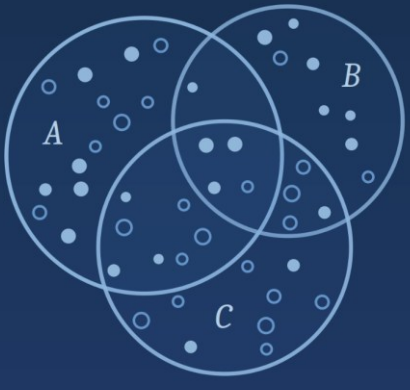

Applied Set Theory

Foundations and Practice for
Computing, Data, and Engineering

{ ∈ }

Applied Set Theory

Foundations and Practice for Computing, Data, and Engineering

Concept and editorial direction:

Dmitry Vostokov, DumpAnalysis.org

Drafting and revision assistance: Fable 5.1 Extra

Independent editorial review: GPT-6 Astra Max

Version 12 • 12 September 2026

ISBN-13: 978-1-919135-02-1

Manuscript Version 12, September 2026.

Contents

Preface

Set theory occupies an unusual position among mathematical disciplines. On the one hand, it is the accepted foundation on which the rest of mathematics is formally erected. On the other hand, it is a working toolkit that practitioners in computing, engineering, and data management use daily, often without noticing. A database query, a type declaration, a probability model, an access-control policy, and a formal specification are all, at bottom, exercises in the manipulation of sets, relations, and functions.

Most textbooks serve one of these audiences and neglect the other. Foundational texts develop ordinals, cardinals, and independence results with full rigor but rarely descend to a `JOIN` clause or a hash set. Discrete-mathematics texts for programmers introduce set notation in a single chapter and then move on. Their readers are left without the conceptual depth to recognize when a "set" in software is not behaving like a set at all, for example, when duplicate rows, mutable elements, or floating equality quietly break the mathematics the design relies on.

We wrote this book for the reader who wants both: precise foundations, honestly presented, and a sustained account of how set-theoretic concepts

operate in applied settings. The first two parts develop the language and the classical theory: operations, relations, functions, orders, cardinality, ordinals, and the ZFC axioms, with formal definitions throughout. The third part follows the concepts into practice: Boolean algebras and logic, the relational data model, collections and types in programming languages, formal specification languages, combinatorial counting, measure-theoretic probability, graphs and set systems, and, finally, the deliberate generalizations (fuzzy sets, rough sets, and multisets) that arise when applications outgrow the classical notion.

Three commitments shape the exposition.

Definitions are load-bearing. We introduce the book's principal concepts by numbered definitions before we use them, and we use them afterward in exactly the defined sense. Auxiliary terms that a chapter uses in passing (a literal in a formula, a preorder, a decision diagram) are explained where they occur, in the standard sense of the cited literature. Common usage is often looser than the formal notion: "function" in programming, "relation" in databases, "set" in almost every language's standard library. Where they diverge, we state the divergence explicitly rather than paper over it.

Epistemic status is stated. Some assertions in this book are theorems of ZFC. Some are conventions of notation. Some are facts about particular software

systems. Some, like the continuum hypothesis, are known to be undecidable from the standard axioms. We distinguish these throughout. Where we make a simplification (and in an applied book some must be made), we flag it as such.

Applications are worked, not gestured at. Each application chapter contains worked examples carried through to a result: a relational schema with an instance and the algebra behind its queries evaluated on it, a specification with its initial states, its invariant, and a proof obligation discharged (or shown to fail), a counting argument carried through to a number. The reader should finish each chapter able to do the corresponding work, not merely to recognize its vocabulary.

The book assumes mathematical maturity at the level of an undergraduate course in discrete mathematics or the practical equivalent: comfort with symbolic notation. No prior exposure to axiomatic set theory is assumed. We state results without proof. The epistemic status of every claim is flagged in the text, and where a result is stated rather than demonstrated we cite the source, by author, year, and chapter or section, in which the demonstration can be found; the bibliography collects those sources. Readers whose interest is primarily applied may read Chapters 1–5, skim Chapters 6–8, and then proceed to any chapter of Part III independently. The dependency notes at the

start of each chapter make the required background explicit.

How to Read This Book: Notation and Conventions

We collect the notation used throughout in Appendix A. The conventions below govern the entire text.

Logical symbols. We write $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\Rightarrow$ (implies), $\Leftrightarrow$ (if and only if), $\forall$ (for all), $\exists$ (there exists). The abbreviation *iff* means "if and only if."

Sets. Membership is $x \in A$; its negation is $x \notin A$. Set-builder notation $\{x \in A : \varphi(x)\}$ denotes the set of members of A satisfying the condition φ. We explain the significance of the bounding set A in Chapter 1. The empty set is $\emptyset$. Number systems are $\mathbb{N}$ (the natural numbers, *including* 0; we chose this convention deliberately and use it consistently), $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$.

Definitions, theorems, examples. Each is numbered within its chapter and type: Definition 3.2 is the second definition of Chapter 3. Citations of the form "by Theorem 6.4" always refer to these numbers.

Named systems. Where an application chapter discusses a concrete technology (SQL, Python, Java, Z, Alloy), we describe version-independent behavior in the present tense and attribute version-dependent behavior explicitly. Statements about

software describe the systems as specified at the time of writing. They are empirical claims about artifacts, not mathematical assertions.

Part I: The Language of Sets

Chapter 1. Sets and Membership

Dependencies: none. This chapter fixes the primitive notions and notation on which every later chapter relies.

1.1 The primitive notions

Set theory is built from two primitive notions: **set** and **membership**. We do not define them in terms of anything simpler. Instead, we constrain their behavior by axioms, stated informally in this chapter and axiomatically in Chapter 8. Informally, a set is a collection of definite, distinct objects that we consider as a single object in its own right. The objects belonging to a set are called its **elements** or **members**.

Definition 1.1. We write $x \in A$ to mean that x is an element of the set A, and $x \notin A$ for its negation. The relation $\in$ is the *membership relation*.

Two principles govern the informal theory. The first principle says that a set is determined entirely by its members.

Definition 1.2 (Extensionality). Sets A and B are equal, written $A = B$, iff they have exactly the same elements: $\forall x\,(x \in A \Leftrightarrow x \in B)$.

Extensionality has immediate practical consequences. A set has no notion of the *order* of its elements and no notion of *multiplicity*: {1,2,3}, {3,1,2}, and {1,1,2,3,3} are three ways of writing the same set. When an application requires order, the right structure is a tuple or sequence (Section 2.4). When it requires multiplicity, the right structure is a multiset (Chapter 16). Choosing a set where the application semantics demand a sequence or multiset is one of the most common modeling errors in practice, and we return to it repeatedly in Part III.

The second principle says which sets exist. The naive version states that for any property φ, there is a set $\{x : \varphi(x)\}$ of all objects satisfying φ. This principle is called **unrestricted comprehension**, and it is false.

Theorem 1.1 (Russell, 1901). There is no set R such that for all x, $x \in R \Leftrightarrow x \notin x$.

Theorem 1.1 is not a paradox *within* a well-formulated theory. It is a demonstration that unrestricted comprehension cannot be an axiom of any consistent theory. Modern set theory adopts a repair called **separation** (also called *restricted comprehension*): given a set A that already exists and a property φ, the collection $\{x \in A : \varphi(x)\}$ is a set. In other words, we may carve subsets out of existing sets, but we may not conjure a set from a bare property. Collections too large to be sets (the collection of all sets, the collection of all groups) are called **proper classes**. We may speak of them, but

they are not elements of anything, and the machinery of set theory does not apply to them. In this book, every set-builder expression is bounded, either explicitly ($\{x \in \mathbb{N} : x \text{ is prime}\}$) or by a bounding set clear from context.

Note. For nearly all applied work the distinction between sets and proper classes never bites: application domains (rows of a table, states of a machine, IP addresses) are unproblematically sets. The distinction matters when we quantify over "everything" and, notably, in the semantics of some type systems and category-theoretic frameworks, where size issues resurface. Honesty requires stating the restriction once, precisely, which we have now done.

1.2 Specifying sets

There are three standard ways to present a set.

Enumeration. We list the elements between braces: $\{2,3,5,7\}$. This way is suitable only for finite (and, in informal ellipsis notation, simply patterned) sets.

Set-builder notation. We write $\{x \in A : \varphi(x)\}$, read "the set of all x in A such that $\varphi(x)$." A common and useful variant maps an expression over a set: $\{x^2 : x \in \mathbb{N}\}$ abbreviates $\{y \in \mathbb{N} : \exists x \in \mathbb{N}\ (y = x^2)\}$. Readers who know Python will recognize the deliberate echo in comprehension syntax: `{x*x for`

`x in range(n)`}. Of course, the mathematical notation came first, by a century.

Characteristic conditions via prior operations. We present a set as the value of operations on previously given sets: $A \cup B$, $\mathcal{P}(A)$, and so on (Chapter 2).

Definition 1.3 (Empty set). $\emptyset$ is the set with no elements: $\forall x\ (x \notin \emptyset)$.

By extensionality the empty set is unique, because any two memberless sets have (vacuously) the same members. Assertions of the form "every element of $\emptyset$ has property φ" are true for every φ. Such statements are called **vacuously true**. Vacuous truth is a frequent source of software defects. For example, a validation routine that checks "all items in the cart satisfy the discount policy" approves the empty cart, whether or not that was intended. The mathematics is unambiguous here. The specification error is in failing to decide what the empty case *should* do.

Definition 1.4 (Singleton, pair). For any objects a, b: $\{a\}$ is the set whose only element is a, and $\{a, b\}$ the set whose only elements are a and b. Note that $\{a, a\} = \{a\}$ by extensionality.

A singleton is not its element: $\{a\} \neq a$ in general, and $\emptyset \neq \{\emptyset\}$ (the latter has one element, the former none). This distinction is exactly the distinction between a value and a container holding one value,

familiar from any programming language. Conflating the two is a type error in both settings.

Example 1.1. Let $A = \{\emptyset, \{\emptyset\}\}$. Then $\emptyset \in A$, $\{\emptyset\} \in A$, and A has exactly two elements. Moreover, $\emptyset \subseteq A$ (vacuously) and $\{\emptyset\} \subseteq A$ (since its one element, $\emptyset$, is in A). Therefore, $\{\emptyset\}$ is *both* an element and a subset of A. Membership and inclusion are different relations, and only careful attention to $\in$ versus $\subseteq$ keeps such examples straight.

1.3 Subsets and inclusion

Definition 1.5 (Subset). $A \subseteq B$ (A is a *subset* of B) iff every element of A is an element of B: $\forall x\,(x \in A \Rightarrow x \in B)$. If additionally $A \neq B$, then A is a **proper subset**, written $A \subsetneq B$.

Theorem 1.2. For all sets A, B, C:
(i) $A \subseteq A$ (reflexivity);
(ii) if $A \subseteq B$ and $B \subseteq A$ then $A = B$ (antisymmetry);
(iii) if $A \subseteq B$ and $B \subseteq C$ then $A \subseteq C$ (transitivity);
(iv) $\emptyset \subseteq A$.

Clause (ii) is the workhorse of set-theoretic practice: to show that two sets are equal, we show that each contains the other. Nearly every set identity in Chapter 2 can be established this way. The same **double-inclusion** discipline is also the correct way to verify, for example, that a rewritten database query returns the same rows as the original: we

show that every row of each result belongs to the other.

Example 1.2 (Inclusion chains in practice). Access-control systems are inclusion structures. Let U be the set of all user accounts, $E \subseteq U$ the employees, $D \subseteq E$ the developers, and $A \subseteq D$ those with production access. Then the policy "production access implies developer status" is precisely $A \subseteq D$, and an audit is a verification of stated inclusions. Formulating policies as inclusions makes their logical consequences computable: from $A \subseteq D$ and $D \subseteq E$, transitivity yields $A \subseteq E$ with no further checking.

1.4 What may be an element?

In pure set theory, everything is a set. The elements of sets are themselves sets, and the number 3, the ordered pair (1,2), and the function $x \mapsto x^2$ are all encoded as particular sets (Chapters 2, 4, and 7 show how). This uniformity is what makes set theory serviceable as a *foundation*: one primitive relation suffices for all of mathematics.

Applied practice is different, and it is worth being precise about the difference. When a database stores a set of customer identifiers or a program manipulates a set of strings, we treat the elements as **atoms** (in the set-theoretic literature, *urelements*): objects with no internal set-theoretic structure. A variant of the standard theory (ZFA) accommodates

urelements officially. For applied purposes, we simply fix a **universe of discourse** $\mathcal{U}$, the set of all objects relevant to the application, and work with subsets of $\mathcal{U}$ and sets built from them. All complementation (Definition 2.4) is relative to this universe. Declaring the universe explicitly is the modeling discipline that prevents most abuses of "the set of all …" in requirements documents. *All* customers, but in which system, as of when, including deactivated accounts or not? A universe is a decision, not a discovery.

Note. Whether the elements of applied sets "really are" sets is a question with no operational content. What matters is which structure the application observes. The database cannot distinguish an integer implemented as a von Neumann ordinal from one implemented as two's-complement bits, and neither can any theorem in this book. Therefore, we adopt atoms freely in applied chapters, use pure sets in foundational ones, and flag the convention where it matters.

1.5 Equality, identity, and equality in software

Extensionality settles what set equality *is*. Applications must also settle how it is *decided*. Three points deserve early emphasis, and each recurs in Chapter 11.

First of all, deciding $A = B$ for finite sets requires deciding equality of *elements*. Mathematics assumes a crisp, given identity on the universe. Software must implement one, and the implementation can diverge from the intent. Floating-point `NaN` is not equal to itself under IEEE 754 comparison. Two structurally identical objects may be distinct by reference. Unicode strings admit multiple encodings of the same visible text. A "set of strings" whose membership test uses byte equality is a set, but over a finer universe than the user may believe.

Then, extensionality presumes that membership is stable. If elements mutate after insertion (an object's hash-relevant fields change while it sits in a hash set), the structure ceases to behave as a set at all: it may report $x \notin S$ immediately after x was inserted. The mathematical concept has no analogue of this failure because sets do not change. The applied lesson is that *mutable elements and set semantics are in tension*, and the mainstream collections libraries document (or should document) the resulting contract.

Finally, a set in the mathematical sense is not a data structure. Hash sets, sorted trees, bit vectors, and Bloom filters all *represent* sets, with different costs and, in the Bloom filter's case, only approximately (membership tests admit false positives). The mapping from concept to representation is a design choice. The concept itself is representation-free. Keeping the two levels separate (the set specified,

the structure implementing it) is the central habit this book aims to build. Chapter 12 gives it formal expression as the relationship between a specification and its refinement.

1.6 Summary

A set is determined by its members (extensionality). Sets are formed by separation from existing sets, not by unrestricted comprehension (Russell). Inclusion $\subseteq$ is a partial order whose antisymmetry gives the standard method for showing set equality. Applied modeling fixes a universe of discourse, treats domain objects as atoms, and must implement the element equality that mathematics takes as given. In the next chapter, we develop the algebra of operations on sets within a fixed universe.

Chapter 2. The Algebra of Sets

Dependencies: Chapter 1. In this chapter, we fix a universe $\mathcal{U}$ and develop the operations, identities, and constructions (union, intersection, difference, complement, power set, and product) used everywhere else in the book.

2.1 The Boolean operations

Throughout this chapter $A, B, C \subseteq \mathcal{U}$ for a fixed universe of discourse $\mathcal{U}$.

Definition 2.1 (Union). $A \cup B = \{x \in \mathcal{U} : x \in A \lor x \in B\}$.

Definition 2.2 (Intersection). $A \cap B = \{x \in \mathcal{U} : x \in A \land x \in B\}$. Sets with $A \cap B = \emptyset$ are **disjoint**.

Definition 2.3 (Difference). $A \setminus B = \{x \in A : x \notin B\}$. The **symmetric difference** is $A \triangle B = (A \setminus B) \cup (B \setminus A)$.

Definition 2.4 (Complement). $A^c = \mathcal{U} \setminus A$.

Complement is the one operation here that is *relative*: it is undefined until a universe is fixed, because "everything not in A" is a proper class absent a bound. In applications, silent shifting of the universe is a genuine source of error. The complement of "customers who ordered this year"

differs depending on whether $\mathcal{U}$ is all current customers, all customers ever, or all people. Every use of c in this book has an explicit universe in scope.

The correspondence between set operations and logical connectives is exact, and it is no accident: $\cup$ is defined by $\vee$, $\cap$ by $\wedge$, c by $\neg$, and $\subseteq$ by $\Rightarrow$. Therefore, every propositional tautology yields a set identity. We make this correspondence precise in Chapter 9. The principal identities follow.

Theorem 2.1 (Algebra of sets). For all $A, B, C \subseteq \mathcal{U}$:
(i) *Commutativity:* $A \cup B = B \cup A$; $A \cap B = B \cap A$.
(ii) *Associativity:* $(A \cup B) \cup C = A \cup (B \cup C)$; likewise for $\cap$.
(iii) *Distributivity:* $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$; $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$.
(iv) *Identity and bounds:* $A \cup \emptyset = A$; $A \cap \mathcal{U} = A$; $A \cup \mathcal{U} = \mathcal{U}$; $A \cap \emptyset = \emptyset$.
(v) *Idempotence:* $A \cup A = A \cap A = A$.
(vi) *Complementation:* $A \cup A^c = \mathcal{U}$; $A \cap A^c = \emptyset$; $(A^c)^c = A$.
(vii) *De Morgan laws:* $(A \cup B)^c = A^c \cap B^c$; $(A \cap B)^c = A^c \cup B^c$.

