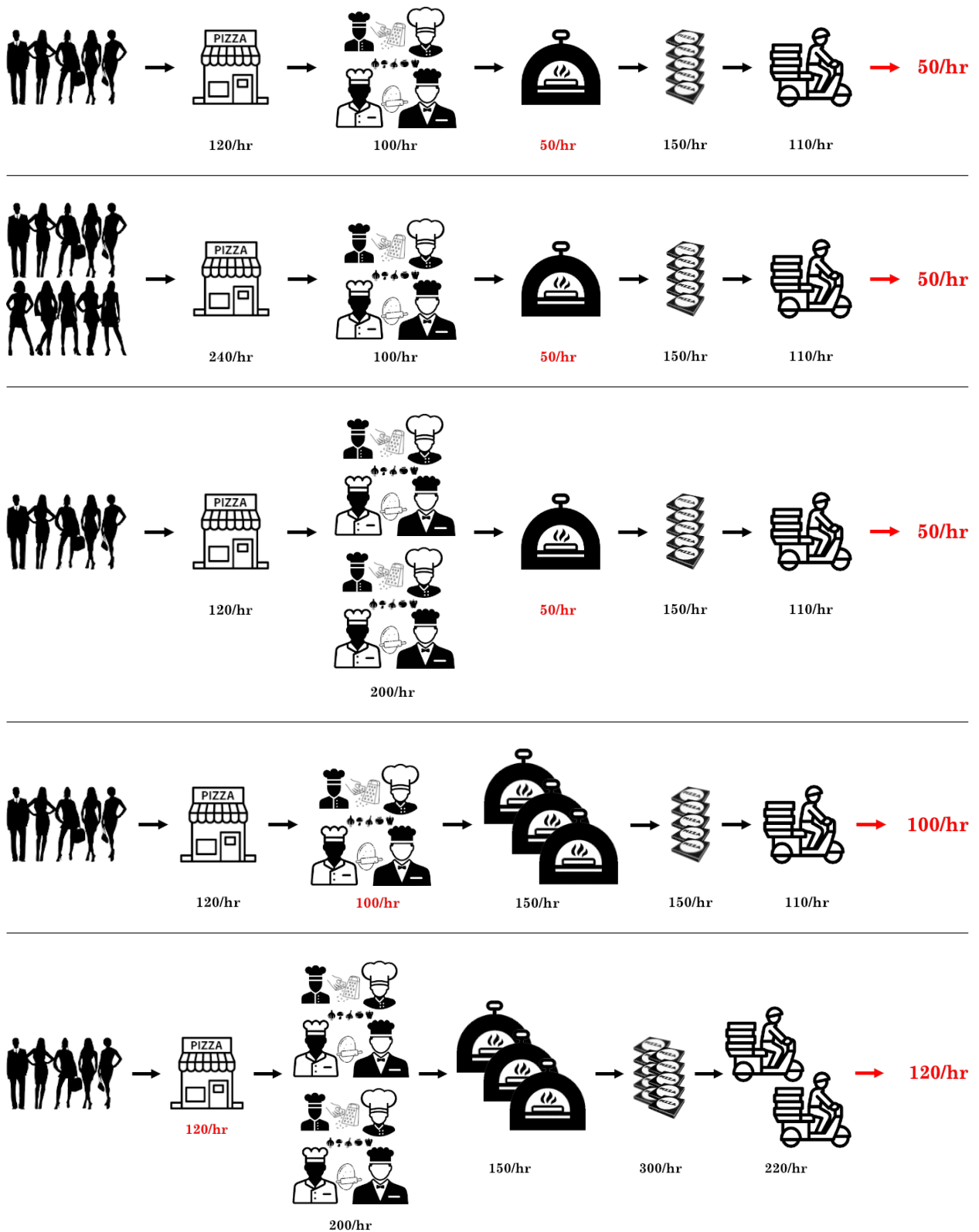


Figure 30.5: A simple analogy for bottlenecks: adding staff does not help the throughput/wait time if the oven is the limiting resource; adding ovens does not reduce an individual pizza’s cooking time, but it might transfer the pipeline bottleneck to another resource.

### 30.4.4 When Parallelism Helps (and When it Does Not)

Parallelism helps **most** when work can be split into pieces that are **nearly independent** and when the **cost of coordination** is small relative to the **cost of computation**.

A useful heuristic is to separate the pipeline into three kinds of work.

**Local work:** operations that can be performed **within each partition** without reference to **other partitions** (many filters, per-user features, per-file parsing, etc.); these often scale well.

**Aggregation:** operations that **combine summaries** (counts, sums, averages, histograms, etc.); these can scale well when **intermediate summaries** are small objects.

**Global coordination:** operations that require moving data **across partitions** to align keys (joins, group-bys with high-cardinality keys, global sorts, etc.). These are often where distributed pipelines become **slow** or **fragile**.

#### Computing Near the Data

In many big data settings, the primary goal is not in general to “do more arithmetic per second,” but to **move fewer bytes**. This motivates the idea of **computing near the data**: we perform the data-intensive parts of an analysis *where the data already lives* (on the workers, close to storage), and communicate only **reduced summaries** back to a coordinating process [7].<sup>42</sup>

Many statistical routines naturally decompose into a two-phase pattern:

1. a simple pass over the full dataset to compute **summaries** (sufficient statistics, objective values, gradients, counts, etc.), and
2. a more complex step carried out on a **small object** (solving a linear system, inverting an  $m \times m$  matrix, constructing an update rule, choosing a step size, etc.).

For example, in **linear regression** with design matrix  $\mathbf{X}$  and response  $\mathbf{Y}$ , the full dataset can be summarized by  $\mathbf{X}^T \mathbf{X}$  and  $\mathbf{X}^T \mathbf{Y}$  (see [3, ch. 8]). Each worker  $w$  computes its **local contributions**  $\mathbf{X}_w^T \mathbf{X}_w$  and  $\mathbf{X}_w^T \mathbf{Y}_w$  and an **aggregation step** sums them:

$$\mathbf{X}^T \mathbf{X} = \sum_w \mathbf{X}_w^T \mathbf{X}_w, \quad \mathbf{X}^T \mathbf{Y} = \sum_w \mathbf{X}_w^T \mathbf{Y}_w,$$

after which the coordinator can compute the **least-square coefficient vector**  $\hat{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{Y}$  using only these small summaries.

**Iterative methods** (which are often required to fit machine learning or deep learning models, see [3, vol. 3 or ch. 31]) follow the same logic: a typical update has the schematic form

$$\beta^{(t+1)} = \beta^{(t)} + U(\mathbf{X}, \beta^{(t)}),$$

where the expensive part is evaluating the **full-data quantity** needed to build  $U$  (often a **gradient** or **related summary**), while the “global” step manipulates only the **resulting reduced object**.

42: This is a **data locality** viewpoint: if the bottleneck is I/O or network traffic, the main “speed-up” often comes from avoiding large transfers rather than from adding compute. The (fundamental) concept of computing near the data is also referred to as query shipping vs. data shipping, pushdown, move compute to data, near data processing, and data gravity.

This is also why many distributed learning workflows repeatedly **broadcast** the current parameter vector to workers and **reduce** local gradient contributions back to the driver.

Seen this way, the election-counting analogy is not just about **splitting a task across workers**. It is about an **intentional communication pattern**: local passes over large data to produce **small summaries**, followed by **centralized logic** on those summaries. When such a pattern is available, parallelism is often effective; when it is not, we are pushed toward **expensive shuffles, global coordination, and/or semantic compromises**.

### The Limits of Parallelism

Parallelism helps **least** when there is a hard **sequential component** or a single constrained resource: this is the pizzeria lesson. It also appears when the “resource” is not hardware but semantics: a step may require a **globally consistent definition**, a **de-duplication rule**, or an **ordering constraint** that cannot be safely parallelized without **changing the question** being answered.<sup>43</sup>

Finally, we note that distributed computing changes the **debugging mindset**: on a laptop, failures are often **obvious** and **immediate**; on a cluster, many problems show up as **performance degradation, intermittent failures, or subtle inconsistencies** across runs, which could go undetected without special precautions. This is why distributed systems place so much emphasis on **monitoring, logging, and reproducibility**. The goal is not only to run faster, but to remain **interpretable** when something does goes wrong.

We will return to these themes when we discuss concrete **frameworks** in Section 30.6: why MapReduce became influential [10], and how later systems (such as Spark) were designed to make **iterative analysis** and **interactive workflows** less painful (or at least, require less manual input from analysts).

## 30.5 Hardware Solutions

One type of response to the big data **constraints** is algorithmic and organizational (distributed frameworks, partitions, workers, and shuffles, as in Section 30.4). The type of response we discuss in this section is **hardware**; we summarize the main hardware configurations that support parallelism at scale.

Our objective is not to turn the reader into a **systems engineer**, but to provide enough high-level concepts and vocabulary to understand **what changes** when we move from a laptop to a workstation, a **cluster**, or a **cloud environment**.

43: A concrete example is **sessionization** in **clickstream** data. Suppose the real question is: “How many user sessions occurred last week?” where a **session** is defined as a maximal sequence of events from the same user with no gap larger than 30 minutes. A tempting parallel strategy is to partition the raw logs by day, say, process each partition on a different worker, build sessions locally, and then add up the per-partition session counts. This runs fast, but it quietly changes the definition: sessions can cross a partition boundary. For example, if a user is active from 23:50 to 00:10, the global 30-minute rule says this is **one** session, but the partition-local pipeline will count **two** sessions because it forces an artificial break at midnight. Parallelism has not merely accelerated the computation; it has replaced the original question with a different one: “How many sessions are there under a modified rule that breaks sessions at partition boundaries?” The same issue appears with **de-duplication**: if duplicates can occur across partitions (retries, multi-service logging, delayed events), then “deduplicate within each partition and sum” answers “unique events per partition,” not “unique events globally.” In both cases, the bottleneck is semantic rather than computational: the step requires a **globally consistent definition** and a **global ordering** that cannot be enforced by partition-local work without either extra coordination or an explicit change in what is being counted.

44: The smallest sequence of programmed instructions that can be managed and executed independently.

45: This expression refers to a program, algorithm, or system design that makes extensive use of **conditional statements** (if/else, switch, for and while loops, or similar control flow structures, see [3, ch. 1]) that cause the execution path to diverge frequently.

46: This expression refers to the ability of a program, system, or process to **alter its execution path dynamically** based on conditions, inputs, or run-time variables, rather than following a **rigid, predefined sequence**.

47: A **cache** is a fast temporary store that keeps copies of recently used data (or computed results) so later steps can reuse them quickly instead of re-reading from a slower source or re-computing from scratch.

48: This is one reason why two machines with the same core count can behave quite differently in practice: cache size, memory channels, and storage speed can matter as much as raw clock speed for data-heavy pipelines.

49: Hyperthreading is best thought of as a way to reduce wasted cycles during stalls: if the job is already heavy on compute, hyperthreading may not offer much in the way of improvement.

50: Dedicated hardware or software components that performs tasks simultaneously with other units.

51: Practically, GPUs are **throughput-oriented** devices: they perform better when presented with a large batch of similar operations with predictable memory access patterns.

### 30.5.1 Device-Level Parallelism

Most modern machines already offer some degree of **device-level parallelism** through **multi-core processors**. A **core** is a processing unit that can execute instructions independently, so a CPU with  $k$  physical cores can run up to  $k$  independent **threads of work**<sup>44</sup> at the same time (subject to memory and I/O constraints).

Less commonly, computers may have multiple processors, each of which may have multiple cores: the *Sunway TaihuLight* supercomputer, for instance, had 40k processors and 10m cores in 2017 [44].

A **CPU** (central processing unit) is designed for general-purpose computing: it is optimized for **low-latency execution**, **branch-heavy logic**,<sup>45</sup> and **flexible control flow**.<sup>46</sup> In many data analysis tasks, the CPU is not limited by arithmetic but by **memory bandwidth** and **cache behavior**:<sup>47</sup> the bottleneck is often feeding data to the cores rather than executing operations once the data arrives.<sup>48</sup>

Some CPUs support **hyperthreading** (or similar simultaneous multi-threading), where one physical core can schedule multiple **hardware threads**. The benefit is not that one core literally becomes two cores, but that the processor **can keep itself busy** when one thread is stalled waiting for memory or I/O.<sup>49</sup> [21]

**GPU** (graphics processing units) were originally designed for **graphics workloads**, but they are now widely used for **large-scale** machine learning [3, vol. 3] and numerical linear algebra [3, ch. 3, 4]. Compared to CPUs, GPUs typically have far more **parallel execution units**,<sup>50</sup> but each unit is simpler and the model of computation favours **regular, repeated operations** over large arrays.

This is why GPUs can be extremely effective for matrix operations, deep learning training, and other workloads where the same operation is applied to many data elements in a uniform way [37].<sup>51</sup>

A recurring practical caveat is **data transfer overhead**. Even if a computation is faster on the GPU, the overall workflow may not be faster if the pipeline must repeatedly move data between CPU memory and GPU memory. For smaller jobs, or for jobs with irregular control flow, the configuration and transfer overhead can outweigh the benefit of GPU parallelism [25].

On a personal machine, it can be useful to know how many physical cores are available. In Python, this could be achieved *via* the following.

```
import psutil
psutil.cpu_count(logical=False) # for physical cores
psutil.cpu_count(logical=True) # for logical cores
                                # (includes hyperthreading)
```

### 30.5.2 Scaling Beyond One Machine

When a dataset is too large for one machine, or when time-to-result becomes unacceptable, we often scale beyond device-level parallelism.

Building on the conceptual distinctions of Section 30.4.2, the key hardware configurations are **clusters**, **grids**, and **clouds**.

A **cluster** is a set of machines connected on a local network (often in the **same physical location**) that cooperate on large jobs. Clusters are attractive because they can be managed as a **single resource pool**, with standardized storage, scheduling, monitoring, and access control. In practice, clusters are often designed to be fairly **homogeneous** (similar CPUs, memory, disk), which simplifies performance tuning and reduces operational surprises.

A **grid** is a looser form of shared computing, often spanning **multiple sites** and potentially involving **heterogeneous machines** and **administrative domains**. Grid computing emphasizes **federation** (in the sense of multiple independent, autonomous entities acting as a single, unified resource pool) and **scheduling** across boundaries rather than tight coordination inside a single cluster. Grids amplify the importance of **task packaging**, **fault tolerance**, and **queueing policies** (see [3, ch. 24]).

Moving away from the buzzword sense, a **cloud** is access to computing resources in one (or more) data centres through **virtualization**. Users rent **virtual machines** with selected resources (CPU, memory, storage, sometimes GPUs), which can be scaled up or down over time (**elastic computing**). Flexibility is the main appeal: a team can acquire substantial capacity quickly **without owning hardware**, but in return they accept a less direct relationship to the physical machine and a cost model that depends on usage patterns.<sup>52</sup>

### 30.5.3 Cost and Performance Trade-Offs

Hardware decisions in big data settings are rarely made based only on performance: they are made under constraints of **budget**, **organizational overhead**, and the kinds of **failure modes** a team is willing (and able) to manage.

**Scaling up (vertical scaling):** buy or rent a **larger** machine (more RAM, more CPU cores, faster storage, possibly GPUs) so that a workflow fits comfortably on **one node**.

**Scaling out (horizontal scaling):** use **more** machines, coordinating them through a distributed framework so that data and computation are partitioned **across workers**.

**Vertical scaling** is attractive when the workload has substantial shared **state** or **sequential** components, when **network coordination** would dominate runtime, or when **simplicity** and **debuggability** matter most. If a pipeline fits on a single large-memory machine, many sources of distributed fragility disappear.<sup>53</sup>

A large join plus a group-by that keeps swapping on a 32 GB laptop provides a concrete example of vertical scaling; moving the same pipeline to a single 256-512 GB RAM machine with a temporary local and very fast solid state drive for storage can keep intermediate objects in memory and dramatically reduce runtime without introducing multi-worker coordination.

52: A useful mental model is that clouds are a way to **rent** hardware-like capabilities on demand. This is not automatically cheaper than local infrastructure, but it can reduce up-front commitment and shorten time-to-capacity.

53: Including the following:

**Network shuffles:** large data transfers across the cluster needed to realign records by key, typically triggered by joins, group-bys, or global sorts.

**Cross-node failures:** failures that arise because work is spread across many machines, so a single job is exposed to node loss, transient network issues, disk hiccups, executor crashes, and similar events. The framework must detect and recover by retrying tasks, re-reading partitions, rebuilding intermediate state, and so on.

**Skew amplification:** performance degradation caused by uneven key frequencies or uneven partition sizes; a hot key can cause one reducer or partition to receive far more data than others, so one worker becomes the bottleneck.

**Configuration overhead:** the organizational and technical overhead of setting up and tuning the distributed stack (cluster provisioning, dependency management, resource allocation, partitioning choices, executor memory, serialization settings, access control, monitoring, logs, version alignment).

**Horizontal scaling** becomes unavoidable when the data is simply too large to store or process on **one machine**, or when the workload naturally decomposes into **local computations** plus **aggregation**. At that point, the question becomes less about raw compute and more about how to manage **data locality**, **communication**, and **bottlenecks**.

The election counting analogy naturally matches horizontal scaling: split ballots into many independent boxes, compute local counts, then aggregate summaries. The pizzeria analogy provides a warning about bottlenecks: adding more cooks (more parallel workers) does not help if one oven limits throughput, so the relevant “fix” is to upgrade the constrained resource.

A useful practical rule is that scaling out pays off most when the workload is either (i) so **straightforwardly parallel** that it is almost trivial (many independent units of work), or (ii) naturally expressed in a **two-phase pattern** (large local passes producing small summaries, followed by centralized logic on those summaries, as in Sections 30.4.4 and 30.7.1).

### 30.5.4 Performance Metrics and Benchmarking

Because hardware options are diverse, we often attempt to summarize performance using a **single number**, always a risky proposition (consider the case of binary classifiers, for a similar notion [3, ch. 19]): **MIPS** (millions of instructions per second) and **FLOPS** (floating point operations per second) are two such metrics.

**MIPS** measures how many machine instructions a processor can execute per second. While intuitive, MIPS is difficult to interpret across architectures and instruction sets: an “instruction” is not a universal unit of work, and modern performance depends heavily on **memory hierarchies**, **vectorization**, and **compiler behaviour**. As a result, MIPS is often treated as outdated [26].

**FLOPS** focuses on floating-point arithmetic. This aligns well with **scientific computing** and parts of machine learning, where **large matrix** and **tensor operations** dominate runtime. FLOPS may be meaningful for **compute-bound workloads**, but it can still be misleading for big data pipelines that are **I/O-bound**, shuffle-heavy,, or constrained by memory bandwidth rather than arithmetic. A practical reminder is that many “big data” jobs are not limited by the speed of multiplication, but by the speed of **moving bytes**.<sup>54</sup>

**Peak vs. Achieved performance.** Even when a metric is well-defined, it is important to distinguish **peak** performance (measured under ideal conditions) from **achieved** performance (measured on a real pipeline).

**Peak** MIPS or FLOPS is usually reported for dense arithmetic with excellent data re-use, while **achieved performance** reflects the end-to-end workflow: parsing, joins, shuffles, caching, serialization, and I/O. Benchmarking is thus most useful when it resembles **realistic** and **practical** workloads.

54: A **compute-bound** workload is one whose runtime is limited mainly by **arithmetic throughput** (how fast the CPU or GPU can execute operations), rather than by waiting to **move data**; in contrast, an **I/O-bound** workload spends a large fraction of its time waiting for bytes to be read, written, transferred, or fetched from memory.

## Scaling “Laws” and Benchmarking Pitfalls

Several classical results [1, 11, 14, 16, 18, 23, 27, 31, 45, 46] may help explain why a single benchmark number is not the same thing as “how fast my job will run.”

**Amdahl’s Law:** if a fraction  $p$  of a workload is **perfectly parallelizable** and the rest is inherently **serial**, the speed-up on  $N$  workers satisfies

$$S(N) = \frac{1}{(1-p) + p/N}.$$

Even small serial components can limit the attainable speed-ups, which is why “more cores” can disappoint on pipelines with a hard sequential stage.

**Gustafson’s Law:** if problem size grows with available compute, parallelism can look much more favourable than Amdahl’s fixed-size setting suggests; one common form is

$$S(N) = N - (1-p)(N-1),$$

which matches many big data workflows where the objective is higher throughput on larger datasets rather than lower runtime on a fixed dataset.

**Little’s Law:** in a steady state, the average number of items in a system satisfies

$$L = \lambda W,$$

where  $\lambda$  is the arrival rate and  $W$  is the average time in system. This compactly links **throughput** and **latency**: queues form when any stage cannot keep up, and latency rises even if individual operations are fast.

**Moore’s Law:** an empirical observation that transistor counts on economically viable chips grew roughly exponentially for decades, often summarized as a **doubling every 18 to 24 months**. The practical implication was sustained long-run gains in cost, density, and compute capacity, although the original cadence has slowed in recent years [2].

**Dennard scaling:** a transistor-scaling rule that, for many years, allowed smaller devices to operate at lower voltage with roughly constant power density, so designers could raise **clock speed** without an unsustainable increase in heat. Once this scaling weakened, frequency growth largely stalled and performance gains shifted toward parallelism and accelerators.

**Koomey’s Law:** an empirical trend that computations per joule improved roughly exponentially over long horizons. The practical implication is that **energy efficiency** and power constraints strongly shape real-world performance, especially in data centers and GPU-heavy workloads.

**The Memory Wall:** compute capacity has tended to improve faster than **memory latency** and **memory bandwidth**, so many workloads

become **memory-bound**: the processor spends time waiting for data to arrive rather than executing arithmetic at its peak rate.

**Roofline model**: the Roofline model summarizes this constraint by bounding achievable performance as

$$P_{\text{achieved}} \leq \min(P_{\text{peak}}, I \cdot B),$$

where  $P_{\text{peak}}$  is peak compute throughput,  $B$  is memory bandwidth, and  $I$  is **arithmetic intensity** (useful work per byte moved). It makes precise the idea that high FLOPS does not help if the pipeline must stream large volumes of data with little re-use.

**Goodhart's Law**: when a measure becomes a target, it ceases to be a good measure. Benchmarks can be optimized in narrow ways that inflate a headline number without improving the end-to-end workload about which we really care.

### Cost Declines and the “Throw Money at It” Heuristic

A recurring empirical pattern in hardware is that the **amount of computation we can buy for a dollar** has increased dramatically over time. One way to summarize this trend is that **MIPS per dollar** has grown by roughly a factor of 10 every five years or so since the mid twentieth century, while **FLOPS per dollar** has grown by roughly a factor of 10 every eight years or so (as a coarse rule of thumb; see Figure 30.6).<sup>55</sup>

This kind of scaling has led to a pragmatic attitude that shows up in many engineering teams: if the bottleneck is purely **compute capacity**, then a large part of the long-run “solution” is simply to **pay** for more hardware, and if the timeline allows it, to **wait** for the same budget to buy more capacity [15, 36].

To see why, suppose we measure the total capacity of a cluster using a compute-centric metric such as MIPS. If we (naïvely) assume that a workload can be parallelized with **near-100% efficiency** across available cores and machines, then the historical tenfold improvement in MIPS per dollar suggests that, every five years or so, the **compute power** of a cluster we can buy for a fixed budget (say \$1000) increases by an order of magnitude.<sup>56</sup>

However, this heuristic comes with an important caveat: the remaining issue is rarely “hardware exists” but rather whether the **software stack and workload** can actually **use** the hardware effectively.

If the task is I/O-bound, if the pipeline triggers expensive shuffles, or if the algorithm has a hard sequential component, then additional cores may not buy much improvement. In that case, “throwing money at it” either fails outright, or it simply moves the bottleneck from compute to **data movement, coordination, and system design**.

Either way, benchmarking numbers are most useful when interpreted **in context**: what kind of workload is being run (compute-bound vs. I/O-bound), what fraction can be parallelized, what kind of data re-use is present, and where bottlenecks are likely to emerge.

55: These are back-of-the-envelope rates and they hide a lot of variation across eras and workloads. They are useful as a high-level economic trend, not as a precise forecasting tool.

56: This assumption is intentionally optimistic. In practice, speed-ups are limited by sequential components and coordination overhead (see Section 30.4.4).

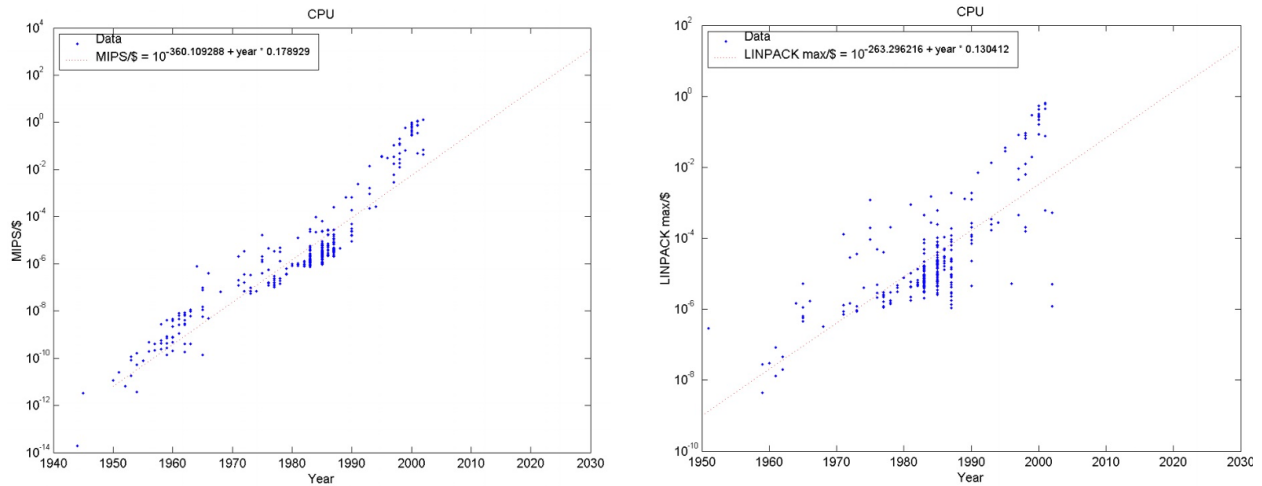


Figure 30.6: Trends in instruction throughput and floating-point throughput per dollar [15].

## 30.6 Software Solutions

Big data constraints do not disappear once we adopt a cluster or a cloud platform: they are merely **managed by software**. In practice, big data software solutions fall into two overlapping categories:

- **single-machine tooling** that stretches what can be done on one computer by reducing copying, streaming from disk, and using efficient storage formats, and
- **distributed frameworks** that coordinate many machines so that data and computation can be partitioned across workers while maintaining a coherent user-facing abstraction.

In this section, we will outline the main ideas and roles of these tools, and discuss their integration with R and Python (in light of the third comment in the Preface, on p. v).<sup>57</sup> The objective is to help the reader recognize **which constraint is binding**<sup>58</sup> and to select software accordingly.

### 30.6.1 R and Big Data Tooling

R was not designed with extremely large datasets in mind: most objects are held in RAM, and many of its base operations historically encouraged copying.<sup>59</sup> That said, there are now several mature tools that make R workable for datasets that are large (relative to a laptop) and that provide on-ramps to distributed systems when a single machine is no longer enough.

The **ladder of escalation** is a useful mental model:

1. first, make the single-machine workflow **less wasteful** (avoid copying; use efficient representations);
2. next, adopt **out-of-memory** storage formats and query engines to avoid loading raw data unnecessarily, and
3. finally, if the workload truly requires it, use R as a *front-end* to a distributed framework.

The three tools on the next page match these three rungs.

57: The alternative, which is to attempt to catalogue and describe every software option on the market, would hit a snag even if we managed to do it: **software has a shelf life**: five years from now (2031), would anyone expect any but a few tools to still be current?

58: Memory, I/O, shuffles, coordination, failure modes, etc.

59: More details are available in [3, ch. 1].

60: This term refers (informally) to workflows that repeatedly allocate, copy, and discard large in-memory objects, so that **memory management** (allocation, copying, garbage collection, cache misses, and sometimes swapping) becomes the bottleneck rather than the intended computation. It is often triggered by pipelines that create many large intermediate tables (for example, repeated joins, reshapes, or copy-on-modify updates), and it is a common reason that an analysis “feels slow” even when each individual operation is conceptually simple.

61: Or key intermediate objects.

62: A common performance trick: when we filter rows (the **predicate**, e.g., `date == "2026-01-01" or amount > 100`), the framework tries to apply that filter as close to the data source as possible (see Section 30.4.4) rather than reading everything and filtering later. In a columnar format such as Parquet (see Section 30.7.3), this can allow the reader to skip whole partitions and row groups using metadata (for example, per block min/max statistics), which reduces **I/O** and **data movement**. If the predicate cannot be pushed down (for example, because it involves a complex user-defined function), the system typically has to scan much more data before discarding non-matching rows.

63: This refers to repeated conversion of records between an **in-memory representation** and a **byte-level encoding** as data crosses stage boundaries (network shuffles, disk I/O, or process- and language-runtime boundaries). Each cycle of “serialize → transfer/store → de-serialize” costs CPU time and often creates many temporary objects, increasing allocation and garbage-collection pressure; it can also inflate I/O and network traffic (especially with verbose formats like CSV or JSON).

64: In a typical workflow, the analyst writes familiar `dplyr`-style code against a remote Spark DataFrame; execution is **lazy** (plans are built first, then executed), and results are only brought back to the local session when explicitly requested.

## data.table

The `data.table` package provides a high-performance table abstraction that is particularly effective when the dataset is large enough that base-R copying is “painful,” but still small enough that a single machine can handle the work. The practical gains come from two design choices:

- operations are optimized to reduce **unnecessary intermediate objects** (which reduces memory pressure and speeds up pipelines);
- common tasks such as grouping, joining, and filtering are implemented with care towards **indexing** and **cache-friendly execution**.

In other words, `data.table` is often the first place to look when a project is failing due to **RAM churn**<sup>60</sup> [13].

## Arrow

The **Arrow** ecosystem addresses a different failure mode: the raw data<sup>61</sup> do not fit comfortably in memory, and repeated scans of large files become I/O-dominated. Arrow provides a language-agnostic, columnar representation of data and integrates with columnar file formats (notably **Parquet**) that support efficient scanning and **predicate pushdown**.<sup>62</sup>

From an R workflow perspective, Arrow is most useful as a bridge between `dplyr`-style transformations and **out-of-memory** storage:

- data can be filtered, selected, summarized **without** being fully loaded into RAM;
- computations can be executed close to storage, while only **reduced results** are pulled into R using `collect()`.

Arrow also facilitates inter-operation between R and Python by reducing the need for expensive **data re-serialization**<sup>63</sup> when passing tables between languages [35].

## sparklyr

When the job truly requires distributed execution, `sparklyr` provides an R-facing interface to **Apache Spark** (see Section 30.7.2), allowing R users to express transformations and analyses that are executed on a cluster<sup>64</sup> [29].

`sparklyr` is particularly attractive when:

- the data is **already stored** in a distributed environment (Hadoop Distributed File System, object storage, a data lake, see [3, sec. 14.5]);
- the pipeline is dominated by **large joins, aggregations, and feature engineering** that Spark can parallelize and **schedule effectively**;
- organizationally, the team **wants** R as an analyst interface while relying on a shared cluster for heavy lifting.

We will return to a concrete `sparklyr` workflow later in the chapter (see Section 30.6.4 and the practical infrastructure discussion in Section 30.8).

### 30.6.2 Python and Distributed Tooling

While R has strong **single-machine** data analysis traditions, Python has become a common language for working directly **inside distributed systems** (and for integrating data engineering with model training).<sup>65</sup>

65: More details are available in [3, ch. 1].

#### PySpark

**PySpark** is the Python interface to **Apache Spark**. Conceptually, it provides the same core abstraction as `sparklyr`: a user manipulates distributed `DataFrames` and transformations are **scheduled** and **executed** across a cluster.

A key practical advantage of Spark, whether used from R or Python, is that it supports the kind of workflow that repeatedly appears in big data projects:

- **parse** and **clean** raw event data into a stable **tabular representation**;
- perform **large-scale joins** and **aggregations** to build **features** and **summaries**;
- run **iterative** or **interactive analyses** on derived tables, often with caching and re-use.

This is precisely where many one-machine workflows become fragile: not because the arithmetic is sophisticated, but because the pipeline repeatedly stresses memory limits, I/O, and coordination.

A practical warning (that applies equally to R interfaces) is that **bringing data back to the Python session** is where distributed workflows often fail; `collect()` is a powerful tool, but it should be used with intention: the point of distribution is to keep the **large objects distributed**, and to pull back only reduced summaries or small samples.

#### Distributed Training Libraries

Once feature engineering and data preparation are distributed, model training often becomes the next scaling pressure, especially for deep learning (DL). There are two broad approaches:

- **data parallelism** replicates the model across workers, splits the data, computes local gradients, then aggregates (often via an all-reduce style communication pattern);
- **model parallelism** splits the model itself across devices when it is too large to fit on a single accelerator.

The important connection is the same “**two-phase**” logic from Section 30.4.4: local passes over large data produce **small summaries**, which are then combined to produce a **parameter update**.

Modern DL libraries expose these patterns directly. For example, **PyTorch** provides distributed training primitives via `torch.distributed`, and higher-level orchestration layers can manage **worker launch**, **resource allocation**, and **fault handling** [9].<sup>66</sup>

66: Readers who want higher-level management of distributed training across CPUs and GPUs should consider *Ray Train*, which provides a structured interface for scaling training functions and coordinating workers [39, 38].

The main DL context and training trade-offs are treated in Chapter 31.<sup>67</sup>

67: Including the relationship between **minibatching**, **communication cost**, and **convergence**.