Example 2.1 (Query rewriting). A monitoring rule pages when a host is *not* (healthy and reachable). By De Morgan, this condition is exactly: unhealthy *or* unreachable. The rewrite is not a heuristic but an identity. Rule engines and query optimizers apply such identities mechanically: distributivity to push

filters inward, De Morgan to eliminate negations. When an "optimized" rule and its original disagree on some input, Theorem 2.1 guarantees that the discrepancy lies elsewhere: in nulls, in duplicates, or in evaluation order against a changing universe. These are the three standard suspects, and we examine them in Chapter 10.

Example 2.2 (Symmetric difference as change). For a directory D_t of files at time t, the sets $D_{t+1} \setminus D_t$ (added), $D_t \setminus D_{t+1}$ (removed), and $D_t \mathbin{\triangle} D_{t+1}$ (changed either way) constitute the *diff*. Symmetric difference is associative and has $\emptyset$ as identity, with every set its own inverse. Therefore, applying the same patch twice restores the original. This algebraic fact underlies XOR-based storage schemes such as RAID parity: a lost disk is the symmetric difference of the surviving ones.

2.2 Indexed families and generalized operations

Binary union and intersection extend to arbitrary collections.

Definition 2.5 (Indexed family). Given an *index set* I and for each $i \in I$ a set A_i, the assignment $(A_i)_{i \in I}$ is an *indexed family*. Its **union** and **intersection** are

$$\bigcup_{i \in I} A_i = \{x : \exists i \in I,\ x \in A_i\},$$

$$\bigcap_{i \in I} A_i = \{x : \forall i \in I,\ x \in A_i\}.$$

For $I = \emptyset$ the union is $\emptyset$, while the intersection is all of $\mathcal{U}$: every x vacuously belongs to all zero of the sets. The empty intersection defaulting to "everything" surprises the unwary in one applied place above all, and it is a common one. A filter built as a conjunction of conditions, applied with zero conditions, selects everything. Query builders and firewall policies must decide the zero-clause case deliberately.

The De Morgan and distributive laws generalize: $(\bigcup_i A_i)^c = \bigcap_i A_i^c$ and dually, by the same reasoning, with quantifier negation replacing connective negation.

Definition 2.6 (Partition). A partition of a set A is a family $\{B_i\}_{i \in I}$ of nonempty, pairwise disjoint sets with $\bigcup_{i \in I} B_i = A$.

Partitions are the mathematical form of classification: every element is in exactly one block. Sharding a keyspace across servers, assigning each transaction to exactly one account, and splitting a dataset into training, validation, and test sets are all partition claims. Each failure mode is a violation of one clause: overlap (an element in two blocks), gap (an element in none), or a spurious empty block. In Chapter 5, we establish the fundamental correspondence between partitions and equivalence relations.

2.3 The power set

Definition 2.7 (Power set). $\mathcal{P}(A) = \{X : X \subseteq A\}$, the set of all subsets of A.

Theorem 2.2. If A is finite with $|A| = n$, then $|\mathcal{P}(A)| = 2^n$.

A subset $X \subseteq A$ is determined by an independent yes/no choice for each of the n elements, namely, whether it belongs to X. The function assigning to each subset its yes/no pattern is the **characteristic function** $\chi_X : A \to \{0,1\}$, with $\chi_X(a) = 1$ iff $a \in X$. This correspondence is the theoretical basis of the *bit-vector* (bitmap) representation of sets. We fix an enumeration of a universe $\mathcal{U}$ of size n; then a subset is a word of n bits, and union, intersection, and complement become bitwise OR, AND, and NOT, executed 64 elements per machine instruction. Database bitmap indexes and the bitboards of chess engines are industrial applications of Definition 2.7.

The exponential growth in Theorem 2.2 has a dark side familiar to most engineers: state spaces are power sets. A system of n independent boolean feature flags has 2^n configurations. Testing "all combinations" is infeasible past small n, which is why combinatorial testing (Chapter 13) trades exhaustiveness for coverage guarantees.

Example 2.3. $\mathcal{P}(\emptyset) = \{\emptyset\}$, $\mathcal{P}(\{\emptyset\}) = \{\emptyset, \{\emptyset\}\}$, and iterating $\mathcal{P}$ from $\emptyset$ generates an unbounded

hierarchy of finite sets. These are the first rungs of the cumulative hierarchy of Chapter 8, and a first hint that "nothing," iterated, suffices to build everything.

2.4 Ordered pairs and Cartesian products

Extensionality makes $\{a, b\} = \{b, a\}$. To record order, we need a construction for which $(a, b) = (c, d)$ iff $a = c$ and $b = d$. That *characteristic property* is the entire requirement. The standard implementation within set theory is due to Kuratowski.

Definition 2.8 (Ordered pair). $(a, b) = \{\{a\}, \{a, b\}\}$.

Theorem 2.3. $(a, b) = (c, d)$ iff $a = c$ and $b = d$.

Note. Definition 2.8 is an *encoding*, not a discovery about what pairs "really are." Any construction satisfying Theorem 2.3 would serve. Several inequivalent ones exist, and no theorem elsewhere in this book depends on the choice, only on the characteristic property. This is precisely the situation of a programmer coding to an interface rather than an implementation, and it is the first of several places where set theory models the discipline of *data abstraction* rather than merely supplying vocabulary for it. Accordingly, from now on we use pairs solely through Theorem 2.3.

Definition 2.9 (Cartesian product). $A \times B = \{(a, b) : a \in A, b \in B\}$. For n sets, $A_1 \times \cdots \times A_n$

consists of the n-tuples $(a_1, \ldots, a_n)$ with $a_i \in A_i$. We write A^n when all factors equal A. An n-tuple may be encoded by nested pairs; again, only the characteristic property matters.

For finite sets, $|A \times B| = |A| \cdot |B|$. This equality is the *product rule* of counting (Chapter 13) and the reason the product is so named.

Example 2.4 (Products are everywhere in data). A record type with fields `user_id : U`, `timestamp : T`, `action : A` denotes the product $U \times T \times A$: the possible records are exactly the triples. A database table's rows live in the product of its column domains (Chapter 10 refines this view). A coordinate pair lives in $\mathbb{R}^2$. The state of a machine with k registers lives in the product of their value sets. Two modeling facts follow directly from the mathematics. First, products grow multiplicatively, so state spaces explode. Second, a product type cannot express *constraints between components*. "End date after start date" is not a product of independent choices but a subset of one: $\{(s, e) \in D \times D : s \leq e\}$. Much of data validation is the maintenance of a distinguished subset of a product. Forgetting this distinction (treating any element of the raw product as valid) is the abstract form of an entire family of bugs. "Make illegal states unrepresentable" is the slogan form of this observation.

Theorem 2.4 (Products distribute over the Boolean operations in each argument).

$A \times (B \cup C) = (A \times B) \cup (A \times C)$; likewise for $\cap$ and $\setminus$. But $\times$ is neither commutative nor associative up to equality: $A \times B \neq B \times A$ in general, and $(A \times B) \times C$ and $A \times (B \times C)$ contain differently shaped elements, though a canonical bijection identifies them (Chapter 4).

The non-associativity remark is a fact about encodings with practical resonance. `((a,b),c)` and `(a,(b,c))` are distinct values in typed languages exactly as in set theory, and flattening between them is an explicit conversion, the same conversion the canonical bijection performs.

Definition 2.10 (Disjoint union). $A \sqcup B = (\{0\} \times A) \cup (\{1\} \times B)$: copies of A and B tagged apart before uniting. For finite sets $|A \sqcup B| = |A| + |B|$ even when $A \cap B \neq \emptyset$, since the tags prevent all collisions. This is the construction a tagged union (sum type) in a programming language denotes. The tag records *which* alternative a value came from, information the plain union $A \cup B$ discards. Now it is obvious why a sum type can hold two alternatives of the same underlying type without ambiguity and a plain union cannot.

2.5 Interval notation and running conventions

For $a, b \in \mathbb{R}$, $[a, b] = \{x \in \mathbb{R} : a \leq x \leq b\}$, with (a, b), $[a, b)$, $(a, b]$ excluding the corresponding endpoints. Context distinguishes the interval (a, b) from the

ordered pair; where both could occur, we say which. For $n \in \mathbb{N}$ we write $[n] = \{1, \dots, n\}$ (so $[0] = \emptyset$), a convention used constantly in combinatorics.

2.6 Summary

Within a fixed universe, the subsets form an algebra under union, intersection, and complementation satisfying the Boolean identities, with De Morgan duality linking the operations. The identities are propositional tautologies transported along the definitions of the operations. Families generalize the operations, with the empty-family cases dictated by quantifier logic. The power set collects all subsets and corresponds to characteristic functions. It grounds bit-vector representations and the exponential growth of configuration spaces. Ordered pairs are encodings valued only for their characteristic property, and products build compound domains whose distinguished subsets carry application constraints.

Chapter 3. Relations

Dependencies: Chapters 1–2. Relations are the book's central applied concept. They underlie databases (Chapter 10), graphs (Chapter 15), orders and equivalences (Chapter 5), and functions (Chapter 4). Each of these is a relation with additional properties.

3.1 Relations as sets of tuples

Definition 3.1 (Binary relation). A *binary relation* between sets A and B is any subset $R \subseteq A \times B$. We write a R b for $(a, b) \in R$. When $A = B$ we call R a relation *on* A. More generally, an *n-ary relation* over sets $A_1, \ldots, A_n$ is a subset $R \subseteq A_1 \times \cdots \times A_n$.

The definition's force is its austerity: a relation *is* the set of tuples for which it holds, and there is no further ingredient. "Less than" on $\{1,2,3\}$ is the set $\{(1,2), (1,3), (2,3)\}$. "Is a parent of" is the set of parent–child pairs. A table of orders is a set of order tuples. This identification is extensional, like everything in set theory, and it has two consequences worth making explicit. First of all, two relations defined by different rules but holding of the same tuples are *equal*: "$x < y$" and "$x \leq y$ and $x \neq y$" define one relation on $\mathbb{N}$, not two. The rule (an *intension*) is not the relation (its *extension*), and computing convergent extensions from divergent intensions is what query engines do all day.

Moreover, any set of pairs whatsoever is a relation; no rule, meaning, or regularity is required. The definition is permissive so that the interesting structure can be imposed as explicit, checkable *properties* (Section 3.3) rather than smuggled in.

Definition 3.2 (Domain of definition, range, field). For $R \subseteq A \times B$: $\mathrm{dom}R = \{a \in A : \exists b\ (a, b) \in R\}$ and $\mathrm{ran}R = \{b \in B : \exists a\ (a, b) \in R\}$. For a relation on A, its *field* is $\mathrm{dom}R \cup \mathrm{ran}R$.

Note the distinction between A (the declared source) and $\mathrm{dom}R$ (where R actually holds of something). They coincide exactly when R relates every element of A to at least one element of B. Such a relation is called *left-total*, and this totality clause is what functions will make mandatory (Chapter 4). In this book, the unqualified word *total* is reserved for the connexity property of Definition 3.5, an unrelated sense. The distinction between declared source and actual domain is the mathematical location of every "no rows found" and "missing key" condition in data systems.

Example 3.1. Extreme cases are legitimate relations and serve as boundary tests for general claims: the **empty relation** $\emptyset \subseteq A \times B$; the **full relation** $A \times B$; the **identity** $\mathrm{id}_A = \{(a, a) : a \in A\}$ on A. We recommend checking any purported statement about relations against all three before believing it.

3.2 Operations on relations

Since relations are sets, the Boolean operations of Chapter 2 apply verbatim: the union of two relations, the complement of a relation within $A \times B$, and so on. Two further operations are proper to relations.

Definition 3.3 (Converse). $R^{-1} = \{(b, a) : (a, b) \in R\} \subseteq B \times A$.

Definition 3.4 (Composition). For $R \subseteq A \times B$ and $S \subseteq B \times C$, the *composition* $S \circ R \subseteq A \times C$ is

$$S \circ R = \{(a, c) : \exists b \ (a \ \mathrm{R} \ b \wedge b \ \mathrm{S} \ c)\}.$$

The order convention in $S \circ R$ ("first R, then S") matches function composition, with which it will agree in Chapter 4. However, database and relation-algebra texts often write composition in the opposite (diagrammatic) order, $R; S$. Both denote the same construction. This book uses $\circ$ with the stated convention and ; when quoting diagrammatic sources.

Theorem 3.1. Composition is associative: $T \circ (S \circ R) = (T \circ S) \circ R$. The identity relations are units: $R \circ \mathrm{id}_A = R = \mathrm{id}_B \circ R$. Converse is involutive and antidistributes over composition: $(R^{-1})^{-1} = R$ and $(S \circ R)^{-1} = R^{-1} \circ S^{-1}$.

Example 3.2 (Composition is the join). Let $R \subseteq$ Customer $\times$ Order hold when the customer placed

the order, and $S \subseteq \text{Order} \times \text{Product}$ when the order contains the product. Then $S \circ R \subseteq \text{Customer} \times \text{Product}$ holds when the customer ordered the product. We compute it by matching the middle component and discarding it. This operation is precisely a relational *join* followed by a *projection* (Chapter 10 gives the general forms). The existential quantifier in Definition 3.4 is why information is lost: $S \circ R$ records *that* some order linked Alice to the widget, not *which*. Whether that loss is acceptable is an application decision, and the notation forces it into the open.

Example 3.3 (Relations as boolean matrices). A relation between finite A and B, with fixed enumerations, is a $|A| \times |B|$ boolean matrix M_R with $M_R[i][j] = 1$ iff $(a_i, b_j) \in R$. Under this encoding, converse is transposition and composition is boolean matrix multiplication ($\vee$ playing addition, $\wedge$ playing multiplication). Therefore, associativity of composition *is* associativity of matrix product, and the graph-reachability algorithms of Chapter 15 are iterated relational composition. The three encodings of a finite relation (set of pairs, boolean matrix, bipartite graph) are interchangeable. Fluency in translation among them is a working skill this book assumes from here on.

3.3 Properties of relations on a set

Definition 3.5. A relation R on a set A is:

- **reflexive** iff $\forall a \in A\colon\ a\ \mathrm{R}\ a$;
- **irreflexive** iff $\forall a \in A\colon\ \neg(a\ \mathrm{R}\ a)$;
- **symmetric** iff $\forall a, b\colon\ a\ \mathrm{R}\ b \Rightarrow b\ \mathrm{R}\ a$;
- **antisymmetric** iff $\forall a, b\colon\ (a\ \mathrm{R}\ b \wedge b\ \mathrm{R}\ a) \Rightarrow a = b$;
- **asymmetric** iff $\forall a, b\colon\ a\ \mathrm{R}\ b \Rightarrow \neg(b\ \mathrm{R}\ a)$;
- **transitive** iff $\forall a, b, c\colon\ (a\ \mathrm{R}\ b \wedge b\ \mathrm{R}\ c) \Rightarrow a\ \mathrm{R}\ c$;
- **total** (or *connex*) iff $\forall a, b\colon\ a\ \mathrm{R}\ b \vee b\ \mathrm{R}\ a$.

Several cautions are in order, and each is a recurrent source of error. *Antisymmetric* is not the negation of *symmetric*: id_A is both, and $R = \{(a, b), (b, a), (b, c)\}$ on $\{a, b, c\}$ is neither (it fails antisymmetry at the pair a, b and symmetry at the pair b, c). On a nonempty carrier, *irreflexive* is stronger than "not reflexive"; on the empty set the empty relation is vacuously both reflexive and irreflexive, which is one more reason to test claims against Example 3.1. Transitivity fails for many relations that intuition wants to close: "is one hop from" in a network, "differs by at most ε" in floating-point comparison (reflexive and symmetric, but ε-close steps chain into arbitrarily large gaps), "is a direct dependency of" in a build system. When an application needs the transitive version of a non-transitive relation, the correct object is the closure.

Definition 3.6 (Closures). The *reflexive closure* of R on A is $R \cup \mathrm{id}_A$; the *symmetric closure* is $R \cup R^{-1}$; the **transitive closure** R^{+} is the smallest transitive

relation containing R, and $R^* = R^+ \cup \mathrm{id}_A$ is the *reflexive–transitive closure*.

Theorem 3.2. R^+ exists and equals both (i) the intersection of all transitive relations on A containing R, and (ii) $\bigcup_{n\geq 1} R^n$, where $R^1 = R$ and $R^{n+1} = R \circ R^n$. In other words, aR^+b iff a finite chain $a = x_0 \,\mathrm{R}\, x_1 \,\mathrm{R} \cdots \mathrm{R}\, x_n = b$ $(n \geq 1)$ connects them.

The two characterizations serve different purposes. Characterization (i) justifies *reasoning* about R^+: it is well-defined and minimal. Characterization (ii) is an *algorithm schema*, but it must be read with care. The individual powers need not settle down. For $R = \{(0,1), (1,0)\}$ on $\{0,1\}$, the odd powers all equal R and the even powers all equal id, while the closure contains both self-pairs, which need walks of length two. The correct procedure accumulates: $C_0 = \emptyset$ and $C_{k+1} = C_k \cup R \cup (R \circ C_k)$, stopping when $C_{k+1} = C_k$. The comparison is between cumulative relations, not between consecutive powers. For $n = |A| > 0$ the accumulation is complete after n stages, that is, $R^+ = \bigcup_{j=1}^{n} R^j$, because a chain of more than n steps repeats a vertex and can be shortened. The familiar bound $n - 1$ applies to chains between distinct vertices and to acyclic relations, but not to self-reachability. Warshall's algorithm computes the closure in $O(|A|^3)$. Incremental maintenance under edge insertion and deletion is harder, and it is the real cost behind "recompute who can access what" in permission systems.