### 30.6.3 What Frameworks Do

It is tempting to think of a **distributed framework** as “a faster way to run `group_by` and `join`,” in reality, frameworks are useful because they automatically handle the **non-mathematical** work that becomes dominant at scale: **scheduling**, **memory management**, **re-tries**, and **monitoring**. These components convert the conceptual ideas of Section 30.4 into an **operational system**.

#### Scheduling

At a minimum, a framework must be able to decide:

- **where** each stage of computation runs (which machines, which cores, which **accelerators**<sup>68</sup>);
- **when** it runs (queuing, priorities, and resource contention);
- **how** data movement is organized (shuffle plans, locality preferences, and broadcast decisions).

Good scheduling is inseparable from the bottleneck logic discussed earlier: the objective is not only to keep CPUs busy, but to avoid turning the pipeline into a data-movement machine.

#### Memory Management

On a cluster, **memory pressure**<sup>69</sup> is not a **rare corner case**.<sup>70</sup> Frameworks therefore must implement explicit strategies to:

- **cache** intermediate results when re-use is valuable;
- “**spill**” to disk when memory is insufficient;
- choose **serialization** formats and execution plans that reduce copying.

From the user perspective, this is why the same high-level computation can behave very differently depending on whether it triggers a **large shuffle**, whether it materializes **wide intermediate tables**, or whether it **repeatedly re-reads raw data**.

#### Fault Tolerance

In long-running workflows, we expect failure: machines **restart**, network links **glitch**, disks **fill**, and processes **crash**. Consequently, distributed frameworks implement mechanisms that make failures **survivable**:

- tasks can be **re-tried** on other workers;
- intermediate data may be **replicated** or **rebuilt from lineage**,<sup>71</sup> and
- checkpoints may be used to **limit re-computation cost**.

This is one of the reasons big data tooling is not only about speed: it is also about making large-scale pipelines **reliable** enough to operate in **production settings**.

68: In distributed systems, **accelerators** are specialized hardware devices that a framework can schedule alongside CPUs to speed up particular kinds of computations. The most common examples are **GPUs** (widely used for deep learning and large matrix operations), but accelerators can also include **TPUs** (Google’s **tensor processing units**), **FPGAs** (reconfigurable hardware sometimes used for low-latency inference or streaming workloads), and so on. Scheduling matters because using an accelerator can reduce compute time, but it can also introduce data-transfer overhead and resource contention that the framework must manage.

69: This means that a job’s input partitions, intermediate tables, shuffle buffers, cached data, per-task overhead, etc. are large enough that available RAM is tight. The system then has to take costly actions such as spilling intermediates to disk, running more frequent “garbage” collection, evicting cached data (causing re-computation), or even failing and re-trying tasks due to out-of-memory errors.

70: It is, in fact, a typical and often encountered situation, which cannot be ignored.

71: Or re-computed from the recorded history of how it dataset was produced, using its **lineage graph** (a description of which upstream partitions and transformations created it). If a cached partition or intermediate file is lost (say, because an executor crashes), the framework can re-run the necessary upstream steps on the relevant input partitions to regenerate **exactly that missing piece**, instead of restarting the entire job.

### 30.6.4 Selecting a Framework

Choosing a framework is not purely a technical decision; it is a decision about **workflow**, **organizational cost**, and the **kinds of failures we are willing to manage**.

The following questions can help keep the choice grounded.

**1. What is the dominant constraint?** If the main problem is **copying** and **memory churn** on one machine, `data.table` may be enough. If the problem is **out-of-memory storage** and **repeated scanning**, Arrow-style columnar formats can provide the right tools. If the problem is **truly distributed** (data too large, or shuffles and joins too costly on one machine, say), then Spark-like systems become relevant.

**2. What workload pattern are we running?** Batch “**extract, transform, load (ETL)**” pipelines, **interactive analysis**, **streaming updates**, and **iterative model training** all use and stress frameworks differently.

Adopting an ecosystem optimized for one pattern (for example, batch throughput) when the real requirement is another (for example, low-latency monitoring or rapid iteration) is a common cause of **mismatched tooling**.

**3. What is the team and infrastructure reality?** A framework that is theoretically ideal can be a poor choice if it increases **organizational cost** beyond what the team can sustain. This includes:

- **operational expertise** (deployment, upgrades, access control, incident response);
- **integration with storage and identity systems** already in use;
- **reproducibility and governance needs**.

This is the same value-vs.-cost perspective emphasized in Section 30.2.3: a tool is justified when it improves the **quality** and **timeliness** of decisions *enough* to outweigh added **complexity**.

**4. What is the cheapest pipeline that answers the question?** A good rule of thumb is to:

- build a **prototype** on a sample (or a restricted time window), **validate invariants**,<sup>72</sup> and **stabilize definitions**,<sup>73</sup>
- **scale out** only once the pipeline is answering the *right* question **reliably**.

This helps avoid the common failure mode where distributed tooling is adopted to **compensate for unclear objectives** or **unstable data semantics** (i.e. schema drift).

72: Checking properties that *must* hold by **definition** (or by basic bookkeeping) before we trust any downstream results: **counts** and **tallies** reconcile across stages, join keys do not unexpectedly inflate **row counts**, **units** and **ranges** are plausible, **rates of missing values** are stable or explainable, and aggregate totals match independent summaries.

73: Making the meaning of variables and events explicit and consistent **before** scaling; decide what exactly counts as an “active user,” “session,” “conversion,” “duplicate,” “valid record,” etc. and **document** those choices (and their edge cases), ensuring that the same definitions are **implemented uniformly** across time windows, data sources, and pipeline versions so that later changes in results reflect **real system changes** rather than **shifting semantics**.

---

We will apply these principles in later sections by focusing on Spark as a concrete framework and by emphasizing which ideas transfer **across platforms**.

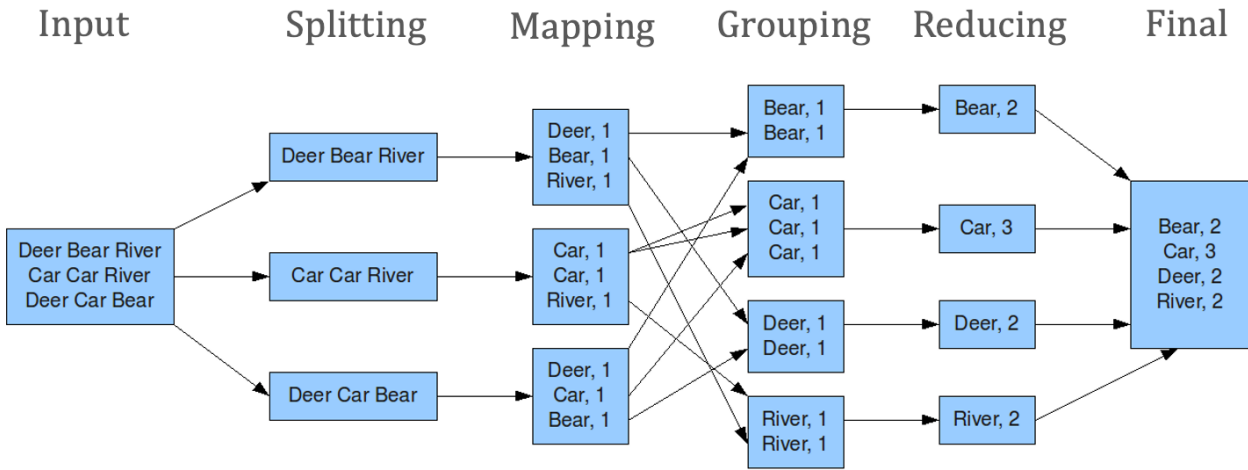


Figure 30.7: The MapReduce pattern: after splitting the data, map produces key-value pairs, shuffle and sort groups by key, and reduce aggregates values per key. In this example, the problem is to tally the frequency of words in a text (author unknown).

### 30.7 Practical Considerations

In this section, we connect the big data principles (**data locality**, **bottlenecks**, and the idea that **scale** often changes the **workflow** before it changes the mathematics) to concrete systems. We focus on the MapReduce lineage (and Hadoop) as a historical baseline, then on Spark as the modern default for many batch and interactive analytics workflows, and we conclude with storage and cluster-facing details that are crucial to real big data projects.

#### 30.7.1 MapReduce and Hadoop

##### Map, Shuffle and Sort, Reduce

When large-scale data processing first became unavoidable in industry, a central design objective was to make distributed computation **conceptually simple** for programmers while still being robust to failures and scalable across large clusters. The **MapReduce** model, introduced at Google in 2004, was an influential answer to that problem, [10].

At a high level, MapReduce decomposes a computation into:

- a **map** step that transforms input records into intermediate **key-value pairs**,
- a **shuffle and sort** step (also called grouping) that groups all values associated with the same key,
- a **reduce** step that aggregates the grouped values key-by-key.

One standard example is word counting; the key is the word, and the value is a small count.<sup>74</sup> The overall logic is to count locally, then add counts across partitions.

##### 1. The Map Phase

- The input data is split into  $M$  pieces. The system then starts multiple copies of the MapReduce program to be run in parallel.

74: The process is illustrated in Figure 30.7.

- One program copy, called the **master**, assigns tasks to the other copies, called **workers**. There are  $M$  map tasks and  $R$  reduce tasks, with the number of workers  $W$  typically much smaller than the number of tasks.
- When assigned a mapping task, a worker takes an input chunk and returns a set of key-value pairs.
- The outputs are temporarily saved in memory and periodically dumped to local disk. After dumping, the worker reports the save location back to the master.

## 2. Shuffle, Sort, Grouping

- The master tells the reduce workers where the map outputs are saved.
- The reduce worker groups values with the same keys together. Consider the following word counting example. If the mapped output is:

```
[('dog', 1), ('dog', 1), ('cat', 1), ('dog', 1), ('cat', 1), ('dog', 1)]
```

then the grouped output would be:

```
[('cat', [1,1,1]), ('dog', [1,1,1,1])]
```

## 3. The Reduce Phase

- The reduce worker iterates over the sorted data. For each key and its corresponding set of values, it applies the reduce function. Continuing the word counting example,

```
# for "dog"
reduce(add, [1,1,1,1])
```

becomes

```
reduce(add, [1,1,1,1]) = add(add(add(1,1),1),1)
                       = add(add(2,1),1) = add(3,1) = 4
```

The key connection to Section 30.4.4 is that MapReduce hard-codes a **communication pattern**, which have seen before: do large, local passes over distributed data to produce small intermediate objects, then move only what is needed to aggregate by key. When this pattern matches the problem at hand, scale can be achieved **without exposing the user to manual, low-level coordination**.

## Why Associativity and Commutativity Matter

MapReduce relies on an important mathematical convenience: the reduce operation should behave **sensibly** even when values arrive in an **arbitrary order** and when partial reductions are combined **hierarchically** (first inside workers, then across workers, then across racks, and so on). In effect, the reduce step must satisfy two algebraic properties.

**Associativity:** an operation  $\circ$  is associative if  $(a \circ b) \circ c = a \circ (b \circ c)$  for all  $a, b, c$ ; associativity guarantees that we can reduce values in **chunks** (or in a tree) without changing the result.

**Commutativity:** an operation  $\circ$  is commutative if  $a \circ b = b \circ a$  for all  $a, b$ ; commutativity guarantees that arbitrary orderings introduced by **shuffles** does not change the result.

75: This is also why many systems distinguish between “reduce” operations that can **safely** combine partial results (associative, often also commutative, but not always necessarily so) and more delicate operations that require **full ordering** or **full grouping**.

76: The basic unit of resource allocation on a cluster node: a bundle of resources (typically **memory** and **vCores**) reserved on a specific machine for a specific application. The **ResourceManager** grants containers, the application’s **ApplicationMaster** requests them and decides what to run in them, and the local **NodeManager** launches and monitors the actual process inside the container (a map task, a reduce task, or a Spark executor). A YARN container relates to resource isolation and accounting in a Hadoop cluster, unlike OS-level containerization in Docker, say.

Most simple functions (sum, count, min, max) satisfy both properties, which is why MapReduce is a natural fit for **aggregations**.

When these properties fail (for example, for operations such as subtraction, division, taking the first element of, averaging averages, etc.), a system must either enforce a **global order** explicitly, which can be expensive, or accept that “parallelizing” the computation **changes the question** being answered (as in the sessionization sidenote in Section 30.4.4).<sup>75</sup>

### Hadoop Components in Context

The original Google MapReduce system was proprietary, but the model quickly influenced open-source infrastructure. In 2008, Yahoo released **Hadoop** as an open-source implementation, [41].

In the Hadoop ecosystem, it is useful to separate three roles:

**HDFS:** the **Hadoop Distributed File System** stores files by splitting them into blocks distributed across machines and replicating blocks for fault tolerance, [17]; this is a storage layer optimized for large sequential reads and writes rather than low-latency random access.

**YARN:** **Yet Another Resource Negotiator** is a cluster resource manager; it schedules **containers**,<sup>76</sup> allocates CPU and memory, and enforces queuing and policy decisions.

**MapReduce:** the execution model that runs map and reduce tasks and coordinates the shuffle.

Hadoop is historically important because it made **large-scale storage** and **batch processing** available outside Google. In many modern stacks, the direct MapReduce execution engine is infrequently used, but the broader ideas (**distributed storage**, **resource management**, and the **map-shuffle-reduce** communication pattern) remain central.

## 30.7.2 Apache Spark

### Why Spark is Faster (memory, DAG scheduling, lazy evaluation)

Spark was designed to address common **workflow bottlenecks** in real analytical workflows, especially when computations are **iterative** and intermediate results must be re-used.

- **In-memory re-use:** intermediate results can be cached (when this is beneficial) rather than written to disk after every stage;
- **DAG scheduling:** Spark builds a **directed acyclic graph (DAG)** of stages so it can pipeline narrow transformations and isolate the true shuffle boundaries;
- **Lazy evaluation:** transformations build a plan first and only execute when an **action** is called, which allows the system to optimize the plan and avoid unnecessary work.

This is the same principle discussed earlier under **computing near the data** (Section 30.4.4): the biggest win is often not faster arithmetic, but less **data movement** and fewer forced materializations.

## RDDs vs DataFrames, and Why DataFrames Are Now Standard

Spark originally centered on the **resilient distributed dataset (RDD)** abstraction: a distributed collection of records with fault tolerance *via* lineage. RDDs remain important conceptually because they make the dependency structure **explicit**.<sup>77</sup>

In most modern Spark workflows, however, **DataFrames** (and Scala's related "Dataset" abstraction) are the default user-facing interface:

- they carry a **schema** which the engine uses to reason about columns, types, and operations at a higher level than arbitrary user code;
- this higher-level representation enables **stronger optimization** of joins, filters, projections, and aggregations, and
- they match how most analysts naturally express work, mirroring tables and the kinds of transformations used in SQL and in dplyr-style pipelines.

From a practical perspective, this is why both **PySpark** and **sparklyr** emphasize DataFrames: they align well with the typical ETL and feature-engineering stages that dominate big data projects.<sup>78</sup>

77: If one partition is lost, Spark can recompute it from upstream transformations.

78: We will see concrete examples shortly.

## Where MapReduce Still Appears in Spark Workflows

Even though Spark is not "MapReduce," MapReduce-style logic appears constantly inside Spark jobs:

- many transformations are still **map-like** (apply a function to each record or each partition);
- many aggregations are still **reduce-like** (combine values by key using an associative aggregation), and
- **shuffles** are still the defining boundary between fast, local stages and expensive, global coordination.

The mental model is that Spark **generalizes** MapReduce by allowing **multiple** map and reduce style stages to be composed into a DAG.<sup>79</sup> But the same warning still applies: if the job triggers repeated wide shuffles or suffers from severe key skew, the pipeline becomes bottlenecked regardless of how modern the framework is.

79: With caching and re-use where helpful.

### 30.7.3 Parquet and Columnar Storage

DataFrames are reminiscent of **comma-separated value (CSV)** files, with observations in rows and features in columns. CSV-type files remain attractive to many analysts because they are simple and portable, but at scale, they become recurring bottlenecks, due to:

- **parsing costs** (CSV is text, so reading it requires expensive parsing);
- **row orientation** (CSV is row-based, but reading a few columns still requires scanning and parsing entire rows);
- **weak metadata** (CSV does not naturally store column statistics that would allow readers to skip large blocks of data during filtering);
- **compression trade-offs** (compression helps storage, but it can make selective access slower if the format does not support good block-level skipping), etc.

In big data regimes, these costs are usually felt as slower iteration and higher storage and compute bills rather than as a single dramatic failure: they therefore tend to accumulate **quietly** until the pipeline becomes unpleasant to run.

**Parquet** is a columnar storage format designed for analytical workloads. The central idea is that a typical analysis uses a subset of columns, so the storage layout should make it cheap to read only what is needed.

In practice, Parquet often improves performance due to:

- **column pruning** (queries that select a small set of columns can avoid reading the rest);
- **better compression and encoding** (within a column, values tend to be similar (or at least type-consistent), which improves compression), and
- **block metadata** (readers can use per-block statistics such as min/-max, etc. to skip blocks that cannot satisfy a filter, which is a form of predicate pushdown) [42].

This connects directly to the chapter's recurring theme: Parquet is valuable not because it is fashionable, but because it helps us **move fewer bytes** for the same analytical question.

### 30.7.4 Working With Clusters

Spark's cluster execution model is often summarized using four terms.

**Driver:** the process that builds the **logical plan** to run the code, requests resources, and coordinates execution.

**Workers:** the machines that **run the computations**.

**Executors:** the long-lived Java Virtual Machine (JVM) processes on workers that **execute tasks** and hold **cached data**.

**Partitions:** the units into which datasets are split for parallel processing.

From R, the standard workflow is to connect using `sparklyr` and then work with Spark DataFrames through familiar `dplyr`-style verbs.

Spark is built in the Scala programming language, which is run on the Java Virtual Machine. Before installing Spark, we need to ensure that Java 8 is installed on the system, as is Hadoop and Java Development Kit; instructions are available in [28, ch. 2].<sup>80</sup>

The following examples illustrate a typical connection, configuration inspection, and reconnection after configuration changes.

To connect R to Spark, we use the `sparklyr` package. The connection is established through the `spark_connect()` function, which allows R to communicate with the Spark cluster.<sup>81</sup> Once `sparklyr` is installed and loaded, we configure Spark.

80: The same highly-recommended reference provides an introduction to `sparklyr`.

81: The paths will change based on where the files have been installed.

#### Connecting to and Configuring Spark (R)

```
Sys.setenv(HADOOP_HOME = "C:/hadoop")
Sys.setenv(PATH = paste("C:/hadoop/bin", Sys.getenv("PATH"), sep = ";"))
```

**Connecting to Spark (R)**

```

hadoop_java_opt <- "-Dhadoop.home.dir=C:\\hadoop"
spark_home <- "C:/Users/boily/AppData/Local/spark/spark-3.5.0-bin-hadoop3"
spark_submit <- normalizePath(file.path(spark_home, "bin", "spark-submit.cmd"),
                               winslash = "\\ ", mustWork = TRUE)
examples_jar <- normalizePath(file.path(spark_home, "examples", "jars",
                                       "spark-examples_2.12-3.5.0.jar"),
                               winslash = "\\ ", mustWork = TRUE)

# install.packages("sparklyr") # uncomment if not installed
library(sparklyr)

conf <- spark_config()
conf$spark.driver.extraJavaOptions <- hadoop_java_opt
conf$spark.executor.extraJavaOptions <- hadoop_java_opt

conf$sparklyr.gateway.address <- "127.0.0.1"
conf$sparklyr.gateway.port <- 8880
conf$sparklyr.gateway.start.timeout <- 240
conf$spark.driver.host <- "127.0.0.1"
conf$spark.driver.bindAddress <- "127.0.0.1"
conf$spark.executor.memory <- "2g"
conf$spark.executor.cores <- 1
conf$spark.dynamicAllocation.enabled <- TRUE

sc <- spark_connect(master = "local", spark_home = spark_home, config = conf)
sc

```

```

$master
[1] "local[12]"

$method
[1] "shell"

$app_name
[1] "sparklyr"

$config
$spark.env.SPARK_LOCAL_IP.local
[1] "127.0.0.1"

$sparklyr.connect.csv.embedded
[1] "^1.*"

$spark.sql.legacy.utcTimestampFunc.enabled
[1] TRUE

$sparklyr.connect.cores.local
[1] 12

$spark.sql.shuffle.partitions.local
[1] 12

```

```
$spark.driver.extraJavaOptions
[1] "-Dhadoop.home.dir=C:\\hadoop"

$spark.executor.extraJavaOptions
[1] "-Dhadoop.home.dir=C:\\hadoop"

$sparklyr.gateway.address
[1] "127.0.0.1"

$sparklyr.gateway.port
[1] 8880

$sparklyr.gateway.start.timeout
[1] 240

$spark.driver.host
[1] "127.0.0.1"

$spark.driver.bindAddress
[1] "127.0.0.1"

$sparklyr.shell.name
[1] "sparklyr"

$`sparklyr.shell.driver-memory`
[1] "2g"

$state
<environment: 0x00000208d65973a8>

$extensions
$extensions$jars
character(0)

$extensions$packages
character(0)

$extensions$initializers
list()

$extensions$catalog_jars
character(0)

$extensions$repositories
character(0)

$extensions$dbplyr_sql_variant
$extensions$dbplyr_sql_variant$scalar
list()

$extensions$dbplyr_sql_variant$aggregate
list()
```

```

$extensions$dbplyr_sql_variant$window
list()

$spark_home
[1] "C:\\Users\\boily\\AppData\\Local\\spark\\spark-3.5.0-bin-hadoop3"

$backend
A connection with
description "->127.0.0.1:8881"
class      "sockconn"
mode       "wb"
text       "binary"
opened     "opened"
can read   "yes"
can write  "yes"

$monitoring
A connection with
description "->127.0.0.1:8881"
class      "sockconn"
mode       "wb"
text       "binary"
opened     "opened"
can read   "yes"
can write  "yes"

$gateway
A connection with
description "->127.0.0.1:8880"
class      "sockconn"
mode       "rb"
text       "binary"
opened     "opened"
can read   "yes"
can write  "yes"

$output_file
[1] "C:\\Users\\boily\\AppData\\Local\\Temp\\RtmpQbBHK0\\file864c67e73671_spark.log"

$sessionId
[1] 20062

$home_version
[1] "3.5.0"

attr(,"class")
[1] "spark_connection"      "spark_shell_connection"
[3] "DBIConnection"

```

When troubleshooting a pipeline, connecting to a **local** Spark session can be simpler than starting on a remote cluster. A local session reduces moving parts and makes it easier to isolate whether a failure is due to code, configuration, or networking.

We will provide examples of machine learning at scale in Section 30.8.

### Common Performance Issues

Most performance surprises in Spark can usually be traced back to a small set of issues.

**Shuffles:** wide transformations (joins, group-bys on large keys, global sorts) trigger **network-heavy** redistribution.

**Skew:** when a **small number of keys** dominate frequency, one partition becomes much larger than others, creating a **straggler task**.

**Caching misuse:** caching can help iterative workflows, but caching the **wrong intermediate** (or caching too many intermediates) can create **memory pressure** and **spill-to-disk** behaviour.

**Collecting too much:** pulling **large results** back to the local R session defeats distribution and often fails due to **memory limits**.

The most reliable way to stay out of trouble is to keep returning to the chapter's **central question:** what is the cheapest workflow that answers the question? In practice, that often means designing transformations so that the heavy work happens **close to storage** and only **small summaries** are collected **locally**.

## 30.7.5 AWS EMR workflow

### Creating an EMR Cluster

In this section, we outline a practical workflow for running Spark on [Amazon EMR](#) (cloud computing). The objective is not to memorize click paths in a console, but to understand which pieces must be configured: the **EMR release** (software versions), the **applications** to install (Spark, Hadoop, Livy, and related components), and the instance groups that determine the cluster's shape.

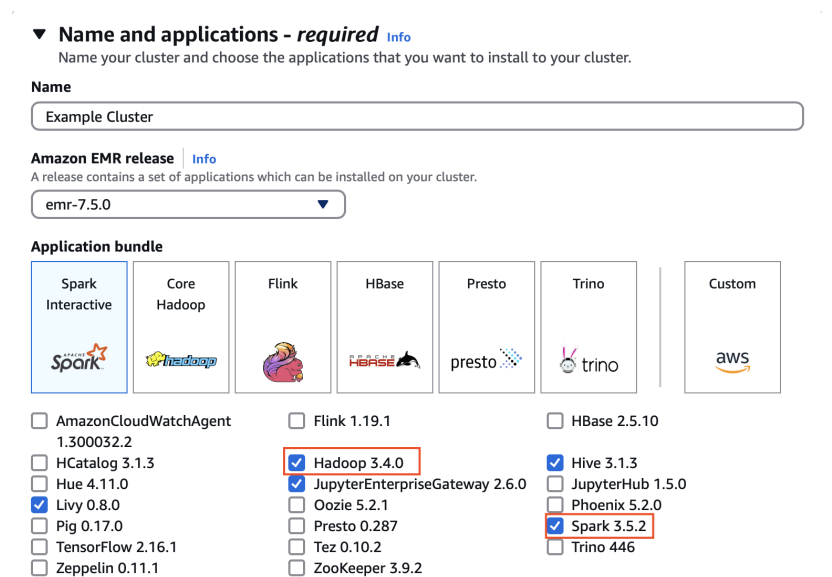


Figure 30.8: Creating the EMR cluster.

## Core Configuration

A typical initial configuration separates nodes into a **master node** (coordination), **core nodes** (storage and compute), and optional **task nodes** (compute-only capacity). Instance types and attached storage determine **memory pressure**, **spill behaviour**, and **shuffle throughput**.

**▼ Cluster scaling and provisioning - required** [Info](#)  
Choose how Amazon EMR should size your cluster.

Choose an option

**Set cluster size manually**  
Use this option if you know your workload patterns in advance.

**Use EMR-managed scaling**  
Monitor key workload metrics so that EMR can optimize the cluster size and resource utilization.

**Use custom automatic scaling**  
To programmatically scale core and task nodes, create custom automatic scaling policies.

**Provisioning configuration**  
Set the size of your core and task instance groups. Amazon EMR attempts to provision this capacity when you launch your cluster.

| Name     | Instance type | Instance(s) size | Use Spot purchasing option |
|----------|---------------|------------------|----------------------------|
| Core     | m5.xlarge     | 4                | <input type="checkbox"/>   |
| Task - 1 | m5.xlarge     | 4                | <input type="checkbox"/>   |

Figure 30.9: Setting up instances (nodes).

**Autoscaling** can reduce cost when workloads are bursty, but it can also introduce **performance variability** and makes benchmarking more subtle.

## Networking and Access

Remote cluster access requires an **SSH key pair**, **security group rules**, and **IAM roles** that allow EMR to interact with S3 and other services. The key pair must be stored securely because it cannot be retrieved again if lost.

**▼ Security configuration and EC2 key pair** [Info](#)  
Choose a security configuration or create a new one that you can reuse with other clusters.

**Security configuration**  
Select your cluster encryption, authentication, authorization, and instance metadata service settings.

**Amazon EC2 key pair for SSH to the cluster** [Info](#)

**⚠ You haven't entered an EC2 key. If you're outside a VPN and want to enable SSH or use Hue SQL assistant with this cluster, you must enter an EC2 key.**

Figure 30.10: Creating a key pair.

Security groups must allow inbound access from the IP on the relevant ports. A common configuration is:

- **Port 8787:** RStudio Server
- **Port 8998:** Livy
- **Port 8088:** YARN ResourceManager

## Connecting With sparklyr and Validating Executors

Once a R session is available, the sparklyr connection logic is conceptually the same as in Section 30.7.4: **create a connection**, **inspect the configuration**, and **validate that Spark executors** are available. In an EMR setting, the details depend on whether we connect through YARN, through Livy, or through a configured Spark master endpoint.

## A Minimal Smoke Test Job and Clean Shutdown

Before running a costly job, it is good practice to run a small **smoke test**: we read a small dataset, do some aggregation, and plot a few graphs, and confirm that results can be collected.

When we are done, it is important to get into the habit of shutting down cleanly:

- **disconnect Spark sessions** from R so executors can be released;
- **terminate the EMR cluster** when it is no longer needed.

For instance, the following code should produce two charts if all components have been installed properly (see [28] for details).

```
library(sparklyr)
library(dplyr)

# Step 0: Connecting to a local cluster
sc <- spark_connect(master = "local")

# Step 1: Create a simple RDD equivalent by parallelizing a sequence
# Copy a small dataset to Spark for parallel processing
df <- copy_to(sc, data.frame(value = 1:10), "values", overwrite = TRUE)

# Step 2: Calculate the mean of the values in the dataset
mean_value <- df |> summarise(mean_value = mean(value)) |>
  collect() # Collect the result back to R

# Step 3: Visualize the simple dataset and its mean
simple_dataset <- data.frame(value = 1:10)
ggplot(simple_dataset, aes(x = value, y = value)) +
  geom_bar(stat = "identity", fill = "skyblue", alpha = 0.8) +
  geom_hline(yintercept = mean_value$mean_value, linetype = "dashed", color = "red") +
  labs(title = "Simple Dataset with Mean", x = "Index", y = "Values") +
  annotate("text", x = 5, y = mean_value$mean_value + 1, color = "red",
    label = paste("Mean:", round(mean_value$mean_value, 2)), fontface = "bold", size = 5) +
  theme_minimal() +
  theme(panel.border = element_rect(colour = "black", fill = NA, size = 1.5),
    axis.title.x = element_text(size = 16, face = "bold"),
    axis.title.y = element_text(size = 16, face = "bold"),
    axis.text.x = element_text(size = 14, face = "bold"),
    axis.text.y = element_text(size = 14, face = "bold"),
    plot.title = element_text(size = 18, face = "bold", hjust = 0.5))
```

```

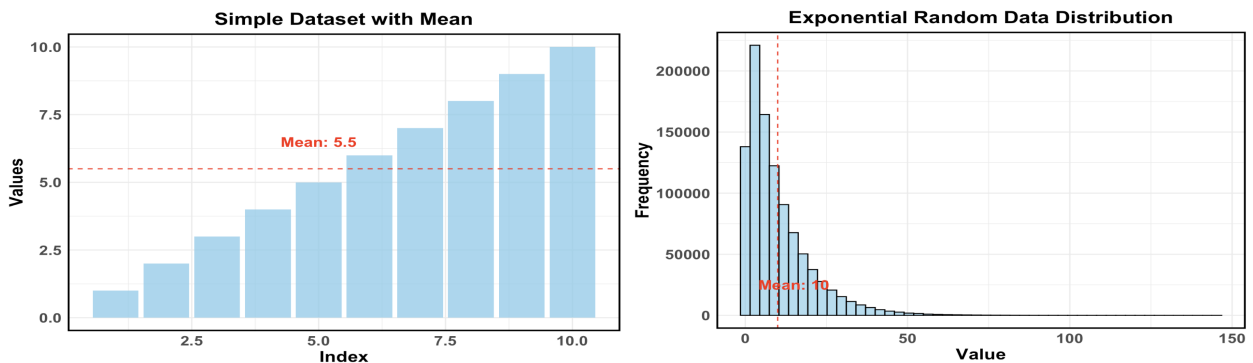
# Step 4: Generate random data in R
set.seed(123) # Set seed for reproducibility
# Generate 1 million random values
random_data <- data.frame(exponential = rexp(1000000, rate = 0.1))
spark_data <- copy_to(sc, random_data, "random_data", overwrite = TRUE) # Copy to Spark

# Step 5: Calculate the mean of the random data in Spark
mean_exponential <- spark_data |>
  summarise(mean_value = mean(exponential)) |>
  collect() # Collect the result back to R

# Step 6: Visualize the random data distribution and its mean
ggplot(random_data, aes(x = exponential)) +
  geom_histogram(bins = 50, fill = "skyblue", color = "black", alpha = 0.7) +
  geom_vline(xintercept = mean_exponential$mean_value, linetype = "dashed", color = "red") +
  labs(title = "Exponential Random Data Distribution", x = "Value", y = "Frequency") +
  annotate("text", x = mean_exponential$mean_value + 5, y = 25000, color = "red",
    label = paste("Mean:", round(mean_exponential$mean_value, 2)), fontface = "bold", size = 5) +
  theme_minimal() +
  theme(panel.border = element_rect(colour = "black", fill = NA, size = 1.5),
    axis.title.x = element_text(size = 16, face = "bold"),
    axis.title.y = element_text(size = 16, face = "bold"),
    axis.text.x = element_text(size = 14, face = "bold"),
    axis.text.y = element_text(size = 14, face = "bold"),
    plot.title = element_text(size = 18, face = "bold", hjust = 0.5))

# Step 7: Disconnect from Spark to free resources
spark_disconnect(sc)

```



Alternatively, we show how to do conduct a simple analysis using `sparklyr`, even though the linear regression problem below does not in any way warrant the use of distributed computing (we can easily do all computations on a local computer).

```

# Load necessary libraries
library(sparklyr)
library(dplyr)
library(ggplot2)

# Connect to Spark
sc <- spark_connect(master = "local")

```

```

# Load the iris dataset and copy to Spark
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_spark", overwrite = TRUE)

# Randomize and split the data into training and testing sets
iris_tbl <- iris_tbl |> sdf_sample(fraction = 1, replacement = FALSE, seed = 123)
train_test_split <- iris_tbl |> sdf_random_split(training = 0.7, testing = 0.3, seed = 123)
train <- train_test_split$training
test <- train_test_split$testing

# Assemble features for training and testing
feature_cols <- c("Sepal_Width", "Petal_Length", "Petal_Width")
train <- train |> ft_vector_assembler(input_cols = feature_cols, output_col = "features")
test <- test |> ft_vector_assembler(input_cols = feature_cols, output_col = "features")

# Train a linear regression model to predict Sepal_Length
model <- ml_linear_regression(train, label_col = "Sepal_Length", features_col = "features")

# Get predictions and evaluate the model
preds <- ml_predict(model, test)
metrics <- ml_regression_evaluator(preds, label_col = "Sepal_Length",
                                   prediction_col = "prediction", metric_name = "rmse")

# Print evaluation metrics
cat('RMSE:', metrics, '\n')

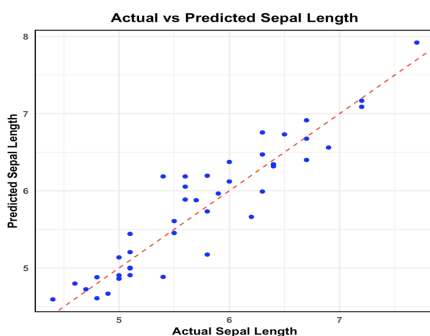
# Plotting the results
preds_r <- preds |> select(Sepal_Length, prediction) |> collect()

# Plot actual vs predicted values
ggplot(preds_r, aes(x = Sepal_Length, y = prediction)) +
  geom_point(color = 'blue') + theme_minimal() +
  geom_abline(intercept = 0, slope = 1, color = 'red', linetype = "dashed") +
  labs(title = 'Actual vs Predicted Sepal Length', x = 'Actual Sepal Length',
       y = 'Predicted Sepal Length') +
  theme(panel.border = element_rect(colour = "black", fill = NA, size = 1),
        plot.title = element_text(face = "bold", hjust = 0.5),
        axis.title = element_text(face = "bold"), axis.text = element_text(face = "bold"))

# Disconnect from Spark
spark_disconnect(sc)

```

RMSE: 0.2960172



## Cost Control Checklist

Cloud workflows are convenient, but they can become expensive **quickly** and **quietly**. We can avoid surprise bills by:

- enabling **auto-termination** (or enforcing teardown discipline) for test clusters;
- turning on **logging** for debugging and post-mortems;
- sizing instances to avoid chronic spilling and shuffle bottlenecks, while also avoiding persistent over-provisioning;
- using **small smoke tests** and **time-window prototypes** before scaling to full data;
- selecting Parquet (or another columnar format) over CSV for large analytical tables (see Section 30.7.3).