Example 3.4 (Closures in production). `A depends on B` in a package manager is the direct-dependency relation R. Installation requires R^+ (all transitive dependencies). A cycle in R^+, that is, some a with aR^+a, is a dependency cycle, reported as an error precisely because it makes the intended order (Chapter 5) impossible. The same holds for `extends` in a class hierarchy, `includes` in a build, and `granted-to` in role inheritance. In each case, the stored relation is the direct one, the semantic relation is its closure, and the gap between them is where both the performance costs and the surprising entitlements hide. Security reviews that examine only direct grants are examining R when the threat model concerns R^+.

3.4 Representing and storing relations

A finite relation admits several representations with distinct operational profiles. Choosing among them is a matter of which operations dominate.

The **extensional table** (set of tuples, as rows) supports enumeration and ad-hoc querying; it is the relational database's choice (Chapter 10). The **adjacency list** (for each a, the set $\{b : a \ \mathrm{R}\ b\}$) supports fast forward navigation and is the graph traversal's choice. Note that it represents R as a *function from A to $\mathcal{P}(B)$*, an equivalence to which Chapter 4 returns. The **boolean matrix** supports composition and closure by bulk bit operations. The **predicate** (code computing membership)

represents infinite or huge relations intensionally: `x divides y` need not be tabulated. The price is that only membership testing is available. Enumeration, converse, and cardinality are not, in general, computable from a membership oracle alone. Much practical system design consists of picking, and converting between, these four representations. The conversions themselves (materializing a predicate over a finite domain, inverting an adjacency list) are standard, and their costs are the subject of any algorithms text.

3.5 Summary

A relation is a set of tuples, individuated extensionally. Declared source and actual domain differ in general. Converse and associative composition give relations an algebra in which finite relations are boolean matrices and composition is join-then-project. The structural properties of Definition 3.5 are distinct properties, checked (not assumed) for each application relation. Where transitivity or reflexivity is needed but absent, closures supply the smallest repair, with Theorem 3.2 providing the justification and the cumulative iteration providing the algorithm. In the next chapter, we single out the relations that are functions; Chapter 5 singles out the equivalences and orders.

Chapter 4. Functions

Dependencies: Chapters 1–3. Functions are relations with a uniqueness property. In this chapter, we develop their anatomy (injectivity, surjectivity, inverses, images) and the applied distinctions (partiality, function-as-rule versus function-as-graph) that software constantly exercises.

4.1 Functions as functional relations

Definition 4.1 (Function). A relation $f \subseteq A \times B$ is a *function from A to B*, written $f : A \to B$, iff for every $a \in A$ there is exactly one $b \in B$ with $(a, b) \in f$. That unique b is written $f(a)$. A is the **domain**, B the **codomain**.

Two conditions are packed into "exactly one": *totality* (at least one value for each $a \in A$) and *single-valuedness* or *functionality* (at most one). Dropping totality gives a **partial function**, written $f : A \rightharpoonup B$, defined on $\mathrm{dom} f \subseteq A$. Mathematics defaults to total functions. Computation defaults, whether it admits it or not, to partial ones. Division lacks a value at zero, a key lookup lacks a value at an absent key, a parser lacks a value at malformed input, and a program may diverge. The applied disciplines differ in how they surface partiality: raising an exception, returning a sentinel (`null`, `NaN`), returning an option/`Maybe` value (which *totalizes* the function by

enlarging its codomain to $B \sqcup \{\bot\}$, provided the computation actually delivers one of the enlarged values for every input; a computation that diverges is not made total by a richer result type), or restricting the domain by a precondition (Chapter 12). These are four engineering renderings of one mathematical situation. Arguments over which is "correct" are best conducted with the shared situation in view.

Note (graph versus rule, and the codomain). Definition 4.1 identifies a function with its graph, that is, with its set of input–output pairs. The identification is standard, but it has a known wrinkle: the graph does not determine the codomain ($x \mapsto x^2$ as a function $\mathbb{R} \to \mathbb{R}$ and as a function $\mathbb{R} \to [0, \infty)$ have the same graph), yet surjectivity (Definition 4.3) depends on the codomain. We therefore fix one convention for the whole book. A function is a triple (domain, codomain, graph), and two functions are *equal* iff their domains, codomains, and graphs all agree. Two functions with the same graph and different codomains are not equal; they have the same *rule*, and we say exactly that when it is what we mean. Separately, the graph deliberately ignores how the values are computed. Mergesort and bubblesort compute the same function; a lookup table and a closed formula may too. Equality of functions, in the sense just fixed, is the mathematical notion. Equality of algorithms is a different, finer relation, and equality of program texts is finer still. We must keep the layers (text, algorithm, function)

distinct: compilers preserve the function while replacing the algorithm, and testing observes only finitely much of the function while purporting to constrain it.

Example 4.1. A configuration file mapping hostnames to IP addresses is (or intends to be) a function $H \to I$: the file is literally a finite graph, stored pair by pair. Two entries assigning different addresses to the same hostname violate single-valuedness, and configuration parsers must choose a resolution (first wins, last wins, error). That choice is invisible in the type "function" and must be specified. Identical repeated entries are a different matter: as pairs in a set they are one pair, and any objection to them concerns multiplicity in the file, not functionality. Totality is a third matter again. A hash map *enforces* functionality by construction (a second `put` overwrites). An association list does not. A database table enforces it exactly when the input columns form a key (Chapter 10). None of these enforces totality over the intended hostname domain: a key constraint guarantees at most one address per hostname, not that every hostname the application will ask about has an entry. "Which invariants does the representation enforce, and which does the application merely hope for?" is the question this chapter equips the reader to ask precisely.

4.2 Injections, surjections, bijections

Definition 4.2 (Injective). $f: A \to B$ is *injective* (an *injection*, one-to-one) iff $f(a_1) = f(a_2) \Rightarrow a_1 = a_2$ for all $a_1, a_2 \in A$.

Definition 4.3 (Surjective). $f: A \to B$ is *surjective* (a *surjection*, onto) iff $\forall b \in B\ \exists a \in A:\ f(a) = b$; equivalently $\text{ran} f = B$.

Definition 4.4 (Bijective). f is *bijective* (a *bijection*, a one-to-one correspondence) iff it is both injective and surjective.

Example 4.2 (The applied readings). Injectivity is *no collisions*: user IDs to users, ISBNs to titles. A hash function h from a large key space to a small index space is deliberately non-injective. Hash-table design is the management of the resulting collisions, and a "perfect hash" is an injection constructed for a fixed finite key set. Surjectivity is *full coverage*: every physical shelf slot has at least one catalog entry pointing to it; every error code the API can emit appears in the documentation (surjectivity of the documenting map onto the emitted codes). Bijectivity is *faithful re-encoding*: serialization/deserialization, encryption/decryption, the pairing of primary keys with rows. Each property is destroyed by typical maintenance events. Merging user accounts breaks injectivity of the old-ID map. Adding an enum variant breaks surjectivity of an exhaustive handler.

Therefore, the properties are not one-time facts but *invariants to be maintained*, a theme Chapter 12 formalizes.

Theorem 4.1 (Composition preserves the properties). If $f: A \to B$ and $g: B \to C$, then $g \circ f: A \to C$ (composition of the underlying relations) is a function; composition of injections is injective, of surjections surjective, of bijections bijective. Conversely, if $g \circ f$ is injective then f is; if $g \circ f$ is surjective then g is.

The converse clauses are diagnostic tools, but only in one direction. If f is not injective then $g \circ f$ is not injective, and if g is not surjective then $g \circ f$ is not surjective. Read contrapositively: a two-stage pipeline that loses no information end to end certifies that its first stage lost none, and one that covers its target end to end certifies that its last stage covers it. The reverse inference is invalid. An end-to-end loss may originate at either stage (f the identity on $\{0,1\}$ and g constant loses everything at the second stage), and so may an end-to-end coverage failure (f the inclusion of $\{0\}$ into $\{0,1\}$ and g the identity fails coverage at the first). Therefore, Theorem 4.1 lets a successful composite certify a stage property, while a failed composite says only that some stage is at fault. Locating the fault needs stage-level observation. This is a small example of algebra setting the limits of what end-to-end evidence can establish.

Theorem 4.2 (Inverses). $f\colon A \to B$ is bijective iff there exists $g\colon B \to A$ with $g \circ f = \mathrm{id}_A$ and $f \circ g = \mathrm{id}_B$; such g is unique, equals the converse relation f^{-1}, and is itself a bijection.

One-sided inverses are strictly weaker and independently useful. The law $g \circ f = \mathrm{id}_A$ alone (a *retraction* of the injection f) is the round-trip law "decode after encode is the identity," satisfied by any lossless serializer with encoder f and decoder g. It constrains g only on $f[A]$, the encoder's range. It says nothing about the decoder's behavior on B-values outside that range, and it does imply that $f \circ g$ fixes every value in $f[A]$. Now it is clear why "encode after decode" may legitimately fail for non-canonical inputs (whitespace-variant JSON decoding to the same value, then re-encoding to the canonical form): such inputs lie outside $f[A]$. The other law, $f \circ g = \mathrm{id}_B$, is a different promise, namely that every B-value is the canonical encoding of something. A serialization contract should state *which* of the two laws it promises, and whether the decoder is total on all accepted encodings or partial on arbitrary byte strings. Most promise only the first, with a partial decoder.

4.3 Images and preimages

Definition 4.5. For $f\colon A \to B$, $X \subseteq A$, $Y \subseteq B$, the **image** of X and the **preimage** of Y are

$$f[X] = \{f(x)\colon x \in X\},$$

$$f^{-1}[Y] = \{a \in A : f(a) \in Y\}.$$

The preimage notation does not presuppose an inverse function: $f^{-1}[Y]$ is defined for arbitrary f. Preimage is the better-behaved operation.

Theorem 4.3. Preimage commutes with all Boolean operations: $f^{-1}[Y_1 \cup Y_2] = f^{-1}[Y_1] \cup f^{-1}[Y_2]$, $f^{-1}[Y_1 \cap Y_2] = f^{-1}[Y_1] \cap f^{-1}[Y_2]$, and $f^{-1}[B \setminus Y] = A \setminus f^{-1}[Y]$. Image commutes with union, but in general only $f[X_1 \cap X_2] \subseteq f[X_1] \cap f[X_2]$, with equality for all pairs iff f is injective; and $f[A \setminus X] \neq B \setminus f[X]$ in general.

Theorem 4.3 explains an asymmetry every data practitioner has met without naming it. Filtering *upstream* attributes by *downstream* conditions (preimage: "all requests whose response code is in the 5xx class") interacts cleanly with conjunction, disjunction, and negation of the conditions, and transformation-then-filtering commutes as expected. Pushing set operations *forward* through a non-injective transformation (image: "the set of customer regions among the affected orders") silently loses the distinctions the transformation collapsed, and intersection-of-images overstates image-of-intersection. For example, two dashboards may each show region EU affected while no single order in *both* incident sets is European. Measure-theoretic probability builds on exactly the well-behaved direction: events pull back through random

variables by preimage (Chapter 14), and Theorem 4.3 is why the pullback of a σ-algebra is a σ-algebra.

4.4 Function spaces, currying, and finite counts

Definition 4.6 (Function space). B^A denotes the set of all functions from A to B.

The exponent notation is earned: for finite sets, $|B^A| = |B|^{|A|}$, since each of $|A|$ arguments is independently assigned one of $|B|$ values. With $B = \{0,1\}$ this count recovers $|\mathcal{P}(A)| = 2^{|A|}$ via characteristic functions (Theorem 2.2). We use the identification $\mathcal{P}(A) \cong \{0,1\}^A$ throughout Part III.

Theorem 4.4 (Currying). For any sets A, B, C there is a canonical bijection $C^{A\times B} \cong (C^B)^A$, sending $f(a,b)$ to $a \mapsto \big(b \mapsto f(a,b)\big)$.

Currying is the formal license for treating a two-argument function as a one-argument function returning a function, the default in functional languages and the reason partial application is meaningful. It also underlies configuration layering: a function from (environment, key) to value *is* a function from environment to key–value maps. The bijection is canonical (definable without arbitrary choices), which is what permits languages to apply it implicitly and bidirectionally without ambiguity.

A second canonical bijection of the same family is used constantly in Part III.

Theorem 4.5 (Adjacency isomorphism). For any sets A, B there is a canonical bijection $\mathcal{P}(A \times B) \cong \big(\mathcal{P}(B)\big)^A$, sending a relation R to the function $a \mapsto \{b \in B : a \mathrel{R} b\}$.

This is Section 3.4's adjacency-list representation promoted to a theorem. "A relation is a set-valued function" is not a slogan but an isomorphism, and Chapter 11 reads it as the equivalence between edge-list and adjacency-map storage.

Example 4.3 (Maps, dictionaries, and total-versus-partial again). The library type `Map<K, V>` denotes a *finite partial* function $K \rightharpoonup V$. It is totalized by `getOrDefault` (codomain trick), guarded by `containsKey` (domain trick), or exposed as `Optional<V>` (codomain trick, typed). An array of length n is a total function $\{0, \dots, n-1\} \to V$ with the domain enforced by bounds checking. An out-of-bounds access is an appeal to a value outside the function's domain, and languages differ only in how loudly the appeal fails.

4.5 Sequences, indexed families, and choice functions

Definition 4.7 (Sequence). A *finite sequence* over B of length n is a function $s : [n] \to B$ (equivalently, with domain $\{0, \dots, n-1\}$; this book indexes from 1

in mathematics and notes zero-based indexing where code is concerned). An *infinite sequence* is a function $\mathbb{N} \to B$. An *indexed family* $(A_i)_{i \in I}$ (Definition 2.5) is officially a function $I \to \mathcal{P}(\mathcal{U})$.

Sequences resolve the deficiency extensionality imposed on sets: order and repetition. The tuple (a, b) of Chapter 2 and the sequence of length 2 are formally different encodings of the same abstraction. As with Kuratowski pairs, no result depends on the difference.

Definition 4.8 (Choice function). For a family $(A_i)_{i \in I}$ of nonempty sets, a *choice function* is $c: I \to \bigcup_i A_i$ with $c(i) \in A_i$ for every i. The general product $\prod_{i \in I} A_i$ is the set of all choice functions.

For finite I, choice functions exist by finite induction, and $\prod$ generalizes the finite Cartesian product. Whether a choice function exists for *every* infinite family of nonempty sets is precisely the Axiom of Choice. We defer it to Chapter 8 but flag it here so that the reader sees how innocuously it arises. "Pick one representative from each shard" is a choice function: unproblematic when a selection rule exists (least element, lowest ID), axiomatic when one must merely *exist*.

4.6 Summary

A function is a total, single-valued relation, officially carrying its codomain. Partial functions are the

computational norm, totalized or guarded by explicit convention. Injectivity, surjectivity, and bijectivity are the no-collision, full-coverage, and faithful-correspondence properties. They are preserved by composition, certified for the relevant stage by a successful composite (though not located by a failed one), and characterized by inverses, with one-sided inverses capturing the round-trip laws of serialization. Preimage respects all Boolean structure while image does not. This asymmetry has daily consequences in data work and becomes the foundation of measurability in Chapter 14. Function spaces obey exponential arithmetic, currying canonically reshapes them, and choice functions name (and expose the axiomatics of) “pick one from each.”

Chapter 5. Equivalence and Order

Dependencies: Chapters 1–4. Two families of relations organize almost all classification and comparison in applications: equivalences, which say "same for present purposes," and orders, which say "before" or "below." In this chapter, we develop both, their canonical constructions (quotients, Hasse diagrams, topological sorts), and the lattice structure that Part III uses repeatedly.

5.1 Equivalence relations

Definition 5.1 (Equivalence relation). A relation $\sim$ on A is an *equivalence relation* iff it is reflexive, symmetric, and transitive.

Definition 5.2 (Equivalence class, quotient). For an equivalence $\sim$ on A and $a \in A$, the *equivalence class* of a is $[a] = \{x \in A : x \sim a\}$. The **quotient set** $A/{\sim} = \{[a] : a \in A\}$ is the set of all classes, and the *canonical projection* $\pi : A \to A/{\sim}$ sends $a \mapsto [a]$.

Theorem 5.1 (Fundamental theorem of equivalence relations). Let $\sim$ be an equivalence on A. Then (i) $a \sim b$ iff $[a] = [b]$; (ii) any two classes are equal or disjoint; (iii) $A/{\sim}$ is a partition of A. Conversely, every partition $\{B_i\}$ of A arises this way from exactly one equivalence, namely $a \sim b$ iff a, b lie in the same block.

Theorem 5.1 is the license behind every "group by." *Classifying* (imposing a partition) and *identifying* (declaring an equivalence) are the same act viewed from two sides. The quotient set is the mathematical object that a `GROUP BY` clause, a clustering, a sharding scheme, or a `HashMap<K, List<V>>` produced by grouping all *are*: its elements are the groups, and the projection π is "assign each item its group."