## 30.8 Machine Learning at Scale With PySpark

Many machine learning (ML) workflows have a repeated structure: we **scan a large dataset** to compute **simple summaries**, and then perform **more complex calculations** on the resulting **reduced objects**.

Spark is well suited to this pattern because it can express large passes over data as **distributed transformations**, while keeping intermediate state available for **iterative updates**.

In this section, we illustrate a selection of standard ML tasks in Spark. We begin with an example where the preprocessing is intentionally written in a MapReduce-like style, and then move toward the more typical modern workflow based on Spark's DataFrame API.<sup>82</sup>

82: In the rest of this chapter's examples we will instead use PySpark, which we assume has been [properly installed](#).

### 30.8.1 Multinomial Naïve Bayes

Spark can implement MapReduce-like operations efficiently, but it also supports higher-level APIs for end-to-end machine learning pipelines. In this first example, we intentionally emphasize the map and reduce steps to make the link to MapReduce explicit.

#### A Word on Parquet Files

For extremely large datasets, row-based file formats like CSV are often impractical. They require significant storage space, and are very slow to query. The examples below use **Parquet files**.<sup>83</sup>

83: The same data stored in a Parquet file will require about 90% less storage space and be about 30 times faster to query than a CSV file [42].

#### Multinomial Naive Bayes

As discussed in section [3, sec. 21.4.4], **multinomial naive Bayes** is a multi-class classification algorithm which assumes that each class's feature vectors have a multinomial distribution.<sup>84</sup>

Let's say that we have a dataset  $M$ , where each represents an email of  $n$  features. The features represent word counts for  $n$  particular terms that occur in the messages.

84: A common use case for this algorithm is **email spam filters**.

85: In practice we might use a binomial classifier for this problem, but for our purposes it is acceptable to use a more general multinomial classifier.

For the purposes of this example, we will assume that there are only two classes: “ham” (good emails) and “spam.”<sup>85</sup> Each email is represented by a feature vector  $\mathbf{x} = (x_1, \dots, x_n)$

In this example, we have  $K = 2$  categories. We will denote the categories as  $\{C_k \mid k = 0, 1\}$ .

**Objective:** Compute  $P(\mathbf{x} \in C_k \mid x_1, \dots, x_n)$  for  $k = 0, 1$ .

**Steps:**

1. Fix  $k$  according to **Bayes' Theorem**.

$$P(\mathbf{x} \in C_k \mid x_1, \dots, x_n) \propto P(C_k) \times P(x_1, \dots, x_n \mid \mathbf{x} \in C_k).$$

2. Our **naive** assumption is

$$P(x_1, \dots, x_n \mid \mathbf{x} \in C_k) = P(x_1 \mid \mathbf{x} \in C_k) \times \dots \times P(x_n \mid \mathbf{x} \in C_k);$$

then

$$P(\mathbf{x} \in C_k \mid x_1, \dots, x_n) \propto P(C_k) \prod_{i=1}^n P(x_i \mid \mathbf{x} \in C_k).$$

3. The **multinomial** assumption is

$$P(x_i \mid \mathbf{x} \in C_k) \propto p_{k,i}^{x_i}, \text{ where } p_{k,i} \in [0, 1] \text{ for all } i.$$

4. Considering all of our assumptions, the posterior probabilities can be expressed as

$$P(\mathbf{x} \in C_k \mid x_1, \dots, x_n) \propto P(C_k) \times \prod_{i=1}^n p_{k,i}^{x_i}.$$

5. We can then linearize the model by taking logarithms:

$$\log P(\mathbf{x} \in C_k \mid x_1, \dots, x_n) \propto b_k + \sum_{i=1}^n x_i \cdot \log p_{k,i}$$

6. We **train** the classifier by estimating  $p_{k,i}$  on a subset of rows, and specifying the prior probabilities  $b_k$ .
7. For example, a possible estimation scheme is

$$\hat{p}_{k,i} = \frac{\sum_{\mathbf{x} \in C_k} x_i + \alpha}{\sum_{\mathbf{x} \in C_k} \sum_{j=1}^n x_j + \alpha d}, \text{ where } 0 < \alpha \ll 1 \text{ and } 0 < d.$$

Estimating  $p_{k,i}$  becomes less trivial when  $M$  has a large number of rows.

To make parameter estimation less computationally demanding (and slow), we pre-process our data using MapReduce-like operations in Spark. For this example, we will use the UC Irvine Machine Learning Repository's *Spambase* dataset [19].

The next block of code illustrates how to scrape the data from UCIML and convert it to a Parquet file (`spambase_35.parquet`). In general, we will skip this step: all of this section's Parquet files can be found [here](#) ↗.

```

import urllib.request
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.ml.feature import VectorAssembler

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/spambase/spambase.data"

selected_features = {
    0: 'make', 1: 'address', 2: 'all', 4: 'our', 5: 'over', 6: 'remove', 7: 'internet',
    8: 'order', 9: 'mail', 10: 'receive', 11: 'will', 12: 'people', 13: 'report',
    14: 'addresses', 15: 'free', 16: 'business', 17: 'email', 18: 'you', 19: 'credit',
    20: 'your', 21: 'font', 23: 'money', 28: 'lab', 29: 'labs', 30: 'telnet', 32: 'data',
    35: 'technology', 37: 'parts', 39: 'direct', 41: 'meeting', 42: 'original', 43: 'project',
    45: 'edu', 46: 'table', 47: 'conference'
}

fids = sorted(selected_features.keys())
feature_names = [selected_features[i] for i in fids]

spark = SparkSession.builder.getOrCreate()

sc = spark.sparkContext

local_file = "/tmp/spambase.csv"
urllib.request.urlretrieve(url, local_file)

df_raw = spark.read.text(local_file)

cols = [f"c{i}" for i in range(57)] + ["label_raw"]

df = (df_raw
      .select(F.split(F.col("value"), ",").alias("parts"))
      .select([F.col("parts")[i].cast("double").alias(cols[i]) for i in range(58)]))

df = df.withColumn("label", F.col("label_raw").cast("int")).drop("label_raw")

df = df.select("label",
              *[F.col(f"c{i}").alias(selected_features[i]) for i in fids])

df = df.dropna()

assembler = VectorAssembler(inputCols=feature_names, outputCol="features")

out = assembler.transform(df)
out_compact = out.select("label", *feature_names, "features")

out_compact.write.mode("overwrite").parquet("spambase_35.parquet")

spark.stop()

```

First, we load a pre-filtered version of our dataset; these are the words we **want** to count. For the purposes of this example, we have selected more interpretable features, i.e., no numbers or punctuation characters. Once that is done, we initialize Spark.

```

from pyspark.sql import SparkSession # already loaded, but here as a reminder
from math import log
import numpy as np
from pyspark.mllib.classification import NaiveBayesModel
from pyspark.mllib.linalg import Vectors
from pyspark.mllib.evaluation import MulticlassMetrics

spark = SparkSession.builder.appName("naive_bayes_filtered").getOrCreate()
sc = spark.sparkContext

parquet_path = "spambase_35.parquet"
df = spark.read.parquet(parquet_path)

# need this later to see which words are associated with spam
feature_word = [c for c in df.columns if c not in ("label", "features")]

M = df.rdd.map(lambda r: (int(r["label"]), r["features"].toArray()))

```

Now that everything is set up, we can move to the “map” phase.

```

def get_feature_counts(label_and_feats):
    label, feats = label_and_feats
    return [(label, j), val) for j, val in enumerate(feats)]

mapped_M = M.flatMap(get_feature_counts)

```

Next, we have the “reduce” phase.

```

reduced_M = mapped_M.reduceByKey(lambda a, b: a + b)

```

The data is now ready for classification. We will do the actual classification shortly; for now we provide a quick sanity check and take a look at which features are most associated with spam emails.

```

label_counts = M.map(lambda x: (x[0], 1)).reduceByKey(lambda a, b: a + b)

feature_totals = dict(reduced_M.collect())
class_counts = dict(label_counts.collect())

n = len(M.first()[1])
spam_ratios = []
for j in range(n):
    spam_freq = feature_totals.get((1, j), 0.0) / class_counts[1]
    ham_freq = feature_totals.get((0, j), 0.0) / class_counts[0]
    if ham_freq > 0:
        spam_ratios.append((spam_freq / ham_freq, j, spam_freq, ham_freq))

top5 = sorted(spam_ratios, reverse=True)[:5]
print("Top 5 spam-associated words:")
for ratio, j, sf, hf in top5:
    print(f"{feature_word[j]}: {ratio:.3f}x ({sf:.3f} vs {hf:.3f})")

```

Top 5 spam-associated words:  
 remove: 29.351x (0.275 vs 0.009)  
 credit: 27.118x (0.206 vs 0.008)  
 addresses: 13.475x (0.112 vs 0.008)  
 money: 12.422x (0.213 vs 0.017)  
 free: 7.044x (0.518 vs 0.074)

Based on our real-world experiences with spam emails, do these words make sense? We can also check the class counts.

```
print(f"Class counts: {class_counts[0]} ham, {class_counts[1]} spam")
```

Class counts: 2788 ham, 1813 spam

We are now ready to build the classifier.

```
n = len(M.first()[1])
spam_ratios = []
for j in range(n):
    spam_freq = feature_totals.get((1, j), 0.0) / class_counts[1]
    ham_freq = feature_totals.get((0, j), 0.0) / class_counts[0]
    if ham_freq > 0:
        spam_ratios.append((spam_freq / ham_freq, j, spam_freq, ham_freq))

top5 = sorted(spam_ratios, reverse=True)[:5]
print("Top 5 spam-associated words:")
for ratio, j, sf, hf in top5:
    print(f"{feature_word[j]}: {ratio:.3f}x ({sf:.3f} vs {hf:.3f})")

print(f"Class counts: {class_counts[0]} ham, {class_counts[1]} spam")

# split into train and test
train_M, test_M = M.randomSplit([0.8, 0.2], seed=0)

# map / reduce on training data
mapped_train = train_M.flatMap(get_feature_counts)
reduced_train = mapped_train.reduceByKey(lambda a, b: a + b)

label_counts_train = train_M.map(lambda x: (x[0], 1)).reduceByKey(lambda a, b: a + b)

class_token_mass = (reduced_train
    .map(lambda kv: (kv[0][0], kv[1])) # drop feature index
    .reduceByKey(lambda a, b: a + b))

feature_totals_train = dict(reduced_train.collect())
class_counts_train = dict(label_counts_train.collect())
class_total_mass = dict(class_token_mass.collect())

classes = sorted(class_counts_train.keys())
N = sum(class_counts_train.values())
pi_arr = np.array([log(class_counts_train[c] / N) for c in classes], dtype=float)
```

```

alpha = 0.1
theta_rows = []
for c in classes:
    denom = class_total_mass.get(c, 0.0) + alpha * n
    row = []
    for j in range(n):
        num = feature_totals_train.get((c, j), 0.0) + alpha
        row.append(log(num / denom))
    theta_rows.append(row)
theta_arr = np.array(theta_rows, dtype=float)

# build the model
model = NaiveBayesModel(labels=np.array(classes), pi=pi_arr, theta=theta_arr)

# predictions on test set
preds_and_labels = test_M.map(
    lambda lp: (float(model.predict(Vectors.dense(lp[1]))), float(lp[0]))
)

metrics = MulticlassMetrics(preds_and_labels)

acc = metrics.accuracy
conf = metrics.confusionMatrix().toArray()

correct = int(conf.trace())
test_n = int(conf.sum())

print(f"\nTest accuracy: {acc:.3f} [{correct}/{test_n}]")
print("Confusion matrix (rows=true, cols=pred):")
for i, t in enumerate(classes):
    print(f"{t}: {[int(x) for x in conf[i]]}")

sc.stop()

```

```

Test accuracy: 0.873 [815/934]
Confusion matrix (rows=true, cols=pred):
0: [480, 83]
1: [36, 335]

```

We achieve 87.3% accuracy, which is ... not bad. The data in question dates back to 1998, however, and the nature of spam has probably changed quite a lot since then; all this to say that it might not be the best model to use for modern spam detection.

Additionally, we discussed the “wisdom” of using accuracy as a performance evaluation metric in [3, ch. 21]; remember that we are using it in this chapter mostly for pedagogical purposes.

The example above uses PySpark’s older RDD API (`pyspark.mllib`) to show more explicitly how MapReduce works. In practice, modern implementations typically use the the newer dataframe API (`pyspark.m`).

```

from pyspark.sql import SparkSession, functions as F
from pyspark.ml.classification import NaiveBayes
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

sc = SparkSession.builder.appName("naive_bayes_filtered_df").getOrCreate()

df = sc.read.parquet("spambase_35.parquet").select(
    F.col("label").cast("double").alias("label"),
    "features"
)

train_df, test_df = df.randomSplit([0.8, 0.2], seed=0)

model = NaiveBayes(
    featuresCol="features",
    labelCol="label",
    predictionCol="prediction",
    smoothing=0.1,
    modelType="multinomial"
).fit(train_df)

pred = model.transform(test_df).select("label", "prediction")

acc = MulticlassClassificationEvaluator(
    labelCol="label",
    predictionCol="prediction",
    metricName="accuracy"
).evaluate(pred)

conf = {
    (int(r.label), int(r.prediction)): r["count"]
    for r in pred.groupBy("label", "prediction").count().collect()
}

total = pred.count()
correct = conf.get((0, 0), 0) + conf.get((1, 1), 0)

print(f"Test accuracy: {acc:.3f} [{correct}/{total}]")
print("Confusion matrix (rows=true, cols=pred):")
for t in [0, 1]:
    print(f"{t}: {[conf.get((t, p), 0) for p in [0, 1]]}")

sc.stop()

```

```

Test accuracy: 0.863 [828/959]
Confusion matrix (rows=true, cols=pred):
0: [479, 100]
1: [31, 349]

```

### 30.8.2 *k*-Means and Initialization

Clustering is another setting where an iterative algorithm can be implemented using repeated MapReduce-like passes over the data. In Spark, these passes are expressed as transformations and actions over distributed datasets.

#### *k*-Means

As discussed in [3, sec. 22.2.1], *k*-means is a clustering algorithm that aims to “achieve minimal in-cluster variation”. For the purposes of this section, we will assume that the algorithm specifically aims to minimize the average **Euclidean** distance between points in the same cluster.

*k*-means has two major steps:<sup>86</sup>

1. **initialization**, where *k* centres are chosen, each corresponding to a cluster, and
2. **iterations**, where each point is assigned a cluster based on which centre it is closest to.

These steps are repeated “until the clusters are **stable**” [3, sec. 22.2.1].

This raises the question: how are the centroids initialized? Two popular initialization schemes are ***k*-means++** and ***k*-means||**.

*k*-means++ is the default choice for the `scikit-learn` library. To initialize with *k*-means++:

1. choose the first centre uniformly at random from the dataset;
2. select the next centre from the remaining points at random, but with each point *x* assigned a selection probability  $p_x$  proportional to its squared distance to the **closest-already-chosen centre**;
3. repeat until there are *k* centres.

In contrast to *k*-means++, *k*-means|| lends itself better to **parallelism**; it is the default initialization scheme used by PySpark, and proceeds as follows:

1. choose a set of points from the dataset; each point is selected with probability  $k p_x$ ;
2. repeat this process several times, on different cores if desired, to produce a set that contains far fewer points than the original dataset;<sup>87</sup>
3. select *k* centres from this set as described in the initialization step of *k*-means++, or standard *k*-means;
4. once the set of centres  $C = \{c_1, \dots, c_k\}$  has been initialized, the iterations proceed as follows:
  - a) assign each point to the nearest centre;
  - b) for all  $i \in \{1, \dots, k\}$ , let  $S_i$  be the set of points assigned to a particular centre  $c_i$ ;
  - c) repeat until the centroids stabilize.

*k*-means|| allows us to use MapReduce-like operations with PySpark:

- the **map** phase assigns points to the nearest centroid;

86: This is a brief overview for those who didn't read chapter 22.

87: But usually more than *k*.

- the **reduce** phase computes the sum of the feature vectors in each cluster and the number of points in each cluster;
- another map phase maps the reduced output to the new centroids by dividing the summed feature vectors by the number of points per cluster.

We can implement standard  $k$ -means and  $k$ -means|| on a synthetic expansion of the penguins dataset [20, 32] with 100000 observations.<sup>88</sup> All data has been pre-processed and standardized appropriately.

88: All features are included except that the year; the data was generated using synthpop's gaussian copula synthesizer

First, we initialize Spark and load the dataset.

```
from pyspark.sql import SparkSession
from pyspark.mllib.feature import StandardScaler
from pyspark.mllib.linalg import Vectors
import numpy as np

spark = SparkSession.builder.appName("kmeans_penguins_mapreduce").getOrCreate()

sc = spark.sparkContext

df_path = "penguins_5_synth_gc.parquet"
df = spark.read.parquet(df_path).select("species", "features")

M_r = df.rdd.map(lambda r: (int(r["species"]), r["features"].toArray()))

# standardize
d_cont = 4 # first 4 columns are continuous

X_cont = M_r.map(lambda lf: Vectors.dense(lf[1][:d_cont]))
model = StandardScaler(withMean=True, withStd=True).fit(X_cont)

mu = np.array(model.mean)
sd = np.array(model.std)
bc_mu, bc_sd = sc.broadcast(mu), sc.broadcast(sd)

# standardize numeric features
M = M_r.map(lambda lf: (
    lf[0],
    np.r_[ (np.asarray(lf[1], dtype=float)[:d_cont] - bc_mu.value) / bc_sd.value,
           np.asarray(lf[1], dtype=float)[d_cont:] ]
))
```

Next, we define functions to compute the nearest centroid, and perform  $k$ -means.

```
def nearest_id(x, centroids):
    best_j, best_d = None, float("inf")
    xv = Vectors.dense(x)
    for j, c in enumerate(centroids):
        d = Vectors.squared_distance(xv, Vectors.dense(c))
        if d < best_d:
            best_d, best_j = d, j
    return best_j
```

```

def kmeans(M, k=3, max_iters=50, tol=1e-6, seed=0):
    np.random.seed(seed)
    centroids = [np.array(c, dtype=float) for _, c in M.takeSample(False, k, seed)]
    prev_move = float("inf")

    for it in range(max_iters):
        broadcast_centroids = M.context.broadcast(centroids)
        mapped = M.map(lambda lf: (nearest_id(np.array(lf[1]), broadcast_centroids.value),
                                   (np.array(lf[1]), 1)))

        reduced = mapped.reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1]))
        mapped_new_centroids = reduced.mapValues(lambda sc: sc[0] / sc[1]).collectAsMap()

        k = len(centroids)
        new_centroids = [mapped_new_centroids.get(j, centroids[j]) for j in range(k)]

        move = sum(np.linalg.norm(nc - c) for nc, c in zip(new_centroids, centroids))
        print(f"iter {it}: centroid movement = {move:.6f}")

        centroids = new_centroids
        if move <= tol or abs(prev_move - move) <= 1e-12:
            break
        prev_move = move

    counts = (mapped.map(lambda kv: (kv[0], kv[1][1]))
              .reduceByKey(lambda a, b: a + b)
              .collectAsMap())

    return centroids, counts, mapped

```

We are now ready to train the model, with  $k = 3$  clusters, say.

```

k = 3
centroids_std, counts, mapped = kmeans(M, k=k, seed=0)
assignments = M.map(lambda lf: (nearest_id(lf[1], centroids_std), lf[0]))

print("\nCluster sizes:", dict(assignments.countByKey()))

```

```

iter 0: centroid movement = 2.779748
iter 1: centroid movement = 1.395443
iter 2: centroid movement = 1.138290
iter 3: centroid movement = 0.308572
iter 4: centroid movement = 0.138653
...
iter 12: centroid movement = 0.001587
iter 13: centroid movement = 0.001483
iter 14: centroid movement = 0.001252
iter 15: centroid movement = 0.000348
iter 16: centroid movement = 0.000304
iter 17: centroid movement = 0.000000

```

```
Cluster sizes: {0: 39944, 1: 24312, 2: 35744}
```

Now, let's see what the clusters look like.<sup>89</sup>

```
pair_counts = (assignments
               .map(lambda cl: ((cl[0], cl[1]), 1))
               .reduceByKey(lambda a, b: a + b))

by_cluster = (pair_counts
             .map(lambda kv: (kv[0][0], (kv[0][1], kv[1])))
             .groupByKey()
             .mapValues(lambda xs: {lab: int(n) for lab, n in xs})
             .collect())

names = {0: "Adelie", 1: "Chinstrap", 2: "Gentoo"}

for cluster_id, counts in sorted(by_cluster, key=lambda x: x[0]):
    total = sum(counts.values())
    purity = (max(counts.values()) / total) if total else 0.0
    pretty = {names.get(l, l): n for l, n in counts.items()}
    print(f"Cluster {int(cluster_id)}: {pretty} | purity = {purity:.3f}")

spark.stop()
```

```
Cluster 0: {'Adelie': 39732, 'Chinstrap': 212} | purity = 0.995
Cluster 1: {'Adelie': 4112, 'Chinstrap': 20200} | purity = 0.831
Cluster 2: {'Gentoo': 35736, 'Chinstrap': 8} | purity = 1.000
```

The clusters certainly seem to be “pure”.

The example above is a manual implementation of standard  $k$ -means that explicitly shows how the map and reduce steps work.

Alternatively, we can use PySpark's built-in  $k$ -means model with  $k$ -means++ initialization. Unlike the first example, which used Spark's RDD API, this implementation uses the newer data frame API.<sup>90</sup>

This implementation is a simple **pipeline** that standardizes the data and then performs  $k$ -means.

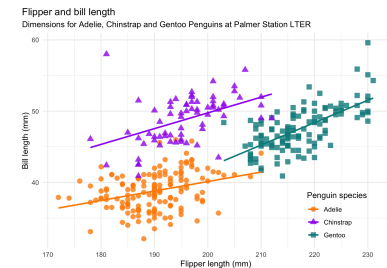
```
from pyspark.sql import SparkSession
from pyspark.ml.clustering import KMeans
from pyspark.ml.feature import VectorAssembler, StandardScaler
from pyspark.ml import Pipeline

spark = SparkSession.builder.getOrCreate()
df_path = "penguins_5_synth_gc.parquet"

# select features by name so we can standardize
df = spark.read.parquet(df_path).select("species", "sex_bin",
    "bill_length_mm", "bill_depth_mm",
    "flipper_length_mm", "body_mass_g")

numeric = ["bill_length_mm", "bill_depth_mm",
    "flipper_length_mm", "body_mass_g"]
```

89: In the original dataset, there are 3 types of penguins: Adelie, Chinstrap, and Gentoo. The scatter plot of flipper lengths vs. bill lengths is shown below [20]:



90: Using the newer API does away with some of the technicalities involved in manual RDD management.

```
# select numeric features
num_assembler = VectorAssembler(
    inputCols=numeric,
    outputCol="cont_vec")

# scale numeric features
scaler = StandardScaler(
    inputCol="cont_vec", outputCol="cont_z",
    withMean=True, withStd=True)

# add sex
assembler = VectorAssembler(
    inputCols=["cont_z", "sex_bin"],
    outputCol="features_std")

k = 3
kmeans = KMeans(
    k=k, seed=0, maxIter=50, featuresCol="features_std",
    predictionCol="prediction", initMode="k-means||",
    initSteps=8, tol=1e-6)

pipeline = Pipeline(stages=[num_assembler, scaler,
                             assembler, kmeans])

model = pipeline.fit(df)
predictions = model.transform(df)

counts = (predictions
    .groupBy("prediction", "species")
    .count())

rows = counts.collect()
by_cluster = {}
for r in rows:
    by_cluster.setdefault(int(r["prediction"]), {})
        [int(r["species"])] = int(r["count"])

names = {0: "Adelie", 1: "Chinstrap", 2: "Gentoo"}

for cid in sorted(by_cluster.keys()):
    d = by_cluster[cid]
    total = sum(d.values())
    purity = (max(d.values()) / total) if total else 0.0
    pretty = {names.get(l, l): n for l, n in d.items()}
    print(f"Cluster {cid}: {pretty} | purity = {purity:.3f}")

spark.stop()
```

```
Cluster 0: {'Chinstrap': 8, 'Gentoo': 35736} | purity = 1.000
Cluster 1: {'Adelie': 39732, 'Chinstrap': 212} | purity = 0.995
Cluster 2: {'Chinstrap': 20200, 'Adelie': 4112} | purity = 0.831
```

### 30.8.3 Streaming $k$ -Means as an Online Extension

In some applications, we want to update a model as new observations arrive, as we discussed in [3, ch. 28]. A streaming variant of  $k$ -means can update centroids incrementally using MapReduce-like passes over each incoming batch.

#### Streaming $k$ -Means

In practice, it is often useful to incorporate new data into a model as it arrives. **Streaming  $k$ -means** is an **online** extension of traditional  $k$ -means.

Consider  $k$  clusters. For each centroid  $c_i$ , let  $x_i$  be the **vector sum** of the new points that are closest to  $c_i$  relative to the other existing centroids, and let  $n_i$  be the number of such new points. Streaming  $k$ -means uses these batch summaries to update the existing centroids.

Let  $m_i$  be the number of points previously assigned to centroid  $c_i$ , and retain the notation above for  $n_i$  and  $x_i$ . The updates at time  $t + 1$  are

$$c_i(t + 1) = \frac{\alpha c_i(t) m_i(t) + x_i(t)}{\alpha m_i(t) + n_i(t)}$$

$$m_i(t + 1) = n_i(t) + m_i(t)$$

where  $\alpha$  is a hyperparameter that controls the **decay** in the weight of older data.

As with standard  $k$ -means, MapReduce-like operations are performed:

- the **map** phase assigns each point to the nearest centroid, and
- for each cluster, the **reduce** phase computes  $x_i$  and  $n_i$ .

To illustrate, we perform streaming  $k$ -means on the penguins dataset. As a proxy for “online” data, we split the dataset into **batches** and add one batch for each **update**.

```
from pyspark.sql import SparkSession
from pyspark.mllib.feature import StandardScaler
from pyspark.mllib.linalg import Vectors
import numpy as np

spark = SparkSession.builder.getOrCreate()
sc = spark.sparkContext

df_path = "penguins_5_synth_gc.parquet"
df = spark.read.parquet(df_path).select("species", "features")

M_r = df.rdd.map(lambda r: (int(r["species"]), r["features"].toArray()))

# standardize
d_cont = 4 # first 4 columns are continuous

X_cont = M_r.map(lambda lf: Vectors.dense(lf[1][:d_cont]))
model = StandardScaler(withMean=True, withStd=True).fit(X_cont)
```

```

mu = np.array(model.mean)
sd = np.array(model.std)
bc_mu, bc_sd = sc.broadcast(mu), sc.broadcast(sd)

# standardize numeric features
M = M_r.map(lambda lf: (
    lf[0],
    np.r_[ (np.asarray(lf[1], dtype=float)[:d_cont] - bc_mu.value) / bc_sd.value,
           np.asarray(lf[1], dtype=float)[d_cont:] ]
))

def nearest_id(x, centroids):
    best_j, best_d = None, float("inf")
    xv = Vectors.dense(x)
    for j, c in enumerate(centroids):
        d = Vectors.squared_distance(xv, Vectors.dense(c))
        if d < best_d:
            best_d, best_j = d, j
    return best_j

def get_batches(M, num_batches):
    M_indexed = M.zipWithIndex().cache()
    return [
        M_indexed.filter(lambda kv: kv[1] % num_batches == i).map(lambda kv: kv[0]).cache()
        for i in range(num_batches)
    ]

batches = get_batches(M, num_batches=5) # 5 batches

```

Now we can cluster the data, as follows.

```

def streaming_kmeans(batches, k=3, alpha=1, seed=0):
    np.random.seed(seed)

    init_batch = None
    for b in batches:
        if not b.isEmpty():
            init_batch = b
            break
    if init_batch is None:
        return [], {}, None

    centroids = [np.array(c, dtype=float) for _, c in init_batch.takeSample(False, k, seed)]
    n = [0.0 for _ in range(k)]
    mapped_last = None

    for t, batch in enumerate(batches):
        if batch.isEmpty():
            print(f"batch {t}: (empty)")
            continue

        broadcast_centroids = batch.context.broadcast(centroids)

```

```

mapped = batch.map(
    lambda lf: (nearest_id(np.array(lf[1]), broadcast_centroids.value),
               (np.array(lf[1]), 1)))
mapped_last = mapped

reduced = mapped.reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1]))
stats = reduced.collectAsMap()

for j in range(k):
    n_old = alpha * n[j]
    if j in stats:
        s_j, m_j = stats[j]
        x_j = s_j / m_j
        denom = n_old + float(m_j)
        centroids[j] = (n_old * centroids[j] + float(m_j) * x_j) / denom
        n[j] = denom
    else:
        n[j] = n_old

counts_batch = {j: int(stats[j][1]) for j in stats}
counts_pretty = {j: counts_batch.get(j, 0) for j in range(k)}
print(f"batch {t}: actual points = {counts_pretty}")

final_counts = (
    mapped_last.map(lambda kv: (kv[0], kv[1][1]))
    .reduceByKey(lambda a, b: a + b)
    .collectAsMap()
) if mapped_last is not None else {}

return centroids, final_counts, mapped_last

k = 3
centroids_stream, counts_stream, mapped_last = streaming_kmeans(batches, k=k)

assignments = M.map(lambda lf: (nearest_id(lf[1], centroids_stream), lf[0]))

print("\nCluster sizes:", dict(assignments.countByKey()))

```

```

batch 0: actual points = {0: 7002, 1: 7038, 2: 5960}
batch 1: actual points = {0: 7551, 1: 7149, 2: 5300}
batch 2: actual points = {0: 7660, 1: 7150, 2: 5190}
batch 3: actual points = {0: 7751, 1: 7148, 2: 5101}
batch 4: actual points = {0: 7734, 1: 7148, 2: 5118}

```

```
Cluster sizes: {0: 38885, 2: 25369, 1: 35746}
```

Let us see how pure these clusters are compared to standard  $k$ -means.

```

pair_counts = (
    assignments.map(lambda cl: ((cl[0], cl[1]), 1))
    .reduceByKey(lambda a, b: a + b)
)

```

```

by_cluster = (
    pair_counts
    .map(lambda kv: (kv[0][0], (kv[0][1], kv[1])))
    .groupByKey()
    .mapValues(lambda xs: {lab: int(n) for lab, n in xs})
    .collect()
)

names = {0: "Adelie", 1: "Chinstrap", 2: "Gentoo"}

for cluster_id, counts in sorted(by_cluster, key=lambda x: x[0]):
    total = sum(counts.values())
    purity = (max(counts.values()) / total) if total else 0.0
    pretty = {names.get(l, l): n for l, n in counts.items()}
    print(f"Cluster {int(cluster_id)}: {pretty} | purity = {purity:.3f}")

spark.stop()

```

```

Cluster 0: {'Adelie': 38722, 'Chinstrap': 163} | purity = 0.996
Cluster 1: {'Chinstrap': 10, 'Gentoo': 35736} | purity = 1.000
Cluster 2: {'Adelie': 5122, 'Chinstrap': 20247} | purity = 0.798

```

Overall, the results are quite similar to standard  $k$ -means, with slightly lower purity in the chinstrap cluster.

### 30.8.4 Linear Regression and Regularization

The previous examples emphasized explicit map and reduce steps. In practice, Spark is typically used through **higher-level** estimators and pipelines, such as those used for **linear regression** and **regularization**.<sup>91</sup>

91: See [3, ch. 20, 21] for theoretical details.