Example 5.1 (Equivalences chosen, not found). Consider the same set of strings. Byte equality, Unicode-normalized equality, case-insensitive equality, and "same after trimming whitespace" are four equivalences, each coarser than the last except where incomparable. A deduplication pipeline must *choose* one and say which. Consider software versions. "Same major.minor" and "same major" are two equivalences with the second coarser, and semantic-versioning compatibility policies are statements about which quotient the API contract respects. Consider floating-point values. "Differs by less than ε" is famously *not* an equivalence (Section 3.3: transitivity fails). It is a *tolerance relation* (reflexive and symmetric), and an "approximate grouping" built on it has three honest options. It may pick canonical representatives first and group by them. It may scan the data and attach each value to the first group it is close to, which produces a partition of the dataset that can depend on traversal order and need not coincide with the connected-component partition of the closeness graph (with

tolerance 1 and first-fit grouping by a group's first member, the values 0,0.75,1.5 yield {{0,0.75}, {1.5}} in one order and {{0,0.75,1.5}} in another). Or it may take the connected components of the closeness graph, that is, the classes of the equivalence generated by the tolerance relation (its reflexive–symmetric–transitive closure, Definition 3.6). The third option is order-independent for a fixed dataset and does yield a partition, at a different price: two members of one component need not themselves be ε-close, since closeness chains. The flaky data jobs of this genre are the ones that intended the third reading and implemented the second, or that reported either output as if closeness itself were transitive.

Definition 5.3 (Refinement). Equivalence $\sim_1$ *refines* $\sim_2$ (is *finer*; $\sim_2$ is *coarser*) iff $a \sim_1 b \Rightarrow a \sim_2 b$; equivalently, every $\sim_1$-class is contained in a $\sim_2$-class.

Refinement organizes hierarchies of granularity. Request, endpoint, service, team: this chain is a sequence of successively coarser partitions of the request stream, and a metrics rollup is a passage to a coarser quotient. Chapter 16's rough sets measure how well a target set can be expressed in a given partition's granularity.

Definition 5.4 (Well-defined on the quotient). Given $f : A \to C$ and an equivalence $\sim$ on A, f

descends to the quotient iff $a \sim b \Rightarrow f(a) = f(b)$; then $\bar{f}: A/\sim \to C, \bar{f}([a]) = f(a)$, is **well-defined**.

The compatibility check is the substance of the common phrase "well-defined." Applied instances abound, and two constructions must be kept apart. First of all, an element-level attribute descends only if it is constant on each class: "the user's country" is a legitimate group attribute only if country agrees across the group, and if two equivalent records carry values 2 and 3 then their value function does not descend at all. Then there are aggregates, which are functions defined directly on classes: for finite classes, $G(C) = \sum_{a \in C} v\,(a)$ is a function on the quotient by construction. Its representative-based form $a \mapsto G([a])$ is constant on classes and so descends trivially. The group sum is independent of the representative because it consumes the whole class, not because the element values agree. A function on canonical forms is legitimate iff canonicalization respects the equivalence. Checking well-definedness before descending is the difference between a group-level fact and an artifact of arbitrary representative choice. In reporting pipelines, this distinction has audit consequences.

Definition 5.5 (Kernel). The *kernel* of $f: A \to B$ is the equivalence $\ker f$ on A given by $a_1 \sim a_2$ iff $f(a_1) = f(a_2)$.

Theorem 5.2 (Canonical factorization). Every function $f: A \to B$ factors as $f = \iota \circ \bar{f} \circ \pi$, where

$\pi: A \to A/\ker f$ is the projection (surjective), $\bar{f}: A/\ker f \to f[A]$ is a bijection, and $\iota: f[A] \hookrightarrow B$ is the inclusion (injective).

Therefore, every data transformation splits into a lossy classification, a faithful renaming, and a widening. This anatomy is worth performing on any pipeline stage. What does this step conflate (its kernel), and what can it never produce (the complement of its image)? The two questions bound what any downstream consumer can recover.

5.2 Partial orders

Definition 5.6 (Partial order, poset). A relation $\leq$ on A is a *partial order* iff it is reflexive, antisymmetric, and transitive; the pair $(A, \leq)$ is a *poset*. A *strict order* $<$ is irreflexive and transitive (asymmetry follows); the two presentations interconvert by adding or removing id_A. Elements a, b are *comparable* iff $a \leq b$ or $b \leq a$; an order is **total** (*linear*) iff all pairs are comparable. A **chain** is a subset totally ordered by $\leq$; an **antichain** is a subset of pairwise incomparable elements.

Example 5.2 (The standard applied posets). (a) $(\mathcal{P}(U), \subseteq)$: the subsets of a universe under inclusion, the ambient poset of Parts I–III, total only when $|U| \leq 1$. (b) Divisibility on positive integers. (c) Version vectors and vector clocks in distributed systems: $(t_1, \ldots, t_k) \leq (s_1, \ldots, s_k)$ componentwise. Here incomparability *is* concurrency; the whole

point of the construction is that some event pairs are ordered by neither happening-before the other. (d) Task dependency: $t_1 \leq t_2$ iff $t_1 = t_2$ or t_1 must complete before t_2 starts (the reflexive–transitive closure of direct dependency, Theorem 3.2, assuming acyclicity; "must complete before" alone is the strict order). (e) Type subtyping $S <: T$ in programming languages, which on type expressions is in general a *preorder* (reflexive and transitive): mutually substitutable but syntactically distinct types must be identified, that is, quotiented by the equivalence $S <: T \wedge T <: S$, before antisymmetry holds and a partial order results. In each case, pretending the order is total (assigning every pair a winner) destroys exactly the information the poset was built to carry. Forcing concurrent events into a sequence, or incomparable versions into a line, is not a harmless convenience but a semantic claim, and occasionally the wrong one.

Definition 5.7 (Extremes). In a poset $(A, \leq)$ with $S \subseteq A$: $m \in S$ is a **maximal** element of S iff no $s \in S$ has $m < s$; it is the **greatest** element (maximum) of S iff $s \leq m$ for all $s \in S$. (*Minimal* and *least* dually.) An **upper bound** of S is any $u \in A$ with $s \leq u$ for all $s \in S$; the **least upper bound** (*join*, *supremum*) $\bigvee S$, when it exists, is an upper bound below all upper bounds; dually the **greatest lower bound** (*meet*, *infimum*) $\bigwedge S$.

Maximal and greatest coincide in total orders and diverge in partial ones, a distinction applications get

wrong at a steady rate. A dependency graph typically has many maximal ("nothing depends on me") packages and no greatest one. A search for "the latest compatible version set" may have several maximal solutions with no best one. A Pareto frontier is precisely the set of maximal elements under the componentwise order, and reporting "the optimum" of a multi-objective problem presumes a greatest element that generally does not exist. When engineering discourse says "the latest," a poset-literate listener asks: greatest, or merely maximal, and under which order?

Definition 5.8 (Lattice). A poset in which every *pair* has a join and a meet is a **lattice**; if every *subset* has both, it is a **complete lattice**.

Theorem 5.3. $(\mathcal{P}(U), \subseteq)$ is a complete lattice with $\bigvee = \bigcup$ and $\bigwedge = \bigcap$; its greatest element is U and least is $\emptyset$.

Lattices are the algebra of "most permissive common restriction" and "least permissive common extension": meets of permission sets, joins of capability grants, meet-over-all-paths in program static analysis, and the security lattices of information-flow control (low/high classification levels with joins tracking taint). Chapter 9 axiomatizes the special case with complements, Boolean algebra, and Chapter 12's refinement calculus lives in the lattice of specifications.

5.3 Order morphisms, Hasse diagrams, and topological sorting

Definition 5.9 (Monotone map). $f : (A, \leq_A) \to (B, \leq_B)$ is *monotone* (order-preserving) iff $a_1 \leq_A a_2 \Rightarrow f(a_1) \leq_B f(a_2)$. An *order isomorphism* is a monotone bijection with monotone inverse.

Monotonicity is a pervasive applied contract, and stating it requires naming both orders. Cache layers assume "more specific query, fewer results": if $q_1 \leq q_2$ means that q_2 is at least as restrictive as q_1, the assumption is $\text{results}(q_2) \subseteq \text{results}(q_1)$, an *order-reversing* (antitone) map from queries to $(\mathcal{P}(\text{rows}), \subseteq)$, or equivalently a monotone map into the opposite order $\supseteq$. Pricing tiers assume "more usage, no less cost." Index structures assume the sort key is monotone in the comparison the application performs. Declared monotonicity that fails on edge cases (discount cliffs, Unicode collation anomalies) produces the characteristic bug of the genre: a binary search over data that is not actually sorted by the search's comparator. The behavior of such a search is not "slightly wrong" but unspecified.

Definition 5.10 (Covering, Hasse diagram). In a poset, *b covers a* iff $a < b$ and no c has $a < c < b$. The **Hasse diagram** of a finite poset draws its elements with an upward edge for each covering pair; the full order is recovered as the reflexive–transitive closure of the covering relation.

The Hasse diagram is the poset's minimal representation: the least relation whose closure is $\leq$ (its *transitive reduction*, unique for finite posets). Storing covers and closing on demand versus storing the closed relation is the standard space/time trade of Section 3.4, now with a canonical minimum.

Theorem 5.4 (Topological sorting). Every finite poset $(A, \leq)$ admits a *linear extension*: a total order $\preccurlyeq$ on A with $\leq \subseteq \preccurlyeq$. Equivalently, every finite directed acyclic graph admits a topological sort.

Build systems, package installers, task schedulers, and spreadsheet recalculation engines all implement Theorem 5.4. Two of its features do daily work. First of all, existence requires acyclicity. The theorem is why cycle detection is the first pass of such tools, and why a cycle is a *fatal* error: no execution order can respect the constraints. Moreover, non-uniqueness is equally consequential. A poset with incomparable elements has multiple linear extensions (their number can be exponential; counting them is a hard problem), so "the build order" is a fiction. Any two runs may legitimately differ wherever the order leaves freedom. This freedom is both the license for parallel execution (antichains may run concurrently) and the origin of order-dependent flakiness when tasks covertly communicate outside the declared dependencies. A reproducible-build discipline is, in this vocabulary, the decision to fix one canonical linear extension.

Example 5.3 (Scheduling with widths). Fix an execution model: tasks are indivisible, each runs on one worker, and the declared dependencies are respected. If every task takes one unit of time (or durations share a positive lower bound δ, which scales the bound by δ), the maximum chain length bounds the elapsed completion time from below: no schedule, however parallel, beats it. With unequal durations the same argument gives the *critical path*, the maximum over precedence chains of the sum of task durations along the chain. By Dilworth's theorem (Dilworth 1950), the maximum antichain size equals the minimum number of chains covering the poset. Under the stated model it bounds the number of tasks that can ever run simultaneously, and hence the number of workers that can be useful. Both bounds are statements about the poset alone, obtained before any scheduler runs, and both depend on the execution model: a single task that itself spreads across several workers is outside the model, and so is a task that can be split.

5.4 Well-orders, briefly

Definition 5.11 (Well-order). A total order on A is a *well-order* iff every nonempty subset of A has a least element.

$(\mathbb{N}, \leq)$ is well-ordered. $(\mathbb{Z}, \leq)$ is not (the whole set has no least element), and neither is $([0,1], \leq)$ despite having least element 0: the nonempty subset $\{1, 1/2, 1/3, \dots\}$ has no least member. Well-ordering

is what makes proof by minimal counterexample and definition by recursion sound on $\mathbb{N}$. Chapter 7 extends both past the finite via ordinals, and Chapter 8 confronts the Well-Ordering Theorem's equivalence with the Axiom of Choice.

5.5 Summary

Equivalences and partitions are one concept with two presentations (Theorem 5.1). Quotients are the objects "group by" computes. Well-definedness governs descent to representatives, and every function factors through its kernel as lossy classification, faithful bijection, widening. Partial orders model precedence and refinement without forcing comparability. Maximal differs from greatest. Lattices supply canonical meets and joins, with $(\mathcal{P}(U), \subseteq)$ the complete archetype. Finite posets are minimally presented by Hasse diagrams and linearized by topological sort, which exists iff the relation is acyclic and is unique almost never, while chains and antichains bound latency and parallelism. Well-orders prepare the ground for the transfinite.

Part II: The Infinite

Chapter 6. Counting and Cardinality

Dependencies: Chapters 1–5. In this chapter, we make "same size" precise via bijections, separate the countable from the uncountable, present Cantor's theorem and the Cantor–Schröder–Bernstein theorem, and draw the applied consequences: what diagonalization says about computation, and what uncountability says about what can be named, stored, or enumerated.

6.1 Equinumerosity

How can we compare the sizes of sets without counting, which fails for infinite sets? Cantor answered: pair the elements off.

Definition 6.1 (Equinumerosity). Sets A and B are *equinumerous*, written $A \approx B$, iff there exists a bijection $A \to B$. We write $A \preccurlyeq B$ iff there exists an injection $A \to B$, and $A \prec B$ iff $A \preccurlyeq B$ and $A \not\approx B$.

Equinumerosity is reflexive (id_A), symmetric (inverses, Theorem 4.2), and transitive (composites, Theorem 4.1). Therefore, it behaves as an equivalence, though on the proper class of all sets rather than a set. "Cardinality" names its abstraction.

Definition 6.2 (Cardinality, working form). $|A| = |B|$ means $A \approx B$; $|A| \leq |B|$ means $A \preccurlyeq B$. For finite

sets, $|A| = n$ means $A \approx [n]$ (by definition, together with the pigeonhole fact that $[m] \approx [n]$ forces $m = n$, so finite cardinality is well-defined). A set is **finite** iff $A \approx [n]$ for some $n \in \mathbb{N}$, and **infinite** otherwise.

Note. What object "$|A|$" *is* for infinite A can be answered properly (initial ordinals: Chapter 7's ordinals plus Section 8.3's well-ordering) or dodged harmlessly. Every statement in this chapter is a statement about the existence of injections and bijections, and the reader may read $|A| \leq |B|$ purely as such. We state facts about the comparisons, not about the abstracted objects.

Theorem 6.1 (Cantor–Schröder–Bernstein). If $A \preccurlyeq B$ and $B \preccurlyeq A$, then $A \approx B$ (Enderton 1977, Chapter 6).

The theorem's applied value is methodological. To establish a bijection, we may exhibit two injections, which is almost always easier. The theorem converts "same size" from a construction problem into two containment-style arguments: the double-inclusion discipline of Chapter 1, lifted to sizes.

Theorem 6.2. $\preccurlyeq$ is reflexive and transitive, and (by Theorem 6.1) induces a partial order on cardinalities. Whether it is *total* (any two sets comparable) is exactly the Axiom of Choice (Chapter 8).

6.2 Countable sets

Definition 6.3 (Countable). A set A is *countably infinite* iff $A \approx \mathbb{N}$, and **countable** iff it is finite or countably infinite; otherwise **uncountable**. The cardinality of $\mathbb{N}$ is written $\aleph_0$.

Countability is enumerability in principle: a set is countable iff its elements can be arranged in a (possibly finite) list $a_0, a_1, a_2, \ldots$ with every element appearing at some finite index. Three notions must be kept apart here, because the applied chapters lean on the distinctions. *Countability* asserts that such a list exists as a mathematical object. *Computable enumerability* asserts that a program can produce the list: every subset of $\mathbb{N}$ is countable, but only countably many subsets are computably enumerable, because only countably many programs exist (Section 6.3 makes this count). *Streamability* in practice asserts further that the list can be produced in acceptable time and space, item by item. Countability is necessary for the other two and sufficient for neither. With that caveat, the closure properties below are the reasons streaming architectures compose: they show that the countable sets are closed under the constructions such architectures use.

Theorem 6.3 (Closure properties). (i) Any subset of a countable set is countable. (ii) $\mathbb{N} \times \mathbb{N} \approx \mathbb{N}$; hence a finite product of countable sets is countable. (iii) A countable union of countable sets is countable.* (iv)

$\mathbb{Z}$ and $\mathbb{Q}$ are countable. (v) The set A^* of finite sequences over a countable alphabet A is countable.

*The asterisk on (iii): enumerating each A_i requires choosing one enumeration per set, infinitely many choices at once. With the enumerations *given* (the usual applied situation: each shard arrives with its own index) the construction is choice-free. In the abstract, (iii) uses a weak form of the Axiom of Choice. This is the book's first sighting of choice doing real work, duly flagged per Chapter 8.

Clause (v) has a consequence worth stating in applied terms before any diagonal argument makes it poignant: **the set of all programs is countable.** A program in any fixed language is a finite string over a finite (or countable) alphabet. So too is every JSON document, every SQL query, every finite proof, every URL, every filename, every message. Everything writable is countable, and everything *finite and discrete* fits in one $\aleph_0$. The set of all *infinite* binary streams, by contrast, is uncountable (Theorem 6.5), which is the first hint that "finite strings" and "streams" are not interchangeable idealizations.

Example 6.1 (Gödel numbering as engineering). Clause (v)'s enumeration is implemented daily. Serialization flattens structured values to byte strings, and byte strings are integers in base 256. Interning tables and enum encodings exploit the corollary that any countable universe can be indexed by $\mathbb{N}$. Injectivity of the encoding (no two values

sharing an index) is the property purchased by escaping and length-prefixing. Its failure is the classic injection-attack family: two structurally different values whose encodings collide. Content-addressable stores belong to this family only in part. Their digests are of fixed width, and a fixed-width digest cannot injectively name an unbounded countable universe of inputs (Section 6.5's pigeonhole). What such a store relies on is not injectivity but the practical unlikelihood of a collision, which Chapter 13 prices.