#### Linear Regression

Returning to the synthetic penguins dataset, we can use PySpark to fit a linear regression model for predicting bill length.

```

from pyspark.sql import SparkSession
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler, StandardScaler
from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator

spark = SparkSession.builder.appName("penguins_bill_length").getOrCreate()

path = "penguins_5_synth_gc.parquet"
df = spark.read.parquet(path)

label = "bill_length_mm"

```

```

numeric = ["bill_depth_mm", "flipper_length_mm", "body_mass_g"]
binary = ["sex_bin"]

df = df.withColumnRenamed(label, "label")

num_assembler = VectorAssembler(inputCols=numeric, outputCol="num_vec")

scaler = StandardScaler(inputCol="num_vec", outputCol="num_z", withMean=True, withStd=True)

final_assembler = VectorAssembler(inputCols=["num_z"] + binary, outputCol="features_lr")

# ols
lr = LinearRegression(
    featuresCol="features_lr",
    labelCol="label",
    elasticNetParam=0,
    maxIter=100
)

# pipeline - regularization plus regression
pipeline = Pipeline(stages=[num_assembler, scaler, final_assembler, lr])

train, test = df.randomSplit([0.8, 0.2], seed=0)

model = pipeline.fit(train)
pred = model.transform(test)
model_info = model.stages[-1]

print("Intercept:", model_info.intercept)
print("Coefficients:", model_info.coefficients)

rmse = RegressionEvaluator(metricName="rmse").evaluate(pred)

r2 = RegressionEvaluator(metricName="r2").evaluate(pred)
print("\nTest RMSE:", rmse)
print("Test R2:", r2)

spark.stop()

```

Intercept: 44.13553683699485

Coefficients: [0.675534980605088,0.7194021479518126,4.3293127870813395, -0.6523403160845529]

Test RMSE: 3.0228481009492545

Test R2: 0.6899589968877013

On this dataset, the model explains just under 70% of the variation in bill length. We can also use various **regularization schemes**: ridge regression, elastic net, lasso.

```

# ridge regression
lr = LinearRegression(
  featuresCol="features_lr",
  regParam = 0.1,
  elasticNetParam= 0,
  labelCol="label",
  maxIter=100
)

# pipeline - regularization plus regression
pipeline = Pipeline(stages=[num_assembler, scaler, final_assembler, lr])

train, test = df.randomSplit([0.8, 0.2], seed=0)

model = pipeline.fit(train)
pred = model.transform(test)
model_info = model.stages[-1]

print("Intercept:", model_info.intercept)
print("Coefficients:", model_info.coefficients)

rmse = RegressionEvaluator(metricName="rmse").evaluate(pred)
r2 = RegressionEvaluator(metricName="r2").evaluate(pred)

print("\nTest RMSE:", rmse)
print("Test R2:", r2)

spark.stop()

```

Intercept: 44.11261659807559  
 Coefficients: [0.6496771799813765,0.7992447366024067,4.163450005173538,-0.5404987255272939]

Test RMSE: 3.024686338708325  
 Test R2: 0.6895818013748536

```

# elastic net
lr = LinearRegression(
  featuresCol="features_lr",
  regParam = 0.1,
  elasticNetParam = 0.5,
  labelCol="label",
  maxIter=100)

# pipeline - regularization plus regression
pipeline = Pipeline(stages=[num_assembler, scaler, final_assembler, lr])

train, test = df.randomSplit([0.8, 0.2], seed=0)

model = pipeline.fit(train)
pred = model.transform(test)
model_info = model.stages[-1]

```

```

print("Intercept:", model_info.intercept)
print("Coefficients:", model_info.coefficients)

rmse = RegressionEvaluator(metricName="rmse").evaluate(pred)
r2 = RegressionEvaluator(metricName="r2").evaluate(pred)

print("\nTest RMSE:", rmse)
print("Test R2:", r2)

spark.stop()

```

Intercept: 44.05343512731549  
Coefficients: [0.49417026722427887,0.5881101024695853,4.20796639339711,-0.2517168395859838]

Test RMSE: 3.027875641434223  
Test R2: 0.6889268312751974

```

# lasso
lr = LinearRegression(
    featuresCol="features_lr",
    regParam = 0.1,
    elasticNetParam= 1,
    labelCol="label",
    maxIter=100
)

# pipeline - regularization plus regression
pipeline = Pipeline(stages=[num_assembler, scaler, final_assembler, lr])

train, test = df.randomSplit([0.8, 0.2], seed=0)

model = pipeline.fit(train)
pred = model.transform(test)
model_info = model.stages[-1]

print("Intercept:", model_info.intercept)
print("Coefficients:", model_info.coefficients)

rmse = RegressionEvaluator(metricName="rmse").evaluate(pred)
r2 = RegressionEvaluator(metricName="r2").evaluate(pred)

print("\nTest RMSE:", rmse)
print("Test R2:", r2)

spark.stop()

```

Intercept: 44.00184957604994  
Coefficients: [0.33493508717307485,0.36543551029964827,4.267227872797717,0.0]

Test RMSE: 3.0373391297060977  
Test R2: 0.6869793023485335

### 30.8.5 Tree-Based Methods

92: See [3, sec. 21.4.1].

**Tree-based methods** usually require little **feature scaling** and can perform well on heterogeneous inputs after basic preprocessing.<sup>92</sup> In this section, we illustrate a workflow for a mixed categorical and numeric dataset, including building a Parquet file suitable for repeated modeling.

#### Tree-Based Methods

The following examples compare the performance of a single **decision tree**, a **random forest**, and **gradient boosting** on the *Secondary Mushroom* dataset [40]. First, we fetch the data and generate a Parquet file.

```
import pandas as pd
from ucimlrepo import fetch_ucirepo

from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler

label = "class"
numeric = ["cap-diameter", "stem-height", "stem-width"]
categorical = ["cap-shape", "cap-surface", "cap-color", "does-bruise-or-bleed",
              "gill-attachment", "gill-spacing", "gill-color", "stem-root", "stem-surface",
              "stem-color", "veil-type", "veil-color", "has-ring", "ring-type",
              "spore-print-color", "habitat", "season"]

# fetch data from UCI
secondary_mushroom = fetch_ucirepo(id=848)

X = secondary_mushroom.data.features
y = secondary_mushroom.data.targets

# pandas dataframe
df_pd = pd.concat([y, X], axis=1)

spark = SparkSession.builder.getOrCreate()

# spark dataframe
df = spark.createDataFrame(df_pd)
```

In Spark, before we can **one-hot encode** our categorical variables, we convert them to integers using indexers.

```
# re-code labels
df = df.withColumn(
    "label",
    F.when(F.col("class") == "e", 0.0) # edible
    .when(F.col("class") == "p", 1.0) # poisonous
    .otherwise(None))
```

```

indexers = [
    StringIndexer(
        inputCol=c,
        outputCol=f"{c}_idx",
        handleInvalid="keep"
    )
    for c in categorical
]

encoder = OneHotEncoder(
    inputCols=[f"{c}_idx" for c in categorical],
    outputCols=[f"{c}_oh" for c in categorical],
    handleInvalid="keep"
)

```

Now we are ready to build the pipeline and save the Parquet file.

```

assembler = VectorAssembler(
    inputCols=numeric + [f"{c}_oh" for c in categorical],
    outputCol="features"
)

pipe = Pipeline(stages=indexers + [encoder, assembler])
model = pipe.fit(df)
out = model.transform(df)

out_compact = out.select("label", *numeric, *categorical, "features")

out_compact.write.mode("overwrite").parquet("mushrooms_secondary.parquet")
spark.stop()

```

Recall that standardization is typically unnecessary for tree-based methods, so our pipeline has only one modeling stage (the model itself).

We start with a single tree.

```

from pyspark.sql import SparkSession
from pyspark.ml import Pipeline
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.evaluation import BinaryClassificationEvaluator,
    MulticlassClassificationEvaluator

spark = SparkSession.builder.getOrCreate()

df_path = "mushrooms_secondary.parquet"
df = spark.read.parquet(df_path).select("label", "features")

train, test = df.randomSplit([0.8, 0.2], seed=0)

dt = DecisionTreeClassifier(
    labelCol="label",
    featuresCol="features",
    seed=0)

```

```

pipeline = Pipeline(stages=[dt])

model = pipeline.fit(train)

predictions = model.transform(test)

evaluator_auc = BinaryClassificationEvaluator(metricName="areaUnderROC")
auc = evaluator_auc.evaluate(predictions)
print(f"\nAUC-ROC = {auc:.6f}")

evaluator = MulticlassClassificationEvaluator()
f1 = evaluator.evaluate(predictions, {evaluator.metricName: "f1"})
print(f'F1: {f1:.6f}')

accuracy = evaluator.evaluate(predictions, {evaluator.metricName: "accuracy",
    evaluator.metricLabel:1})
print(f'Accuracy: {accuracy:.6f}')

conf = {
    (int(r.label), int(r.prediction)): r["count"]
    for r in predictions.groupBy("label", "prediction").count().collect()
}

print("\nConfusion matrix (rows=true, cols=pred):")
for t in [0, 1]:
    print(f"{t}: {[conf.get((t, p), 0) for p in [0, 1]]}")
spark.stop()

```

```

AUC-ROC = 0.620548
F1: 0.733512
Accuracy: 0.737399

```

```

Confusion matrix (rows=true, cols=pred):
0: [5019, 483]
1: [2742, 4037]

```

While risk tolerance is very personal, most people would probably not trust this model to tell them which mushrooms are safe to eat. Is a random forest any better?

```

from pyspark.sql import SparkSession
from pyspark.ml import Pipeline
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.evaluation import BinaryClassificationEvaluator,
    MulticlassClassificationEvaluator

spark = SparkSession.builder.getOrCreate()

df_path = "mushrooms_secondary.parquet"
df = spark.read.parquet(df_path).select("label", "features")

train, test = df.randomSplit([0.8, 0.2], seed=0)

```

```

dt = DecisionTreeClassifier(
    labelCol="label",
    featuresCol="features",
    seed=0
)

pipeline = Pipeline(stages=[dt])
model = pipeline.fit(train)
predictions = model.transform(test)

evaluator_auc = BinaryClassificationEvaluator(metricName="areaUnderROC")
auc = evaluator_auc.evaluate(predictions)
print(f"\nAUC-ROC = {auc:.6f}")

evaluator = MulticlassClassificationEvaluator()
f1 = evaluator.evaluate(predictions, {evaluator.metricName: "f1"})
print(f'F1: {f1:.6f}')

accuracy = evaluator.evaluate(predictions, {evaluator.metricName: "accuracy"})
print(f'Accuracy: {accuracy:.6f}')

conf = {
    (int(r.label), int(r.prediction)): r["count"]
    for r in predictions.groupBy("label", "prediction").count().collect()
}

print("\nConfusion matrix (rows=true, cols=pred):")
for t in [0, 1]:
    print(f"{t}: {[conf.get((t, p), 0) for p in [0, 1]]}")
spark.stop()

```

AUC-ROC = 0.881035

F1: 0.778471

Accuracy: 0.781207

Confusion matrix (rows=true, cols=pred):

0: [3730, 1772]

1: [915, 5864]

Compared to a single tree, the random forest is more likely to misclassify edible mushrooms, but less likely to misclassify poisonous mushrooms. The AUC is also much higher.

Finally, we compare to gradient-boosted trees.

```

from pyspark.sql import SparkSession
from pyspark.ml import Pipeline
from pyspark.ml.classification import GBTClassifier
from pyspark.ml.evaluation import BinaryClassificationEvaluator,
    MulticlassClassificationEvaluator

spark = SparkSession.builder.appName("gbt_mushroom").getOrCreate()

```

```

df_path = "mushrooms_secondary.parquet"
df = spark.read.parquet(df_path).select("label", "features")

train, test = df.randomSplit([0.8, 0.2], seed=0)

gbt = GBClassifier(labelCol="label", featuresCol="features", maxIter=10, seed=0)

pipeline = Pipeline(stages=[gbt])
model = pipeline.fit(train)
predictions = model.transform(test)

evaluator_auc = BinaryClassificationEvaluator(metricName="areaUnderROC")
auc = evaluator_auc.evaluate(predictions)
print(f"\nAUC-ROC = {auc:.6f}")

evaluator = MulticlassClassificationEvaluator()
f1 = evaluator.evaluate(predictions, {evaluator.metricName: "f1"})
print(f'F1: {f1:.6f}')

accuracy = evaluator.evaluate(predictions, {evaluator.metricName: "accuracy"})
print(f'Accuracy: {accuracy:.6f}')

conf = {
    (int(r.label), int(r.prediction)): r["count"]
    for r in predictions.groupBy("label", "prediction").count().collect()
}

print("\nConfusion matrix (rows=true, cols=pred):")
for t in [0, 1]:
    print(f"{t}: {[conf.get((t, p), 0) for p in [0, 1]]}")
spark.stop()

```

AUC-ROC = 0.969121

F1: 0.928007

Accuracy: 0.928100

Confusion matrix (rows=true, cols=pred):

0: [4992, 510]

1: [373, 6406]

### 30.8.6 Handling Class Imbalance

Severe **class imbalance** is common in applied settings, especially in clinical, operational, and anomaly detection datasets. One practical approach is to construct a synthetic training set that oversamples the minority class using **SMOTE**-style procedures,<sup>93</sup> while keeping evaluation on a separate test set that reflects the original population imbalance.

We work with the *Sepsis Survival Minimal Clinical Records* dataset [8]. This dataset has 102212 instances of patients who survived sepsis, and 8129 instances of patients who did not.

93: See [3, sec. 21.3].

```

import pandas as pd
from ucimlrepo import fetch_ucirepo
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler
from imblearn.over_sampling import BorderlineSMOTE
from sklearn.model_selection import train_test_split

target = "hospital_outcome_1alive_0dead"
continuous_preds = ["age_years", "episode_number"]
bin_preds = ["sex_0male_1female"]

sepsis = fetch_ucirepo(id=827)

X = sepsis.data.features
y = sepsis.data.targets

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0, stratify=y)

# standardize Tr data and apply the same transformation to Te data (avoid leakage)
mean = X_train[continuous_preds].mean()
std = X_train[continuous_preds].std(ddof=0) # population-level

X_train[continuous_preds] = (X_train[continuous_preds] - mean) / std
X_test[continuous_preds] = (X_test[continuous_preds] - mean) / std

# SMOTE on training data ONLY
sm = BorderlineSMOTE(k_neighbors=2, m_neighbors=5, random_state=0)
X_train, y_train = sm.fit_resample(X_train, y_train)

df_train_r = pd.concat([y_train, X_train], axis=1)
df_test_r = pd.concat([y_test, X_test], axis=1)

spark = SparkSession.builder.appName("sepsis_parquet").getOrCreate()

df_train = spark.createDataFrame(df_train_r)
df_test = spark.createDataFrame(df_test_r)

keep_cols = [target] + continuous_preds + bin_preds
df_train = df_train.select(*keep_cols)
df_test = df_test.select(*keep_cols)

df_train = df_train.dropna(subset=keep_cols)
df_test = df_test.dropna(subset=keep_cols)

num_assembler = VectorAssembler(
    inputCols=continuous_preds,
    outputCol="cont_vec"
)

```

```
assembler = VectorAssembler(  
    inputCols=["cont_vec"] + bin_preds,  
    outputCol="features"  
)  
  
pipe = Pipeline(stages=[num_assembler, assembler])  
pipe_model = pipe.fit(df_train)  
  
train_out = pipe_model.transform(df_train).select(  
    F.col(target).alias("label"),  
    "features"  
)  
  
test_out = pipe_model.transform(df_test).select(  
    F.col(target).alias("label"),  
    "features"  
)  
  
train_out.write.mode("overwrite").parquet("sepsis_train.parquet")  
test_out.write.mode("overwrite").parquet("sepsis_test.parquet")  
  
spark.stop()
```

Now that we have a balanced synthetic training set, we can compare the performance of Spark's logistic regression and multilayer perceptron implementations.

First, the logistic regression model.

```
from pyspark.sql import SparkSession  
from pyspark.ml.classification import LogisticRegression  
from pyspark.ml.evaluation import BinaryClassificationEvaluator # for AUC  
from pyspark.ml.evaluation import MulticlassClassificationEvaluator # for  
  
spark = SparkSession.builder.appName("log_reg_sepsis").getOrCreate()  
  
train_path = "sepsis_train.parquet"  
train = spark.read.parquet(train_path).select("label", "features")  
  
test_path = "sepsis_test.parquet"  
test = spark.read.parquet(test_path).select("label", "features")  
  
lr = LogisticRegression(maxIter=500)  
  
model = lr.fit(train)  
  
predictions = model.transform(test)  
  
evaluator_auc = BinaryClassificationEvaluator(metricName="areaUnderROC")  
auc = evaluator_auc.evaluate(predictions)  
print(f"\nAUC-ROC = {auc:.6f}")
```

```

evaluator = MulticlassClassificationEvaluator()
f1 = evaluator.evaluate(predictions, {evaluator.metricName: "f1"})
print(f'F1: {f1:.6f}')

accuracy = evaluator.evaluate(predictions,
    {evaluator.metricName: "accuracy", evaluator.metricLabel:1})
print(f'Accuracy: {accuracy:.6f}')

conf = {
    (int(r.label), int(r.prediction)): r["count"]
    for r in predictions.groupBy("label", "prediction").count().collect()
}

print("\nConfusion matrix (rows=true, cols=pred):")
for t in [0, 1]:
    print(f"{t}: {[conf.get((t, p), 0) for p in [0, 1]]}")

spark.stop()

```

AUC-ROC = 0.700602

F1: 0.706717

Accuracy: 0.615775

Confusion matrix (rows=true, cols=pred):

0: [1661, 778]

1: [11941, 18723]

Remember that the test data is not synthetic, and is subject to the original class imbalances. This explains the relatively low accuracy alongside stronger  $F_1$  and AUC scores.