6.3 Uncountable sets: diagonalization

Theorem 6.4 (Cantor). For every set A: $A \prec \mathcal{P}(A)$. In particular there is no surjection $A \to \mathcal{P}(A)$ (Enderton 1977, Chapter 6).

Theorem 6.5. $\mathbb{R}$ is uncountable; indeed $\mathbb{R} \approx \mathcal{P}(\mathbb{N}) \approx \{0,1\}^{\mathbb{N}}$.

Therefore, there are at least two magnitudes of infinity. In fact, there is an endless ascent, since Theorem 6.4 iterates: $\mathbb{N} \prec \mathcal{P}(\mathbb{N}) \prec \mathcal{P}\big(\mathcal{P}(\mathbb{N})\big) \prec \cdots$. The cardinality of $\mathbb{R}$ (equivalently $\mathcal{P}(\mathbb{N})$) is the **continuum**, written $2^{\aleph_0}$ or $\mathfrak{c}$.

Example 6.2 (What diagonalization means for computing). We combine the countability of programs with Theorem 6.4.

Uncomputable functions exist, in overwhelming abundance. Functions $\mathbb{N} \to \{0,1\}$ form a set of size $2^{\aleph_0}$. Programs number $\aleph_0$. Therefore, all but countably many such functions are computed by no program whatsoever. This is a pure counting argument, and it names no specific uncomputable function. Naming one is Turing's sharper achievement: the halting function is uncomputable by an argument that transposes Theorem 6.4's diagonal construction into software. We feed the purported decider a program built to do the opposite of whatever is predicted of it.

Most reals are unnameable. Every real number *describable by any finite text in any fixed formal language* falls in a countable set. The rest, almost all of $\mathbb{R}$, admit no finite description. The applied corollary: every value ever stored in a floating-point register, database, or file is drawn from a countable set of representable values, a set that is finite once a fixed storage bound is imposed (a 64-bit register has 2^{64} bit patterns) and countably infinite without one (files of arbitrary finite length already encode every natural number). Numerical computing is the art of living in the sliver while reasoning about the whole. Chapter 14 meets the same gap as the reason probability needs measure theory rather than counting.

Note. Diagonalization is constructive and finitary in each instance (no appeal to choice or to completed infinities beyond the sets in play), and it is *robust*. It

relativizes (Turing degrees), it resource-bounds (the time-hierarchy theorems of complexity theory are diagonal arguments with stopwatches), and it recurs as Gödel incompleteness (diagonalizing provability). One argument, many theaters. Recognizing its silhouette is a transferable skill.

6.4 Cardinal arithmetic, briefly

For cardinals $\kappa = |A|$, $\lambda = |B|$ with A, B disjoint: $\kappa + \lambda = |A \cup B|$, $\kappa \cdot \lambda = |A \times B|$, $\kappa^\lambda = |A^B|$ (functions $B \to A$; note the direction). These operations are well-defined (independent of representatives) and, on finite cardinals, agree with school arithmetic. For infinite cardinals, addition and multiplication trivialize: for infinite κ (and $0 < \lambda \leq \kappa$), $\kappa + \lambda = \kappa \cdot \lambda = \kappa$ (for $\kappa = \aleph_0$ this is Theorem 6.3(ii); the general case uses Chapter 8's machinery). However, **exponentiation is where the action is**: Theorem 6.4 is the statement $\kappa < 2^\kappa$.

The continuum problem. Is there a cardinality strictly between $\aleph_0$ and $2^{\aleph_0}$? Cantor's *continuum hypothesis* (CH) says no. Its status is one of the most instructive facts in all of mathematics. Gödel (1940) showed that CH cannot be *refuted* from ZFC. Cohen (1963) showed that it cannot be *proved*. Jointly, CH is **independent** of the standard axioms (assuming their consistency; Jech 2003, Chapters 13–14; Kunen 1980, Chapters VI–VII). This is not a statement of human ignorance but a theorem *about the axiom system*: ZFC simply does not decide the question. The

mathematician's options (adopt new axioms, study both extensions, or treat the question as underdetermined) are all live positions in current practice. For this book's purposes, CH is the standing illustration that "true," "provable," and "provable from these axioms" must be kept distinct, the sharpest possible warrant for this book's habit of stating epistemic status.

6.5 Finite pigeonholes, one level up

The finite shadow of this chapter is the **pigeonhole principle**: if $|A| > |B|$ then no injection $A \to B$ exists. Any assignment of $|A|$ items to $|B|$ boxes puts two items in one box (formally: for finite sets, an injection $A \to B$ forces $|A| \leq |B|$). Its applied force is that it is *non-constructive at zero cost*. A 64-bit hash over more than 2^{64} possible inputs has collisions, certainly, exhibited or not. A lossless compressor that shortens some file lengthens another (there are 2^n binary strings of length n and only $2^n - 1$ strings of length less than n, so no injection of the former into the latter exists). A load balancer over n servers receiving $n + 1$ simultaneous requests co-locates two. Chapter 13 quantifies the principle (how *many* collisions; how soon, via the birthday bound), and Chapter 14 makes the "how soon" probabilistic.

6.6 Summary

Size is bijection, and comparison is injection. Cantor–Schröder–Bernstein makes two injections a

bijection, and comparability in general awaits choice. Countability (closure under subsets, finite products, countable unions, finite sequences) is the cardinality of everything finite, discrete, and writable, programs included; it is weaker than computable enumerability, which is weaker again than practical streamability. Diagonalization separates $\mathcal{P}(A)$ from A, makes $\mathbb{R}$ uncountable, and, transposed to computation, establishes uncomputability and unnameability by counting alone. Cardinal exponentiation carries the structure. CH's independence from ZFC fixes the book's epistemic vocabulary. Pigeonhole is the finite residue, free of charge, quantified in Part III.

Chapter 7. Ordinals, Induction, and Recursion

Dependencies: Chapters 1–6, especially well-orders (Definition 5.11). In this chapter, we build the natural numbers inside set theory, extract the twin engines of induction and recursion from well-foundedness, and extend both into the transfinite. The applied payoffs are the termination arguments, fixpoint constructions, and hierarchy-ranking techniques used across program analysis and verification.

7.1 The natural numbers as sets

Set theory earns its foundational keep by *constructing* the number systems rather than presupposing them. The construction of $\mathbb{N}$, due to von Neumann, defines each number as the set of its predecessors:

$$0 = \emptyset, \quad 1 = \{0\} = \{\emptyset\}, \quad 2 = \{0,1\}, \quad 3 = \{0,1,2\}, \quad \ldots$$

Definition 7.1 (Successor). $S(x) = x \cup \{x\}$.

Definition 7.2 (Inductive set; $\mathbb{N}$). A set I is *inductive* iff $\emptyset \in I$ and $x \in I \Rightarrow S(x) \in I$. The Axiom of Infinity (Chapter 8) guarantees that at least one inductive set exists; $\mathbb{N}$ is defined as the intersection

of all inductive subsets of any fixed one, that is, the *least* inductive set.

The definition by least closure should look familiar. It is the same "smallest set containing the seeds and closed under the rules" pattern as the transitive closure (Theorem 3.2), and it recurs wherever a language, a datatype, or a state space is defined inductively: the well-formed formulas of Chapter 9, the reachable states of a protocol, an algebraic data type in a functional language. Minimality is not decorative. It is exactly what makes induction valid.

Theorem 7.1 (Mathematical induction). If $P(0)$ and $\forall n \in \mathbb{N}\ \Big(P(n) \Rightarrow P\big(S(n)\big)\Big)$, then $\forall n \in \mathbb{N}\, P(n)$.

In other words, induction is the *definition* of $\mathbb{N}$ read back as an inference rule. Likewise, structural induction on any inductively defined set is that set's minimality read back. The reader who internalizes this equation ("no junk in the datatype" = "induction is sound on it") owns the correctness argument for the recursive tree-walks they will write, together with its standard failure: on a structure admitting values outside the inductive closure (cyclic "trees" built by mutation), structural induction, and the recursive procedure whose termination it justifies, is unsound. This is why cycle detection guards production tree-walkers.

Each von Neumann natural is a *transitive set* (every element of it is also a subset of it) strictly well-

ordered by $\in$, in the sense that $\in$ restricted to it is a strict order (Definition 5.6; it is irreflexive by Foundation, Chapter 8) whose associated non-strict order "$\in$ or $=$" is a well-order. On $\mathbb{N}$, the relations $m \in n$, $m \subsetneq n$, and $m < n$ coincide. Order is thus not additional structure bolted onto the numbers but membership itself. Moreover, $n = \{0, \dots, n-1\}$ makes the combinatorial convention $[n]$ (Section 2.5) nearly official: the number *is* the canonical n-element set, the one every zero-indexed array in every programming language silently uses as its index domain.

7.2 Recursion

Induction proves; recursion *builds*. The two are the same phenomenon in different modes, and the following theorem (used implicitly by every recursive definition since) makes the building legitimate.

Theorem 7.2 (Recursion on $\mathbb{N}$). For any set X, element $x_0 \in X$, and function $g\colon X \times \mathbb{N} \to X$, there is exactly one function $f\colon \mathbb{N} \to X$ with $f(0) = x_0$ and $f\big(S(n)\big) = g(f(n), n)$ for all n.

The construction standardly used to establish Theorem 7.2 (assembling f as the union of all coherent finite approximations) is itself an applied idea. It is how we show that a stream computation, defined step by step, determines a unique total behavior, and its transfinite extension below is how

the semantics of iteration and recursion are constructed in program analysis. Two corollaries of Theorem 7.2 follow immediately. First, addition, multiplication, and exponentiation on $\mathbb{N}$ exist and are unique once their recursions are written down. Second, *any* two structures satisfying the set-based Peano conditions are isomorphic (build the map by Theorem 7.2, then verify bijectivity by induction). The conditions are these: a distinguished zero; an injective successor whose image excludes zero; and induction for *every subset* of the carrier (a subset containing zero and closed under successor is the whole carrier). All three are needed. A one-element structure whose successor fixes its only element has a zero, an injective successor, and induction, but zero is a successor there, and the structure is not $\mathbb{N}$. This is the sense in which "the" natural numbers are unique, and the reason no one need care whether the implementation is von Neumann's. The uniqueness belongs to the set-based (second-order) characterization, where induction ranges over all subsets (Enderton 1977, Chapter 4). First-order Peano arithmetic, whose induction scheme ranges only over definable properties, has nonstandard models (Boolos, Burgess, and Jeffrey 2007, chapter on nonstandard models), and the distinction returns in Section 7.5.

7.3 Well-founded induction

Both engines generalize far beyond $\mathbb{N}$, and the right level of generality is the well-founded relation.

Definition 7.3 (Well-founded relation). R on A is *well-founded* iff every nonempty $S \subseteq A$ has an R-minimal element: some $m \in S$ with no $s \in S$ satisfying s R m. Equivalently (the equivalence uses a weak choice principle in one direction): there is no infinite descending chain $\cdots$ R a_2 R a_1 R a_0.

Theorem 7.3 (Well-founded induction). Let R be well-founded on A. If for every $a \in A$, $\Big(\forall b \,(b$ R $a \Rightarrow P(b))\Big) \Rightarrow P(a)$, then $\forall a \in A\, P(a)$.

Well-founded induction has no base-case clause. The base cases are the R-minimal elements of A, for which the hypothesis quantifies over an empty set of predecessors and must be discharged outright. The applied translation is the **termination argument**, and it needs one orientation fixed carefully. Let $\rightarrow$ be a system's transition relation on states. Definition 7.3 forbids infinite *descending* chains, whereas an execution $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots$ runs forward: the relation $n \rightarrow n+1$ on $\mathbb{N}$ is well-founded in the predecessor sense yet has an infinite run. The relation that matters is the reversed one, $s' \prec s$ iff $s \rightarrow s'$. Here termination means *universal* termination: every maximal run from an allowed initial state is finite, where a run is maximal if it is infinite or ends in a *terminal* state, one with no outgoing transition. (For a nondeterministic system this is a choice of reading. *Existential* termination from s is the different and weaker property that some finite run from s reaches a terminal state,

$\exists t\,\big(s \to^{*} t \wedge \nexists u\,(t \to u)\big)$; a finite prefix of a run establishes nothing by itself, since the single state q with $q \to q$ has finite prefixes of every length and terminates in neither sense. A state with one step to a terminal state and another step back to itself terminates existentially but not universally; rewriting theory calls the two properties weak and strong normalization.) A system terminates in this sense iff $\prec$ restricted to the reachable states is well-founded, with the choice remark of Definition 7.3 applying in one direction. Equivalently, one says that $\to$ is *terminating* or *conversely well-founded*. The universal method for establishing this property is to exhibit a *measure*: a map μ from states into a known well-founded order (usually $(\mathbb{N}, <)$, sometimes lexicographic tuples, occasionally genuine ordinals) with every transition $s \to s'$ strictly decreasing μ, that is, $\mu(s') < \mu(s)$. Loop variants in Hoare logic and Dafny, fuel parameters, and "the recursion is on a smaller subterm" are all instances. The lexicographic order on $\mathbb{N}^k$ is well-founded (a nonempty subset has a least first coordinate, then a least second among those, and so on) but *not* order-isomorphic to $\mathbb{N}$ for $k \geq 2$. This is a first sign that well-founded orders come in essentially different lengths, which is exactly what ordinals measure.

7.4 Ordinals

Definition 7.4 (Ordinal). An *ordinal* is a transitive set *strictly well-ordered* by $\in$: membership restricted

to it is a strict total order (irreflexive, transitive, and any two distinct elements comparable) whose non-strict companion "$\in$ or $=$" is a well-order in the sense of Definition 5.11. Under the Axiom of Foundation (Chapter 8), any transitive set on which $\in$ is a strict total order is an ordinal, since Foundation supplies the well-foundedness. Ordinals are conventionally denoted α, β, γ; the class of all ordinals is Ord.

Every von Neumann natural is an ordinal, and so is $\mathbb{N}$ itself, viewed as the set of all naturals; as an ordinal it is renamed ω. The successor operation continues past it: $S(\omega) = \omega \cup \{\omega\} = \omega + 1$, then $\omega + 2$, and so on. After all of those comes $\omega + \omega = \omega \cdot 2$, then eventually ω^2, ω^ω, and beyond. Four structural facts follow, stated here and established in the standard references (Jech 2003, Chapter 2; Enderton 1977, Chapter 7; their demonstrations are instructive but belong to a foundations course):

Theorem 7.4. (i) Every element of an ordinal is an ordinal, and $\alpha \in \beta$ iff $\alpha \subsetneq \beta$; one writes $\alpha < \beta$ for either, and $\alpha \leq \beta$ for "$\alpha < \beta$ or $\alpha = \beta$." (ii) Any two ordinals are comparable, and every nonempty class of ordinals has a $<$-least element: Ord is strictly well-ordered by $<$ (in the class sense). (iii) **Every well-ordered set is order-isomorphic to exactly one ordinal** (its *order type*). (iv) Ord is a proper class: no set contains all ordinals (Burali-Forti).

Clause (iii) is the point of the whole construction. Ordinals are the canonical measuring sticks for well-orders, exactly as cardinals are for raw size. Distinct ordinals can share a cardinality (ω, $\omega + 1$, and ω^ω are all countable) because order type remembers *arrangement*, not just quantity: $\omega + 1$ (all the naturals, then one more element after all of them) is a different shape of queue than ω, though both hold countably many. Clause (iv) is Russell's lesson recurring: the collection of all measuring sticks is too big to be a stick.

Definition 7.5 (Successor and limit). An ordinal α is a *successor* iff $\alpha = \beta + 1$ for some β, and a **limit** iff $\alpha \neq 0$ and α is not a successor (equivalently, $\alpha \neq 0$ and $\alpha = \bigcup\alpha$: it is the supremum of its predecessors; the condition $\alpha \neq 0$ must be kept, because $0 = \bigcup 0$ as well). ω is the least limit ordinal.

Theorem 7.5 (Transfinite induction and recursion). Induction: if P holds at 0, is preserved by successor, and holds at each limit whenever it holds at all smaller ordinals, then P holds at every ordinal. Recursion: a function on the ordinals may be defined by specifying its value at 0, at $\alpha + 1$ in terms of the value at α, and at limits in terms of all earlier values (typically a union or supremum); such a function exists and is unique on any ordinal domain.

7.5 Transfinite methods in computation

Ordinals beyond ω can look like theology from a machine room. They are not. They arise on their own wherever an iterative process must be run "to stability, then possibly restarted," and the three uses below are working tools.