How does a multilayer perceptron classifier compare?

```

from pyspark.sql import SparkSession
from pyspark.ml.classification import MultilayerPerceptronClassifier
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

spark = SparkSession.builder.appName("nn_sepsis").getOrCreate()

train_path = "sepsis_train.parquet"
train = spark.read.parquet(train_path).select("label", "features")

test_path = "sepsis_test.parquet"
test = spark.read.parquet(test_path).select("label", "features")

mlp = MultilayerPerceptronClassifier(layers=[3, 8, 2], seed=0,maxIter=500, stepSize=0.01)

model = mlp.fit(train)

predictions = model.transform(test)

```

```

evaluator_auc = BinaryClassificationEvaluator(metricName="areaUnderROC")
auc = evaluator_auc.evaluate(predictions)
print(f"\nAUC-ROC = {auc:.6f}")

evaluator = MulticlassClassificationEvaluator()
f1 = evaluator.evaluate(predictions, {evaluator.metricName: "f1"})
print(f'F1: {f1:.6f}')

accuracy = evaluator.evaluate(predictions,
    {evaluator.metricName: "accuracy", evaluator.metricLabel:1})
print(f'Accuracy: {accuracy:.6f}')

conf = {
    (int(r.label), int(r.prediction)): r["count"]
    for r in predictions.groupBy("label", "prediction").count().collect()
}

print("\nConfusion matrix (rows=true, cols=pred):")
for t in [0, 1]:
    print(f"{t}: {[conf.get((t, p), 0) for p in [0, 1]]}")

spark.stop()

```

AUC-ROC = 0.687611

F1: 0.773285

Accuracy: 0.704196

Confusion matrix (rows=true, cols=pred):

0: [1319, 1120]

1: [8672, 21992]

## 30.9 Exercises

- Consider each of the following scenarios and identify which of the 5 Vs are the dominant pressures. For each scenario, justify the choice and name one concrete workflow implication (storage format, sampling, validation, tooling choice, etc.).
  - A national pharmacy chain logs every transaction, but one region changes its product codes every six months.
  - A ride-sharing platform needs to update surge-pricing features every 2 minutes.
  - A hospital consortium merges patient records from 12 hospitals, each using slightly different schema conventions.
  - A marketing team runs 50,000 A/B tests per year on small changes to an app interface.
- Give two examples of datasets that are large in row count but not “big” for analysis, and two examples of datasets that are “big” despite having a modest number of rows. For each example, identify the binding constraint (RAM, I/O, shuffles, organizational cost, data quality, etc.).

3. We are given a pipeline that (i) reads a 200 GB CSV file from disk, (ii) filters 5% of the rows, (iii) performs a group-by aggregation on a single key, and (iv) writes a 5 GB result back to disk.
  - a) Explain why this can still be I/O-dominated even if the group-by arithmetic is simple.
  - b) List three changes that would reduce runtime without changing the question being answered.
  - c) List one change that might make the job faster but also changes the question (explain why).
4. Suppose we test  $m$  unrelated hypotheses at level  $\alpha = 0.05$ .
  - a) Compute  $1 - (1 - \alpha)^m$  for  $m \in \{10, 100, 1000\}$ .
  - b) Find the smallest  $m$  such that the probability of at least one false positive exceeds 0.95.
  - c) Briefly explain how this connects to both spurious correlations and  $p$ -hacking in big data settings.
5. Construct a simple example where an effect is statistically significant at level 0.05 but practically irrelevant.
  - a) Specify the effect size we might consider practically irrelevant.
  - b) Specify a sample size  $n$  large enough that a classical test would likely reject.
  - c) Explain why this is a recurring failure mode in large observational datasets.
6. For each of the following, decide whether it is primarily covariate shift, label shift, or concept drift, and provide an explanation.
  - a) A fraud model is deployed; the ranking quality remains similar but the base rate of fraud doubles.
  - b) A hospital introduces a new screening device that changes which patients get tested.
  - c) A platform changes the definition of “active user” from “any login” to “10 minutes of activity.”
7. We are building a clickstream pipeline that joins page-view logs to purchase logs.
  - a) Give three invariants that should hold if the join and aggregation are behaving correctly.
  - b) Give two failure modes that would violate those invariants (and how we would detect each).
8. For each operation, state whether it is associative and whether it is commutative. If it fails, give a concrete counterexample.
  - a)  $(a, b) \mapsto a + b$
  - b)  $(a, b) \mapsto a - b$
  - c)  $(a, b) \mapsto \max(a, b)$
  - d) “average of averages”:  $(\bar{x}_1, \bar{x}_2) \mapsto (\bar{x}_1 + \bar{x}_2)/2$
  - e) concatenation of strings
9. We want the global mean of values split across workers.
  - a) Explain why averaging the per-worker means and then averaging those means is generally incorrect.
  - b) Provide a reduce-friendly summary object that is safe to aggregate and explain why.
  - c) Repeat (a) and (b) for the global variance.
10. **Amdahl’s Law.** A pipeline has a serial fraction of  $(1 - p)$  and a parallel fraction  $p$ .
  - a) For  $p = 0.95$ , compute  $S(N) = \frac{1}{(1-p)+p/N}$  for  $N \in \{1, 2, 4, 8, 16, 32\}$ .
  - b) For the same  $p$ , compute the limiting speed-up as  $N \rightarrow \infty$ .
  - c) Interpret the result in the language of bottlenecks.
11. **Gustafson’s Law.** For  $p = 0.95$ , compute  $S(N) = N - (1 - p)(N - 1)$  for  $N \in \{1, 2, 4, 8, 16, 32\}$ . Compare the trend to Amdahl’s law and explain when each viewpoint is more relevant.
12. **Little’s Law.** A streaming system receives  $\lambda = 2000$  events per second and the average time in system is  $W = 0.8$  seconds.
  - a) Compute  $L = \lambda W$  and interpret what it means operationally.
  - b) If  $W$  doubles due to a bottleneck, what happens to  $L$ ?
  - c) Give one concrete big data cause of rising  $W$  that is not “more CPU arithmetic.”
13. Explain what predicate pushdown is in a columnar format workflow (such as Parquet). Give one example of a filter that is likely to be pushed down, and one example that is unlikely to be pushed down (explain why).

14. Describe three reasons Parquet is often preferable to CSV at scale. Then describe one situation where CSV might still be acceptable or even preferable.
15. Consider linear regression with design matrix  $X$  and response  $Y$  split across workers.
  - a) Write down the two summary matrices/vectors whose aggregation enables the normal-equation solution.
  - b) Explain why this is an example of a 2-phase pattern (full-data pass, then small-object computation).
  - c) Give one reason we might still prefer an iterative method at scale, even though the normal equations exist.
16. A session is defined as a maximal sequence of events with no gap larger than 30 minutes.
  - a) Explain why partitioning logs by day and sessionizing within each day can change the answer.
  - b) Propose two remedies: one that preserves the definition (at higher cost), and one that changes the definition (explicitly) to enable cheaper computation.
17. For each operation, state whether it might typically triggers a shuffle in Spark, and explain why.
  - a) `select()` a subset of columns
  - b) `filter()` on a simple predicate
  - c) `group_by(key) |> summarise(count = n())`
  - d) `arrange()` globally
  - e) `left_join()` on a high-cardinality key
18. In a group-by on key `user_id`, we observe that one task runs  $20\times$  longer than others.
  - a) Explain how key skew can produce stragglers.
  - b) Propose two mitigation strategies (one data-level, one execution-level) and state the trade-offs.
19. Consider an iterative workflow that repeatedly queries the same derived table.
  - a) Explain when caching helps and when it hurts.
  - b) Give a concrete example of caching the wrong intermediate and describe the resulting failure mode (spill-to-disk, executor OOM, or degraded performance).
20. We are using `sparklyr` or `PySpark`.
  - a) Explain why `collect()` is a common point of failure in distributed workflows.
  - b) Rewrite a workflow goal of the form “compute  $X$  from a huge table and bring it to the local session” so that only a reduced summary (or a small sample) is collected.
  - c) Give two practical rules of thumb for deciding whether a result is safe to collect.
21. Build an end-to-end pipeline for synthetic (or real) clickstream-like event data with fields `user_id`, `timestamp`, `event_type`, and optional `session_id`.
  - a) Generate or acquire at least 50 million events (synthetic is acceptable). Store the data in both CSV and Parquet.
  - b) Implement: ingestion, filtering, sessionization (with an explicit definition), and weekly aggregates (active users, sessions, conversions).
  - c) Include at least five validation invariants and show how we check them.
  - d) Diagnose at least one bottleneck (I/O, shuffle, skew, or collect) and demonstrate one mitigation.
  - e) Deliverables: code, a short technical report (2-4 pages), and a brief discussion of how changing the definition of a “session” changes results.
22. Using a text dataset (emails or messages) with spam/ham labels:
  - a) Implement MapReduce-style preprocessing: tokenize, compute term counts, and build features suitable for multinomial Naïve Bayes.
  - b) Train and evaluate a classifier (accuracy and at least one other metric). Include a train/test split that respects time if timestamps exist.
  - c) Compare two storage and execution choices (CSV vs Parquet, or local vs cluster, or caching vs no caching) and report the impact on runtime and memory pressure.
  - d) Deliverables: code, confusion matrix, and a discussion of one failure mode related to veracity (label noise, concept drift, or selection bias).

23. Create (or obtain) a dataset of entity-level time-stamped categorical events (synthetic ICD-like codes are acceptable).
  - a) Compute pairs of events occurring within a fixed window (for example, 5 years or 30 days) and propose a filtering rule that reduces the number of candidate pairs.
  - b) Define and compute a notion of directionality (event A tends to precede event B) and propose a statistical validation strategy (for example, stratification, permutation, or a model-based approach).
  - c) Construct 3-event and 4-event trajectories from validated pairs, then cluster trajectories using any method (state clearly what similarity means).
  - d) Deliverables: code, at least two visualizations of trajectory structure, and a discussion of how regime change or coding drift could invalidate conclusions.

## Chapter References

- [1] G.M. Amdahl. 'Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities'. In: *AFIPS Conference Proceedings*. ACM, 1967, pp. 483–485.
- [2] M. Anderson. *Understanding Moore's Law in 2025* [↗](#). Medium. June 2025.
- [3] P. Boily. *Data Understanding, Data Analysis, and Data Science (Course Notes)* [↗](#). Data Action Lab, 2022.
- [4] P. Boily, S. Davies, and J. Schellinck. *The Practice of Data Visualization* [↗](#). Data Action Lab, 2023.
- [5] CERN. *The Large Hadron Collider* [↗](#). Accessed: 2026-01-09.
- [6] CERN. *The Worldwide LHC Computing Grid (WLCG)* [↗](#). Accessed: 2026-01-09.
- [7] F. Chen and B.D. Ripley. 'Statistical Computing and Databases: Distributed Computing Near the Data' [↗](#). In: *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*. Ed. by K. Hornik, F. Leisch, and A. Zeileis. DSC 2003, March 20–22. ISSN 1609-395X. Vienna, Austria, Mar. 2003.
- [8] D. Chicco and G. Jurman. *Sepsis Survival Minimal Clinical Records* [↗](#). UCI Machine Learning Repository. 2020.
- [9] PyTorch Contributors. 'PyTorch Distributed Overview'. In: *The Linux Foundation* (July 2025).
- [10] J. Dean and S. Ghemawat. 'MapReduce: Simplified Data Processing on Large Clusters'. In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.
- [11] R.H. Dennard et al. 'Design of Ion-Implanted MOSFETs with Very Small Physical Dimensions'. In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268.
- [12] P. Diaconis and F. Mosteller. 'Methods for Studying Coincidences'. In: *Journal of the American Statistical Association* 84.408 (1989), pp. 853–861.
- [13] M. Dowle et al. *Rdatatable* [↗](#).
- [14] C.A.E. Goodhart. 'Problems of Monetary Management: The U.K. Experience'. In: *Papers in Monetary Economics*. Reserve Bank of Australia, 1975.
- [15] K. Grace. *Trends in the Cost of Computing* [↗](#). AI Impacts. Mar. 2015. (Visited on 01/09/2026).
- [16] J.L. Gustafson. 'Reevaluating Amdahl's Law'. In: *Communications of the ACM* 31.5 (1988), pp. 532–533.
- [17] *Hadoop Distributed File System (HDFS)*. Dec. 2025. URL: <https://www.databricks.com/glossary/hadoop-distributed-file-system-hdfs>.
- [18] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. 6th ed. Morgan Kaufmann, 2019.
- [19] M. Hopkins et al. *Spambase* [↗](#). UCI Machine Learning Repository. 1999.
- [20] A.M. Horst, A.P. Hill, and K.B. Gorman. *palmerpenguins: Palmer Archipelago (Antarctica) penguin data* [↗](#). R package version 0.1.0. 2020.
- [21] Intel. *What Is Hyper-Threading?* [↗](#). Accessed: 2026-01-09.

- [22] A.B. Jensen et al. 'Temporal disease trajectories condensed from population-wide registry data covering 6.2 million patients'. English. In: *Nature Communications* 5 (2014). DOI: [10.1038/ncomms5022](https://doi.org/10.1038/ncomms5022).
- [23] J.G. Koomey et al. 'Implications of Historical Trends in the Electrical Efficiency of Computing'. In: *IEEE Annals of the History of Computing* 33.3 (2011), pp. 46–54.
- [24] D. Laney. *3D Data Management: Controlling Data Volume, Velocity, and Variety*. Tech. rep. META Group, Feb. 2001.
- [25] G. Lanzafame. *Accelerating data science with Apache Spark and GPUs* [↗](#). June 2025.
- [26] Lenovo. *The Importance of FLOPS and its Impact on Your PC's Speed and Efficiency* [↗](#). Accessed: 2026-01-09.
- [27] J.D.C. Little. 'A Proof for the Queueing Formula:  $L = \lambda W$ '. In: *Operations Research* 9.3 (1961), pp. 383–387.
- [28] J. Luraschi, K. Kuo, and E. Ruiz. *Mastering Spark with R: The Complete Guide to Large-Scale Analysis and Modeling* [↗](#). O'Reilly Media, 2020.
- [29] J. Luraschi et al. *sparklyr: R Interface to Apache Spark* [↗](#). Sept. 2016.
- [30] J. Młacki. *7 Vs of Big Data: What Are They and Why Are They So Important?* [↗](#). DS Stream (blog post). May 2025.
- [31] G.E. Moore. 'Cramming More Components onto Integrated Circuits'. In: *Electronics* 38.8 (1965), pp. 114–117.
- [32] M.C. Nakhaee, J. Van Goey, and M. Chan. *Easily load the Palmer Penguins dataet in Python* [↗](#). Sept. 2024.
- [33] Penn LPS Online. *The Impact of Big Data on Scientific Research* [↗](#). Feature article. Accessed 2026-01-07.
- [34] Pure Storage. *Parallel vs. Distributed Computing: An Overview* [↗](#). Pure Storage Knowledge Base (JP).
- [35] N. Richardson et al. *arrow: Integration to 'Apache' 'Arrow'* [↗](#). 2023.
- [36] A. Sandberg and N. Bostrom. *Whole Brain Emulation: a Roadmap* [↗](#). Tech. rep. 2008.
- [37] J. Schneider and I. Smalley. *CPU vs. GPU for machine learning* [↗](#). Jan. 2025.
- [38] The Ray Team. *Ray Train Overview – Ray 2.48.0* [↗](#). 2025.
- [39] The Ray Team. *Ray Train: Scalable Model Training – Ray 2.48.0* [↗](#). 2025.
- [40] D. Wagner, D. Heider, and G. Hattab. *Secondary Mushroom* [↗](#). UCI Machine Learning Repository. 2021.
- [41] *What Is Hadoop?* Dec. 2025. URL: <https://www.databricks.com/glossary/hadoop>.
- [42] *What Is Parquet?* Dec. 2025. URL: <https://www.databricks.com/glossary/what-is-parquet>.
- [43] Wikipedia contributors. *SETI@home* [↗](#). Accessed: 2026-01-09.
- [44] Wikipedia contributors. *Sunway TaihuLight* [↗](#). Accessed 2026-01-09.
- [45] S. Williams, A. Waterman, and D. Patterson. 'Roofline: An Insightful Visual Performance Model for Multicore Architectures'. In: *Communications of the ACM* 52.4 (2009), pp. 65–76.
- [46] W.A. Wulf and S.A. McKee. 'Hitting the Memory Wall: Implications of the Obvious'. In: *ACM SIGARCH Computer Architecture News* 23.1 (1995), pp. 20–24.