Fixpoint iteration. Let $F\colon \mathcal{P}(U) \to \mathcal{P}(U)$ be monotone ($X \subseteq Y \Rightarrow F(X) \subseteq F(Y)$). We define by transfinite recursion $X_0 = \emptyset$, $X_{\alpha+1} = F(X_\alpha)$, $X_\lambda = \bigcup_{\alpha<\lambda} X_\alpha$ at limits. The chain is increasing, and by a cardinality argument it stabilizes at some ordinal, whose value is the *least fixpoint* of F: the Knaster–Tarski theorem (Tarski 1955) in iterative form. A "keep applying the rules until nothing changes" computation is this construction: transitive closure (via $F(X) = R \cup (R \circ X)$, whose least fixpoint is R^+), datalog evaluation, type inference constraint solving, reachability, garbage-collection marking. Over a *finite* universe the iteration stabilizes at a finite stage, which is why engineers meet the construction as a mere while-loop. The ordinal bookkeeping earns its keep in infinite or parametrized settings. For example, program analyses over infinite abstract domains use *widening* precisely to force the stabilization that the ordinals alone will not deliver in finite time. It also earns its keep in the semantics of inductive definitions generally: the *closure ordinal* of an operator is a real, occasionally $> \omega$, measure of how much iteration "to stability, then again" a definition needs.

Termination measures beyond $\mathbb{N}$**.** A distinction between existence and accessibility must be drawn first. For a deterministic system that terminates from every reachable state, a natural-number measure always *exists*: the number of steps remaining. For an effectively presented program it is even computable in principle (run the program from the given state and count), but it may be prohibitively expensive, and as a termination *argument* it is circular, since establishing that the count is well-defined is establishing termination. For an abstract system with no effective presentation assumed, it need not be computable at all. Either way it is there. What the ordinals buy is not existence but a *simple, provable* argument. Often a step decreases a major component while resetting a minor one to something arbitrarily large, and no readily expressible natural-valued quantity is seen to decrease. Lexicographic measures (ordinals of the form ω^k and beyond, in bookkeeping terms) handle exactly this situation: "the pair (outstanding files, bytes left in current file) drops lexicographically." Proof assistants accept ordinal-valued variants for the same reason. Ordinal *height* can also be forced outright when the system branches infinitely. Consider a root state from which a single step may move to a counter holding any natural number n, after which the counter counts down to zero. Every run is finite, so the system terminates, yet no natural number bounds the run length from the root, and the well-founded height of the root is exactly ω.

Hydra games and Goodstein sequences are the celebrated pure specimens: processes that provably terminate, where the argument *requires* induction up to the ordinal ε_0, beyond what first-order Peano arithmetic can furnish (Kirby and Paris 1982). Therefore, a termination proof can be a strictly stronger commitment than first-order arithmetic reasoning. This is an epistemically calibrated fact worth knowing when a verifier's "termination checker" is presented as routine.

Rank and hierarchy. Transfinite recursion assigns every set a *rank* (Chapter 8: the least stage V_α of the cumulative hierarchy of which the set is a *subset*, so that it first appears as a *member* one stage later), every well-founded structure a depth, every inductive definition a stratification. Stratified designs in software (layered architectures, dependency ranks, generation numbers in garbage collectors) are finite shadows of the same idea: a well-founded "built after" relation, with the rank function making the well-foundedness manifest and the induction sound.

Note. All ordinals used in mainstream applied mathematics are countable, and almost all termination arguments in verification practice live below ε_0, a minute initial segment of Ord. The transfinite apparatus is not a claim that computers manipulate infinite objects. It is the accounting system that makes "iterate to stability" and "this

recursion bottoms out" into theorems rather than hopes.

7.6 Summary

ℕ is the least inductive set. Induction is its minimality restated, recursion is its constructive twin, and both generalize to any well-founded relation, the mathematical form of every termination argument (with the transition relation reversed, so that "descending" means "later"). Ordinals canonically measure well-orders (each well-order has a unique order type), distinguish arrangement from size, and continue induction and recursion through limit stages. Transfinitely iterated monotone operators yield least fixpoints, the semantics of "apply rules until stable," with finite universes hiding the ordinals inside while-loops. Lexicographic and ordinal measures give simple, provable termination arguments where no usable natural-number measure is at hand, and rank functions turn layered designs into inductively tractable ones.

Chapter 8. The Axioms: ZFC and Choice

Dependencies: Chapters 1–7, whose informal practices this chapter regularizes. Our goal is a working reading of the Zermelo–Fraenkel axioms with Choice: what each says, what work it does, which applied practice it licenses. We also give a calibrated account of the Axiom of Choice and of what axiomatization can and cannot deliver.

8.1 Why axioms

Chapters 1–7 practiced *informal rigor*: precise definitions and precisely stated results, conducted in prose, with set existence handled by good manners (separation, not comprehension). Axiomatization converts the manners into an explicit contract: a first-order language whose only non-logical symbol is $\in$, and a finite list of axioms and axiom schemas from which the theorems follow by ordinary logic. There are three reasons to care, in ascending order of applied relevance. First of all, *safety*: Russell's paradox showed that intuition about set existence cannot be trusted wholesale, and an explicit contract localizes the trust. Then, *interchangeability*: theorem provers (Lean, Isabelle/ZF, Metamath, Mizar) mechanize mathematics against explicit foundations, and reading their kernels requires knowing what the contract says. Finally, *calibration*: only relative to a fixed axiom system do the independence

phenomena of Chapters 6 and 7 (CH undecidable, Goodstein termination unprovable in weaker systems) even make sense. An axiom system is an *interface specification for mathematics*. In this chapter, we read the one in production use.

8.2 The ZF axioms

We state the axioms in plain language with their formal content indicated. We encourage the reader to match each to the chapter that used it silently.

Extensionality. Sets with the same elements are equal. (Chapter 1's Definition 1.2, promoted to axiom. It makes sets *data*, identified structurally, as opposed to objects with hidden identity. Every value-semantics data model copies this design decision.)

Pairing. For any a, b, the set $\{a, b\}$ exists.

Union. For any set F, the set $\bigcup F = \{x : \exists Y \in F,\ x \in Y\}$ exists. (With Pairing, binary union follows.)

Power Set. For any A, $\mathcal{P}(A)$ exists. (Chapter 2. Jointly with Union and Pairing it builds products, hence relations and functions. Chapters 3–4 rest on these three.)

Separation (schema). For any set A and any formula $\varphi(x)$ (parameters allowed), $\{x \in A : \varphi(x)\}$ exists. (The licensed form of comprehension: Chapter 1's repair of Russell. It is a *schema*, one

axiom per formula, infinitely many in total, because first-order logic cannot quantify over formulas.)

Replacement (schema). For any set A and any formula defining a class function $x \mapsto y$, the image $\{y : x \in A\}$ exists. (Images of sets under definable functions are sets. Replacement is needed where Separation is too weak: constructing $\omega + \omega$, justifying transfinite recursion at the level of generality Chapter 7 used it. Most working mathematics outside set theory never touches it, a calibration worth possessing.)

Infinity. An inductive set exists. (Chapter 7's Definition 7.2 needs it. Without it, ZF proves only the existence of finite sets; indeed, the hereditarily finite sets form a model of all the other axioms. Every claim about $\mathbb{N}$, streams, or asymptotics stands on this axiom.)

Foundation (Regularity). Every nonempty set A has an element disjoint from A; equivalently, $\in$ is well-founded, and no set is an element of itself, nor is there any infinite descending $\in$-chain. (The tidiness axiom: it forbids $x \in x$ and membership cycles.)

Foundation is equivalent, over the other axioms, to the statement that every set appears in the **cumulative hierarchy**. We define by transfinite recursion (Theorem 7.5) $V_0 = \emptyset, V_{\alpha+1} = \mathcal{P}(V_\alpha), V_\lambda = \bigcup_{\alpha<\lambda} V_\alpha$ at limits. Then every set is a subset of some

V_α, and the least such stage index is the set's **rank**: $\text{rank}(x) = \min\{\alpha : x \subseteq V_\alpha\}$, or equivalently, by transfinite recursion, $\text{rank}(x) = \sup\{\text{rank}(y) + 1 : y \in x\}$. A set of rank α is a subset of V_α and first becomes a *member* one stage later, at $V_{\alpha+1}$. For example, $\emptyset \subseteq V_0$ and $\emptyset \in V_1$, so $\text{rank}(\emptyset) = 0$; each von Neumann natural n has rank n (Jech 2003, Chapter 6, on the cumulative hierarchy and rank). The picture (all sets built bottom-up in well-founded stages from nothing) is the ontological analogue of an immutable, acyclic data world. It is a *choice*, not a necessity. Dropping Foundation in favor of an Anti-Foundation Axiom (Aczel) yields an equally consistent theory whose non-well-founded sets model streams, cyclic object graphs, and bisimulation-based process equivalence directly. Which theory to use is a modeling decision. Mainstream mathematics takes Foundation, and so does this book. However, the reader who has compared value semantics with reference-and-cycle semantics in software has already faced the analogous trade.

Note (what "is" a set). Within ZF, everything is a set built in this hierarchy. Chapter 1's applied atoms (urelements) are accommodated either by the variant theory ZFA or, in practice, by silently encoding atoms as sets and never inspecting the encoding: the same move as Kuratowski pairs (Definition 2.8), governed by the same discipline of using only the interface.

8.3 The Axiom of Choice

Axiom of Choice (AC). Every family $(A_i)_{i \in I}$ of nonempty sets admits a choice function (Definition 4.8): some c with $c(i) \in A_i$ for all $i \in I$.

ZF plus AC is **ZFC**, the standard foundation of contemporary mathematics. AC differs in character from the ZF axioms. They *construct* sets from given sets by explicit operations. AC asserts the *existence* of a set (the choice function's graph) that no rule need define. When a uniform selection rule exists (least element of each nonempty set of naturals, minimum ID in each shard), choice is a theorem, not an axiom. AC supplies a choice function for an arbitrary family when the function's existence has not already been established by the other axioms together with a uniform selection construction; the issue is provability of existence, not whether some formula happens to describe a choice. Finite families never need it (induction on the family's size, Chapter 4).

Theorem 8.1 (The standard equivalents). Over ZF, the following are equivalent: (i) AC; (ii) **Zorn's Lemma**: a nonempty poset in which every chain has an upper bound contains a maximal element; (iii) the **Well-Ordering Theorem**: every set admits a well-order; (iv) every surjection has a right inverse (a *section*); (v) arbitrary Cartesian products of nonempty sets are nonempty; (vi) any two sets are

$\preccurlyeq$-comparable (the totality left open in Theorem 6.2). (Jech 2003, Chapter 5.)

The equivalents are where working mathematics actually consumes AC. Zorn's Lemma yields maximal ideals, bases for arbitrary vector spaces, ultrafilters. The Well-Ordering Theorem yields the cardinal arithmetic of Chapter 6 in full generality (every set bijects with an initial ordinal, so cardinalities are well-ordered and $\kappa \cdot \kappa = \kappa$ holds for all infinite κ). Clause (iv) is quietly invoked whenever "pick a representative of each equivalence class" or "for each output, fix one input producing it" occurs over arbitrary sets. On the countable scale, the *countable* axiom of choice and its cousin *dependent choice* (DC: for a nonempty set A and a relation R on A with $\forall a \in A\ \exists b \in A\ (a\ \mathrm{R}\ b)$, there is a sequence $(a_n)_{n\in\mathbb{N}}$ with $a_n\ \mathrm{R}\ a_{n+1}$ for every n, and it may be started at any chosen a_0) suffice for classical analysis. We used them, flagged, in Theorem 6.3(iii) and the chain characterization of well-foundedness (Definition 7.3).

The cost. AC's non-constructive existences include some famously pathological ones: a well-ordering of $\mathbb{R}$, which AC guarantees without supplying any rule for it; non-measurable subsets of $\mathbb{R}$ (Vitali; Chapter 14 meets them as the reason a probability with the natural properties cannot be defined on every subset of an interval); and the Banach–Tarski decomposition of a solid ball into five pieces reassembling, by rotations and translations alone,

into two balls of the original size. These are not contradictions but demonstrations of how far bare existence can float from any exhibited instance, the mathematical extreme of a specification satisfiable only by objects no algorithm produces. One calibration is needed. Non-constructive existence, absence of a defining formula, and absence of an algorithm are three different things. AC supplies no rule, but that does not show the object is undefinable: whether some formula defines a well-ordering of $\mathbb{R}$, or a non-measurable set, depends on further axioms ("definable" must itself be pinned down, since with arbitrary set parameters allowed every set V is defined by the formula $x \in V$; the convention here is definability by a formula *without parameters*. Under Gödel's axiom $V = L$, whose universe L appears in the next paragraph, there is a parameter-free definable well-ordering of $\mathbb{R}$, and hence a parameter-free definable non-measurable set; Jech 2003, Chapters 13 and 25). Constructive mathematics and type theory (the foundations under several proof assistants) accordingly restrict or refuse AC in its full form. In those settings a "choice" is data, literally a function term. The applied programmer who has demanded "don't just tell me a valid configuration exists; produce one" has taken the constructivist's side of the argument in the practical register.

The status. Gödel (1938): if ZF is consistent, so is ZFC (AC cannot be refuted); the constructible universe L models it. Cohen (1963): if ZF is

consistent, so is ZF + ¬AC (AC cannot be proved); forcing produces the models. AC is thus independent of ZF, exactly as CH is independent of ZFC (same two authors, same two methods; Gödel's inner models and Cohen's forcing are the twin engines of all such results; Jech 2003, Chapters 13–14; Kunen 1980, Chapters VI–VII). Mainstream mathematics adopts AC and uses it freely. This book has flagged each use, not from squeamishness, but because knowing *which* results depend on *which* commitments is what "foundation" means operationally.

8.4 What axiomatization buys, and what it cannot

Consistency is conditional. By Gödel's second incompleteness theorem, ZFC (if consistent) cannot prove its own consistency. All consistency claims above are relative ("if ZF is consistent, then…"). A century of unproblematic use, and a coherent intended picture (the cumulative hierarchy), ground reasonable confidence. However, it is empirical-style confidence, not proof, and stating it as such is the correct epistemic posture.

Completeness is unavailable. Gödel's first incompleteness theorem: any consistent, effectively axiomatized theory extending basic arithmetic leaves some sentences undecided. CH is the celebrity example for ZFC, but the phenomenon is pervasive. Stronger axioms (large cardinals) settle some questions and leave others, without end. An axiom

system is an interface, and every interface underdetermines its implementations.

Multiple foundations coexist. ZFC is the incumbent, not the only tenant. Type theories (Martin-Löf, the Calculus of Constructions under Coq/Rocq and Lean) found mathematics on typed functions rather than untyped membership. Categorical foundations (ETCS, topos theory) found it on composition of maps rather than element-of. Each proves essentially the same body of ordinary mathematics through different primitives. For the practitioner, the parallel with data modeling is exact and worth taking seriously: relational, document, and graph databases also encode the same information under different primitives with different ergonomics. Foundational monism is as unnecessary in mathematics as storage monism in engineering. What matters, in both, is knowing the contract in force and translating faithfully at the boundaries.

For applied work, the axioms are a safety certificate, not a toolkit. No applied chapter of this book manipulates axioms directly; Part III consumes the *theorems*. The certificate says: the constructions of Parts I–II (products, quotients, closures, fixpoints, cardinal bounds) are formalizable within ZFC and introduce no foundational assumptions beyond those identified in this chapter, because they all factor through the licensed operations. Their consistency is therefore conditional on the consistency of the underlying theory, exactly as the

first paragraph of this section says, and no stronger certificate is on offer. That is the precise sense in which a database schema, a specification language, or a type system "rests on set theory": the mathematical semantics assigned to it lives in the cumulative hierarchy, and the axioms warrant the semantics.

8.5 Summary

ZF regularizes Part I's practice. Extensionality identifies sets by content; pairing, union, power set, separation, and replacement construct; infinity admits $\mathbb{N}$; foundation stratifies the universe into ranked stages built from nothing. AC adds rule-free selection, equivalent to Zorn, well-ordering, sections of surjections, and cardinal comparability. It is independent of ZF, indispensable to infinite-dimensional mathematics, and priced in non-constructive pathology, with countable and dependent choice the mild doses classical analysis actually requires. Incompleteness makes every effective foundation partial, consistency claims conditional, and rival foundations legitimate. The axioms function as mathematics' interface specification, consumed in Part III only through its theorems.

Part III: Applications

Chapter 9. Sets and Logic: Boolean Algebras

Dependencies: Chapters 1–2 and 5 (lattices). In this chapter, we make exact the correspondence, used informally since Chapter 2, between set operations and logical connectives. We axiomatize Boolean algebra, state the finite representation theorem, and follow the structure into its applied habitats: digital circuits, query conditions, feature flags, and the satisfiability problem.

9.1 Propositions and sets: the fundamental dictionary

We fix a universe $\mathcal{U}$ of situations: possible worlds, database rows, system states, test inputs. Every *property* meaningful for those situations determines its **extension**, the subset of $\mathcal{U}$ where it holds. Under this reading, the logical connectives *are* the set operations:

Logic	Sets
$\varphi \wedge \psi$	$E_\varphi \cap E_\psi$
$\varphi \vee \psi$	$E_\varphi \cup E_\psi$
$\neg\varphi$	E_φ^c
$\varphi \Rightarrow \psi$ valid	$E_\varphi \subseteq E_\psi$
$\varphi \Leftrightarrow \psi$ valid	$E_\varphi = E_\psi$
contradiction	$\emptyset$
tautology	$\mathcal{U}$

This dictionary is the engine behind Theorem 2.1 (every propositional tautology is a set identity) and behind a daily applied act: a `WHERE` clause, a firewall rule, an alerting condition, and a cohort definition are all formulas *evaluated as their extensions*. Implication deserves emphasis, being the entry practitioners most often garble. "Every premium user is active" is an inclusion $P \subseteq A$, refuted by exhibiting one element of $P \cap A^c$, and *unaffected* by how many non-premium users are active or inactive. The set reading dissolves at a glance the classic confusions (converse for implication, correlation-shaped readings of $\subseteq$) that the symbolic reading invites.

9.2 Boolean algebras

The structure shared by $\mathcal{P}(\mathcal{U})$ with union, intersection, and the complementation map $A \mapsto A^c$,

and by propositional logic with its connectives, merits its own axiomatization.

Definition 9.1 (Boolean algebra). A *Boolean algebra* is a set B with binary operations $\vee,\wedge$, a unary operation $\neg$, and distinguished elements $0 \neq 1$, satisfying: commutativity and associativity of $\vee,\wedge$; distributivity of each over the other; identity laws $x \vee 0 = x$, $x \wedge 1 = x$; and complement laws $x \vee \neg x = 1$, $x \wedge \neg x = 0$.

All other familiar laws (idempotence, absorption, De Morgan, double negation, uniqueness of complements) are *derivable*, each in a few lines from the stated equations (Givant and Halmos 2009, Chapter 2). This is a small but real demonstration of the axiomatic method's economy: a short list of equations (ten as stated above, two for each of commutativity, associativity, distributivity, identity, and complement, and known to be reducible further) governs the entire propositional calculus. Every Boolean algebra is a lattice (Definition 5.8) under $x \leq y \Leftrightarrow x \wedge y = x$, with $\vee,\wedge$ as join and meet. In $\mathcal{P}(\mathcal{U})$ this order is $\subseteq$. Inclusion, implication, and lattice order are one relation in three dresses.

Example 9.1 (The algebras in the field). (a) $\mathcal{P}(\mathcal{U})$ for any nonempty $\mathcal{U}$ (for $\mathcal{U} = \emptyset$ the power set has one element and $0 = 1$, which Definition 9.1 excludes; some authors admit this *trivial* algebra, and we do not); (b) the two-element algebra $\mathbb{B} = \{0,1\}$: the algebra of a single wire, of one bit, of truth

values; (c) the finite-or-cofinite subsets of a countably infinite set such as $\mathbb{N}$ (a Boolean algebra not isomorphic to any power set: it is countably infinite, since there are countably many finite subsets of $\mathbb{N}$ and countably many cofinite ones, while a power set is finite or uncountable by Theorem 6.4; over an uncountable set the same construction is uncountable, and the argument no longer applies); (d) the *regular open* sets of a nonempty topological space X (those equal to the interior of their closure), with $U \wedge V = U \cap V$, $U \vee V = I\big(\overline{U \cup V}\big)$, and $\neg U = I(X \setminus U)$, where $I(A)$ denotes the interior of A in X and $\overline{A}$ its closure; the join is not plain union, since $(0,1) \cup (1,2)$ is not regular open in $\mathbb{R}$ (Givant and Halmos 2009, Chapter 10; this algebra is used in forcing, Chapter 8's engine, a pointer for the curious); (e) divisors of a square-free integer $n > 1$ under lcm, gcd, and $x \mapsto n/x$; (f) the clopen subsets of any nonempty topological space, the example that, via Stone's theorem below, is secretly universal.

Theorem 9.1 (Representation, finite case). Every finite Boolean algebra is isomorphic to $\mathcal{P}(A)$ for some finite set A, namely A = its set of **atoms** (minimal nonzero elements), each element mapping to the set of atoms below it. Consequently every finite Boolean algebra has cardinality 2^n for some $n \geq 1$, and two finite Boolean algebras of equal cardinality are isomorphic (Givant and Halmos 2009, Chapter 15).

The infinite case (Stone 1936; Givant and Halmos 2009, Chapters 22 and 34) represents any Boolean algebra as the clopen sets of a topological space. Power sets no longer suffice (Example 9.1(c) is the witness), and the corrected statement founded a whole duality theory. For applied purposes, the finite theorem already carries the working moral: **finite Boolean reasoning, however exotically presented, is reasoning about subsets of atoms**, and the atoms are discoverable. In a system of n independent binary features, the atoms are the 2^n full configurations (minterms), and every feature predicate is a union of them.

9.3 Normal forms, circuits, and the cost of Boolean reasoning

Definition 9.2 (Boolean function). An *n-ary Boolean function* is any $f : \mathbb{B}^n \to \mathbb{B}$. There are 2^{2^n} of them (the function-space count of Section 4.4, applied twice), a double-exponential growth that dominates the engineering of everything below.

Theorem 9.2 (Functional completeness and normal forms). Every Boolean function is expressible using $\wedge$,$\vee$, $\neg$, indeed in *disjunctive normal form* (DNF: a disjunction of conjunctions of literals, one conjunct per satisfying input row) and dually in CNF. Moreover $\{\neg,\wedge\}$, $\{\neg,\vee\}$, and the single connectives NAND and NOR are each functionally complete (Huth and Ryan 2004, Section 1.5).

The theorem grounds digital electronics: any specified combinational behavior is buildable from one gate type, which is why fab economics could standardize on NAND. However, its fine print prices the grounding. The truth-table DNF has one term per satisfying row, and functions with exponentially many such rows admit no small normal form of that shape (parity is the standard example: its DNF *requires* 2^{n-1} terms). Minimizing two-level forms is the Quine–McCluskey / Karnaugh-map problem, itself computationally hard at scale. Deciding whether a CNF formula is satisfiable *at all* is **SAT**, the original NP-complete problem (the Cook–Levin theorem; Cook 1971; Levin 1973). Therefore, applied Boolean practice splits by problem shape. Hardware synthesis restructures formulas into multi-level circuits and *binary decision diagrams* (Bryant 1986). A *reduced ordered* BDD, built with a fixed variable order and with duplicate and redundant nodes merged, is a canonical DAG representation of a Boolean function, so two functions are equal iff their reduced ordered BDDs over the same order are identical. In an implementation that shares all nodes through one unique table, that identity test is a pointer comparison and takes constant time; two BDDs built separately must be compared structurally. The price is variable-order sensitivity and worst-case exponential size (integer multiplication is the classic example, Bryant 1986, Section 3.3 and the appendix). Verification and planning encode into

SAT and hand the hard instances to conflict-driven solvers (Biere et al. 2021), whose practical performance on industrial instances is one of computing's happier empirical surprises. The reader should hold the facts in calibrated tension, and the calibration is finer than "hard versus easy." Truth-table enumeration costs 2^n steps, established. NP-completeness says that a polynomial-time SAT algorithm would give $P = NP$; it does *not* establish an exponential lower bound, and even assuming $P \neq NP$ leaves subexponential algorithms open. The stronger belief that SAT needs exponential time is a separate, named hypothesis (the exponential time hypothesis; Impagliazzo, Paturi, and Zane 2001). Typical tractability is observed, not guaranteed.

Example 9.2 (Feature flags, atoms, and dead configurations). A product gated by flags `beta`, `eu`, `mobile` lives over the atom set $\mathbb{B}^3$: eight minterms. Each rollout rule is a Boolean function. The *reachable* configurations, however, may be fewer than eight (a dependency "beta requires mobile" kills the atoms with `beta` $\wedge$ `¬mobile`). Two engineering artifacts drop out of the algebra directly: (i) rule equivalence and rule subsumption ("this rollout condition is already implied by that one") are inclusions in the quotient algebra over reachable atoms, decidable by SAT query; (ii) *dead code and dead configuration* detection is emptiness of an extension, again SAT. Teams that reason about flags row-by-row in a spreadsheet are executing Theorem 9.1 manually. Past a dozen flags,

mechanization is the only honest option, and it is cheap.

9.4 Quantifiers, predicates, and the limits of the propositional dictionary

First-order logic extends the dictionary. A predicate $\varphi(x)$ over $\mathcal{U}$ has extension $E_\varphi \subseteq \mathcal{U}$, and the quantifiers become the generalized operations of Definition 2.5,

$$E_{\forall x\, \varphi(x,\cdot)} = \bigcap_{a \in \mathcal{U}} E_{\varphi(a,\cdot)}, \qquad E_{\exists x\, \varphi(x,\cdot)} = \bigcup_{a \in \mathcal{U}} E_{\varphi(a,\cdot)},$$

and relational-algebra projection (Chapter 10) implements ∃ while division implements a bounded ∀. Two calibration points follow, both with applied bite. First of all, over *infinite* domains the quantifier operations outrun Boolean algebra. First-order validity is undecidable (Church–Turing; Huth and Ryan 2004, Section 2.5), so the "just SAT-solve it" strategy has a boundary, and specification tools either restrict to decidable fragments (Presburger arithmetic, monadic classes, SMT theories) or accept incompleteness and interactivity. However, over *finite* domains everything re-propositionalizes (∀ is a big conjunction), which is precisely the move model checkers and Alloy (Chapter 12) make: bound the universe, ground the quantifiers, ship to SAT. The applied art is choosing bounds that make grounding feasible while keeping the bugs findable, a cardinality trade made with Chapter 6's eyes open.

9.5 Summary

Extensions translate logic into set algebra: connectives to operations, implication to inclusion, validity to equality with $\mathcal{U}$. The stated equations axiomatize the shared structure. Finite Boolean algebras are exactly finite power sets, with atoms as the hidden coordinates and minterms their formula dress. Normal forms exist universally but not economically. Circuit synthesis, reduced ordered BDDs, and SAT solving are the three industrial responses to the double-exponential landscape, with NP-completeness the established boundary (an exponential lower bound being a further hypothesis) and solver performance the empirical grace. Quantifiers extend the dictionary through infinitary unions and intersections, purchasing undecidability in general and re-propositionalization over finite bounds, the operating principle of bounded model checking.

Chapter 10. The Relational Model: Sets in Databases

Dependencies: Chapters 3–5 (relations, functions, equivalences) and 9 (Boolean conditions). Codd's relational model is applied set theory's flagship: a data architecture derived, nearly axiomatically, from the concept of a finite relation. In this chapter, we develop the model, the relational algebra, keys and dependencies as set-theoretic constraints, and the exact points (nulls, duplicates, ordering) where industrial SQL departs from the mathematics, with the practical consequences of each departure.

10.1 Relations as tables, precisely

Definition 10.1 (Attribute, schema, tuple, relation instance). Fix a set $\mathcal{A}$ of *attribute names*, each attribute A carrying a *domain* $\mathrm{dom}(A)$ of atomic values. A *relation schema* is a finite set $H \subseteq \mathcal{A}$ of attributes. A *tuple over* H is a function t with domain H satisfying $t(A) \in \mathrm{dom}(A)$ for each $A \in H$. A **relation instance** over H is a finite *set* of tuples over H. A *database schema* is a finite collection of named relation schemas; a *database instance* assigns each a relation instance.

Three deliberate choices in Definition 10.1 constitute the model's engineering content. First of all, tuples are functions *from attribute names*, not

positional sequences. Therefore, columns have no inherent order, and the model is stable under schema display order (contrast a CSV file, whose tuples are position-indexed). Then, instances are *sets*. Rows have no order and no multiplicity, and "the same row twice" is not a representable state. Finally, domains are *atomic*: no set-valued or nested fields in the classical model. "First normal form" is this clause read as a discipline. Nested relational and document models relax exactly it, at the price of reintroducing the navigation the flat model was invented to remove. Each choice discards representational detail to buy *data independence*: queries address content (what values are present), never layout (where and in what sequence they are stored). This is the same interface-versus-implementation discipline as Definition 2.8's pairs, elevated to an architecture.

Example 10.1. Schema Emp = {eid,name,dept}, instance $\{ t_1, t_2 \}$ with t_1 = {eid ↦ 7,name ↦ Noor,dept ↦ QA}, t_2 = {eid ↦ 9,name ↦ Wei,dept ↦ QA}. The instance equals itself with rows listed in any order: there is no "first employee." Any question whose answer depends on physical row order is, in this model, not a question about the data.

10.2 The relational algebra

Queries are built from six operators. Each is an operation on relation instances definable in Part I's vocabulary.

Definition 10.2 (Core operators). Let R, S be instances over schemas H_R, H_S.

1. **Selection** $\sigma_\varphi(R) = \{t \in R : \varphi(t)\}$, for a Boolean condition φ over H_R's attributes. This is Chapter 1's separation, with Chapter 9 supplying the condition algebra.
2. **Projection** $\pi_X(R) = \{\, t \restriction X : t \in R \,\}$ for $X \subseteq H_R$, where $t \restriction X$ is function restriction. Projection is the image of R under a restriction map and, being an image, potentially collapsing: distinct tuples agreeing on X become one. Projection is the algebra's existential quantifier (Section 9.4) and its one *inherently* duplicate-removing operator.
3. **Natural join** $R \bowtie S$: the set of tuples t over $H_R \cup H_S$ with $t \restriction H_R \in R$ and $t \restriction H_S \in S$, that is, the tuples whose restrictions land in both instances. With $H_R \cap H_S = \emptyset$ it degenerates to the Cartesian product (up to the function/pair encoding). With $H_R = H_S$ it degenerates to intersection. In general it is the model's engine of *recombination*, and Chapter 3's relational composition is

$\pi_{(H_R \cup H_S) \setminus B}(R \bowtie S)$ for the shared middle attributes B.

4. **Union, difference** $R \cup S$, $R \setminus S$, requiring $H_R = H_S$ (*union compatibility*). Intersection is derived: $R \cap S = R \setminus (R \setminus S)$.
5. **Renaming** $\rho_{A \to B}(R)$, adjusting attribute names. Renaming is bureaucratic but necessary, for example, to join a relation with itself on different roles (manager/report).

Theorem 10.1 (Algebraic identities drive optimization). Among the identities provable directly from Definition 10.2: joins are commutative and associative (up to the trivial schema identifications); selections commute with each other and distribute over union and difference; $\sigma_\varphi(R \bowtie S) = \sigma_\varphi(R) \bowtie S$ whenever φ mentions only H_R's attributes ("predicate pushdown"); $\pi_X \pi_Y = \pi_X$ for $X \subseteq Y$; and cascaded selections merge: $\sigma_\varphi \sigma_\psi = \sigma_{\varphi \wedge \psi}$.

A query optimizer is a rewriting system over exactly these identities plus cost estimates. The plans it enumerates are equal *as sets of tuples* (Theorem 10.1's guarantee), differing only in evaluation cost. The guarantee is what makes optimization *safe*, and its precondition, set semantics, is what SQL will shortly disturb.

Example 10.2 (A query, algebraically, on an instance). We extend Example 10.1 with a third employee $t_3 = \{\text{eid} \mapsto 4, \text{name} \mapsto \text{Ada}, \text{dept} \mapsto \text{Ops}\}$ and two more schemas, Asg = {eid,pid} and Proj = {pid,status}. Writing tuples in the attribute order just given, for brevity only, the instances are

$$\text{Asg} = \{(7, p_1), (7, p_2), (9, p_2), (4, p_1), (9, p_3)\},$$

$$\text{Proj} = \{(p_1, \text{active}), (p_2, \text{active}), (p_3, \text{closed})\}.$$

"Names of QA employees assigned to an active project" is, with $E = \sigma_{\text{dept=QA}}(\text{Emp})$ and $P = \sigma_{\text{status=active}}(\text{Proj})$,

$$\pi_{\text{name}}(E \bowtie \text{Asg} \bowtie P).$$

We evaluate it in the displayed order. $E = \{t_1, t_2\}$ and $P = \{(p_1, \text{active}), (p_2, \text{active})\}$. Joining E with Asg on eid keeps every assignment of employees 7 and 9, that is, four tuples: $(7, \text{Noor}, \text{QA}, p_1)$, $(7, \text{Noor}, \text{QA}, p_2)$, $(9, \text{Wei}, \text{QA}, p_2)$, and $(9, \text{Wei}, \text{QA}, p_3)$; the tuple $(4, p_1)$ is dropped here because Ada is not in QA. Joining with P on pid keeps three of the four, each gaining status active, and drops $(9, \text{Wei}, \text{QA}, p_3)$ because p_3 is not active. The final projection collapses Noor's two rows into one. The result is {Noor,Wei}. The projection at the end is an existential quantifier: *there exists* a qualifying assignment.

Ask instead for QA employees assigned to *every* active project, and the quantifier flips to ∀. The algebra expresses it by **division**, which needs three

inputs, not two. Let R be over H_R, S over H_S with $H_S \subseteq H_R$, and let C be a *candidate* relation over $H_R \setminus H_S$. Then

$$R \div_C S = \{c \in C : \{c\} \bowtie S \subseteq R\},$$

the set of candidates related in R to every tuple of S; equivalently, the largest $T \subseteq C$ with $T \bowtie S \subseteq R$. The candidate relation cannot be omitted. Without it, an empty divisor S makes every relation over $H_R \setminus H_S$ a solution, and over an infinite attribute domain there is no largest finite one (Definition 10.1 admits only finite instances). The standard *active-domain* division takes $C = \pi_{H_R \setminus H_S}(R)$. Here, $R = \pi_{\text{eid,pid}}(E \bowtie \text{Asg})$ is the set

$$\{(7, p_1), (7, p_2), (9, p_2), (9, p_3)\},$$

with $S = \pi_{\text{pid}}(P) = \{p_1, p_2\}$ and $C = \pi_{\text{eid}}(R) = \{7,9\}$; candidate 7 has both $(7, p_1)$ and $(7, p_2)$ in R, while candidate 9 lacks $(9, p_1)$ (its pair $(9, p_3)$ is irrelevant, since $p_3 \notin S$), so $R \div_C S = \{7\}$, and joining back to E names Noor. The empty-divisor case now has a definite answer that depends on C, exactly as the English question does: with no active projects, active-domain division returns every employee holding *some* assignment, {7,9}, whereas the reading "every QA employee, assignments or not" requires $C = \pi_{\text{eid}}(E)$ and returns all of them. The double-negation encoding $\neg\exists\neg$ computes the same set by two differences, $C \setminus \pi_{H_R \setminus H_S}\big((C \bowtie S) \setminus R\big)$: the candidates for which no required pair is missing.

That is the shape a "for all" in SQL (`NOT EXISTS ... NOT EXISTS`) takes, a small applied triumph of Chapter 9's quantifier dictionary, and the SQL form makes the candidate table explicit whether the author noticed or not.

10.3 Keys and functional dependencies

Constraints are where the model stops describing data and starts legislating it, and every constraint class is a set-theoretic property from Part I.

Definition 10.3 (Functional dependency, key). Let $X, Y \subseteq H$. An instance R satisfies the *functional dependency* (FD) $X \to Y$ iff any two tuples agreeing on X agree on Y: formally, $t \restriction X = t' \restriction X \Rightarrow t \restriction Y = t' \restriction Y$. A **superkey** is an X with $X \to H$; a **candidate key** is a minimal superkey; a *primary key* is a designated candidate key.

An FD $X \to Y$ says exactly that the projection-pairs map $t \restriction X \mapsto t \restriction Y$ is a well-defined *function* (Definition 4.1's single-valuedness; Chapter 5's well-definedness on the quotient by agreement-on-X). A key says that the tuple-identity map from X-values is injective. FDs entail one another: $X \to Y$ and $Y \to Z$ give $X \to Z$, and Armstrong's axioms (reflexivity: $Y \subseteq X$ gives $X \to Y$; augmentation; transitivity) generate exactly the entailed set (Abiteboul, Hull, and Vianu 1995, Chapter 8). Therefore, a schema's declared FDs have a computable closure, and key-finding is

closure computation (iterate the axioms to a fixpoint, Section 7.5's pattern in miniature).

We can now compress **normalization** to its set-theoretic core. Two levels must be distinguished first: the FDs *declared for the schema* (together with everything Armstrong's axioms derive from them), which every legal instance must satisfy, and the FDs that merely happen to hold in one small instance. Normalization concerns the former. An FD is *trivial* iff $Y \subseteq X$; trivial FDs hold in every instance and say nothing. A nontrivial schema-level FD $X \to Y$ with X not a superkey means that the schema *permits* the function $X \mapsto Y$ to be stored redundantly, once per tuple sharing an X-value. Update anomalies are the resulting consistency obligations. The decomposition cure factors the relation: project onto XY and onto $(H \setminus Y) \cup X$. The correctness conditions are again Part I's. The decomposition must be *lossless* ($R = \pi_{X_1}(R) \bowtie \pi_{X_2}(R)$, a join-reconstruction equation with a clean FD-based criterion) and ideally *dependency-preserving*. Boyce–Codd normal form is the terminus: for every nontrivial FD $X \to Y$ implied by the schema's dependencies, X is a superkey (Abiteboul, Hull, and Vianu 1995, Chapter 11). What BCNF removes is the *possibility* of the redundancy the dependencies describe. It does not assert that an instance in a violating schema actually contains repetitions, and it does not exclude repetitions that no declared dependency governs. The reader who has factored a function through its kernel (Theorem 5.2) has performed normalization's

essential act. A schema in BCNF is a database that has performed it wherever its declared dependencies called for it.

10.4 Where SQL departs from set theory, and what it costs

SQL implements the model with three deliberate deviations. Each was adopted for defensible engineering reasons. Each has a precise mathematical description and a characteristic bug family. An engineer fluent in both layers converts mysterious behavior into predicted behavior.

Bags, not sets. SQL tables and intermediate results are *multisets* (Chapter 16): duplicates are permitted and significant. `SELECT dept FROM Emp` retains one row per employee, not per department. The projection that mathematically removes duplicates operationally keeps them unless `DISTINCT` is spoken. The identities of Theorem 10.1 must be re-audited under bag semantics, and some fail. π no longer commutes with $\cup$ the way set union suggests. `UNION` versus `UNION ALL` split the concept in two. Cardinality-sensitive aggregates (`COUNT`, `SUM`, `AVG`) make the difference between bag and set projections a difference in *totals*: the classic double-counting join bug, in which a one-to-many join silently multiplies a measure before aggregation. The repair discipline is always the same. Know which multiplicity the query's semantics require, and place

DISTINCT or pre-aggregation exactly at the collapse point. The algebra says where.

Nulls and three-valued logic. SQL adjoins to every domain a marker NULL ("value absent") and evaluates conditions in a three-valued logic: comparisons with NULL yield UNKNOWN, and WHERE keeps only TRUE rows. The Boolean algebra of Chapter 9 is thereby replaced by a structure in which the law of excluded middle fails: $\sigma_{\varphi}(R) \cup \sigma_{\neg\varphi}(R) \neq R$ when φ can evaluate to UNKNOWN, because rows with null attributes fall out of *both* branches. The notorious concrete case: `x NOT IN (subquery)` returns no rows at all if the subquery yields a single NULL, since the membership test can never reach TRUE (the PostgreSQL documentation states the rule explicitly in its section on subquery expressions; PostgreSQL Global Development Group 2026). Mathematically, NULL is a totalization sentinel (Chapter 4's $B \sqcup \{\bot\}$) pushed *through* the condition algebra rather than handled at the boundary. The discipline it demands is precisely the boundary-handling that the sentinel strategy always demands, in any language: decide the null semantics per attribute, use `IS [NOT] NULL` and COALESCE at the edges, and prefer NOT EXISTS (which quantifies correctly) to NOT IN.

Order as a costume. Relation instances are unordered; ORDER BY is a presentation directive appended to a query, not a property of data. The predictable bug family: relying on "natural" order

(insertion order, index order) without `ORDER BY`. Such behavior is unspecified by the model and delivered by the implementation until a plan change withdraws it. The model's advice is unambiguous. Order is either demanded explicitly (making the result a *sequence*, Definition 4.7, a different mathematical object) or not at all.

Note. These paragraphs state facts about SQL as standardized and commonly implemented. Specific products deviate further (for example, in null handling within `GROUP BY`, or duplicate elimination in set operators). The mathematical layer is the stable reference against which each product's behavior can be audited. That is much of its applied value.

10.5 Views, integrity, and the closed world

A **view** is a named query: an intensional relation, in Chapter 3's vocabulary, whose extension is computed on demand. A *materialized* view trades the intension for a stored extension plus a maintenance obligation (Section 3.3's incremental-closure problem in another dress). **Integrity constraints** (keys, FDs, inclusion dependencies $\pi_X(R) \subseteq \pi_Y(S)$; foreign keys are Definition 1.5 doing daily labor) are inclusions and functionalities the DBMS enforces at write time. The theory of when a constraint set is enforceable incrementally is applied set theory of a high order. Finally, query answering presumes the **closed-world assumption**: a tuple

absent from the instance is *false*, not unknown. Negation-as-absence is the semantic root of both the power (difference queries work) and the peril (absence of evidence read as evidence of absence) of analytical conclusions drawn from a database. The open/closed distinction is not pedantry. It decides what a `LEFT JOIN ... WHERE right IS NULL` (anti-join: set difference in SQL costume) *means* about the world, and misreading it is a reporting-layer bug that no query optimizer will ever catch.

10.6 Summary

The relational model is the theory of finite sets of attribute-indexed tuples. Schemas are attribute sets, instances are tuple sets, and queries are compositions of six set-definable operators whose provable identities constitute the optimizer's license. Quantifiers enter through projection and its division dual (which needs an explicit candidate relation), and constraints (keys, FDs, foreign keys) are injectivity, functionality, and inclusion. Normalization is kernel-factoring applied to storage, removing the redundancy that the declared dependencies permit. SQL's three departures (bags, nulls with three-valued logic, incidental order) are locally motivated, globally consequential, and diagnosable against the set-theoretic reference layer, which is the practitioner's standing advantage in wielding it.

Chapter 11. Sets in Programming: Collections, Types, and Interfaces

Dependencies: Chapters 1–5; Chapter 9 for Boolean structure. Programming languages traffic in set-theoretic concepts under assumed names. Collections implement sets and functions, type systems approximate sets of values, and interfaces denote sets of behaviors. In this chapter, we map the correspondences exactly, because the mismatches (mutation, equality, partiality, bounded universes) are where production defects live.

11.1 The set interface and its implementations

Every mainstream standard library ships a type called `Set` promising, at minimum: membership testing, insertion, and iteration, with "no duplicates" as the advertised invariant. Judged against Definition 1.1 and extensionality, the promise decomposes into obligations the *library* can enforce and obligations only the *application* can.

The library enforces: no two stored elements compare equal (under the structure's equality; see below); operations preserve this invariant.

The application must supply: an equality that is a genuine equivalence relation (Definition 5.1) on the

intended universe, consistent with hashing (`a equals b` ⇒ `hash(a) = hash(b)`) or with ordering (a total order, Definition 5.6, for tree-based sets); and *stability* of both under whatever mutation the elements permit while they are members.

Each unenforced obligation is a bug family with a standard name. Equality that is not symmetric or not transitive (a tolerance-based `equals`: ε-closeness is not transitive) makes deduplication order-dependent. Hash inconsistent with equality makes "equal" elements coexist in one hash set. Mutating an element's hash-relevant state while it is a member (Section 1.5) makes the set report the element absent while iterating over it. Comparators inconsistent with equality make tree sets and hash sets *disagree about the same data*: two "sets" with the same inserts and different members. None of these failures is exotic. The contracts are stated in the platform documentation: Java's `Set` interface warns against mutable elements and requires `equals` and `hashCode` to be consistent (Oracle 2025), and Python's data model requires that objects comparing equal have the same hash (Python Software Foundation 2026). All are violations of Part I's stated preconditions rather than library defects. The mathematics is the contract, and the contract is testable. Property-based testing of the equivalence and consistency laws is cheap and catches real errors.

Here are the standard representations, with their contracts. Hash sets: expected $O(1)$ membership, no order guarantee (an unordered set honestly rendered). Tree sets: $O(\log n)$ with iteration in key order (a set *plus* a total order, a richer object, Chapter 5). Bit sets: Chapter 2's characteristic vectors, unbeatable when the universe is small, enumerable, and fixed; the representation *is* the universe declaration. Bloom filters: approximate membership with one-sided error (false positives only). A Bloom filter is not a set but a *superset oracle*: it answers "possibly in S" or "certainly not in S." Correctness arguments must use $S \subseteq \tilde{S}$, and the design question is the acceptable measure of $\tilde{S} \setminus S$ (Chapter 13's counting rules and Chapter 14's independence machinery yield the standard sizing formulas). Sorted arrays of immutable snapshots: sets as *values*, which is where the deeper design axis lies.

Mutability is the real divide. A mutable set is not a set in the mathematical sense at all. It is a *variable whose value is a set*: a function from time to $\mathcal{P}(\mathcal{U})$, in this book's vocabulary (Definition 4.7 with a temporal domain). Most set-theoretic reasoning (extensionality-based equality, use as map keys, safe sharing across threads) applies to the *values*, not the variable. The persistent/immutable collections of functional languages (and of version-controlled storage generally) restore the mathematics by making every update produce a new value. This is Chapter 8's cumulative hierarchy as an engineering

pattern: nothing changes; things are only built anew from what exists. The costs (allocation, sharing structure) and the mitigations (hash array mapped tries, structural sharing) are the standard engineering literature's subject. The conceptual point is that "should this set be a value or a place?" is the first design question, and everything in Sections 1.5 and 5.1 flows from its answer.

11.2 Maps, records, and functions again

A `Map<K, V>` denotes a finite partial function (Example 4.3), and the anatomy of Chapter 4 reads directly onto its API: totality (`getOrDefault`, `Optional`), injectivity (invertible maps; a bidirectional map is a bijection maintained as two inverse functions), image and preimage (`values()`, and filtering keys by value predicate), composition (map lookup chains), kernel (grouping: `groupBy` literally constructs $\ker f$ as a map from images to preimage classes, Theorem 5.1 as a library function). A record/struct type denotes a product (Example 2.4) with named projections. A tagged union denotes a disjoint union (Definition 2.10), with pattern-matching exhaustiveness the surjectivity check on the tag: a compiler-enforced instance of Example 4.2's "adding a variant breaks the exhaustive handler."

One correspondence deserves its own paragraph because entire architectures turn on it: **the**

adjacency isomorphism $\mathcal{P}(A \times B) \cong \left(\mathcal{P}(B)\right)^A$ (Theorem 4.5). A relation stored as a set of pairs (edge list, junction table) and the same relation stored as a map from each a to its image set (adjacency list, "document with embedded array") are the two sides of a natural bijection: logically interchangeable, operationally divergent (Section 3.4). Schema migrations between the two shapes are applications of the isomorphism, and their reversibility is exactly its bijectivity. What is *not* preserved is the cost model, which is why the migrations happen.

11.3 Types as sets: how far the reading goes

The slogan "a type is the set of its values" is the working semantics for most applied purposes. `bool` denotes a two-element set; `int32` denotes $\{z \in \mathbb{Z} : -2^{31} \leq z < 2^{31}\}$ (an integer range, not the real interval of Section 2.5); a product type denotes a product; a sum type a disjoint union; a function type `A -> B` a subset of B^A (those functions the language can express); subtyping denotes inclusion, with the substitution principle (a value of the subtype usable wherever a value of the supertype is expected) being the statement that every program context accepting values from the larger set accepts values from the smaller one (Pierce 2002, Chapter 15). What does *not* follow is that every *type constructor* is monotone along $\subseteq$ in each argument. Function types are contravariant in the input and covariant in the

output: a handler accepting any animal serves where a cat handler is required, but a cat-only handler cannot stand in where arbitrary animals may arrive. Mutable read–write containers are not safely covariant without run-time checks. Variance is a property to be declared per constructor, not assumed. Static type checking, on this reading, is a *decidable, conservative approximation of membership*: the checker establishes $v \in [\![T]\!]$ for a machine-checkable sublanguage of claims, rejecting some true ones (the price of decidability, Chapter 9's boundary met again) and admitting escape hatches (casts) whose safety burden returns to the programmer.

The applied art the reading enables is **carving the type to the invariant**: Example 2.4's "distinguished subset of a product" made compile-time. A `NonEmptyList` type for the parser's output, a `ValidatedEmail` distinct from `String`, a state-machine encoding where each state's type admits only its legal transitions. Every such refinement moves a run-time check (separation performed on every use) to a construction-time check (separation performed once, at the boundary). This is the same economy normalization buys databases, and the precise content of "make illegal states unrepresentable."

Note (where the slogan bends). Three calibrated caveats are in order. (1) *Recursive types and functions*: naive set readings of `T = T -> bool`

collide with Cantor (Theorem 6.4: no set surjects onto its power set). The resolution (domain theory: types as structured partial orders, functions as continuous maps; or types as syntactic entities with behavioral meaning) is a large subject that we flag here rather than develop. (2) *Effects and partiality*: an `A -> B` in a language with exceptions, divergence, and IO denotes, at best, a partial function into an enlarged codomain: Chapter 4's totalization moves, now with the moves' costs (checked exceptions, monadic types) part of language design. The enlargement repairs *absent values*, not *absent termination*. A function returning `Option<B>` that diverges on some input is still partial, whatever its signature says, and only a termination argument (Section 7.3) makes it total. (3) *Type theories as rival foundations* (Section 8.4): in them, types are primitive rather than carved from sets, and "is a type a set?" becomes a translation question between foundations rather than a fact. For this book's purposes: the slogan is the right first approximation, the caveats are the honest fine print, and the fine print rarely invalidates an applied design argument conducted at slogan level.

11.4 Interfaces, specifications, and behavioral subsets

We now lift the type-as-set reading one level. An *interface* (trait, protocol, abstract base) denotes the set of implementations satisfying its contract: a

subset of a function-space product, cut out by the contract's conditions (separation once more). "Program to the interface" is then quantification over that set. Substitutability of implementations is membership. A *conformance test suite* is a finite approximation of the membership predicate. Whether it is complete depends on the contract, not on finiteness alone: a stateless Boolean function of two inputs is tested exhaustively by four cases, and Section 11.5 exploits such closed universes deliberately. For unbounded input domains, or for history-dependent behavior, where the contract quantifies over infinitely many inputs or traces, an ordinary finite suite leaves behavior unexamined, and the gap between "passes the suite" and "belongs to the set" is precisely where interface-compatible-but-wrong implementations live. Design-by-contract systems (preconditions, postconditions, invariants) tighten the carved set. Chapter 12 gives the discipline its own notation and calculus, including the refinement order under which "every behavior of the implementation is permitted by the specification" is inclusion again, now of behavior sets, and the master invariant of the entire correctness enterprise.

11.5 Universes, closed worlds, and enumeration in the small

Many of programming's universes are *finite and enumerable by design*: 64-bit words, Unicode scalar

values, the variants of an enum, the states of a fixed protocol. Many others are not. Strings of arbitrary length, lists, and unbounded buffers have countably infinite semantic domains (Theorem 6.3(v)), and a model checker that explores such a domain does so only under an explicit bound or a justified finite abstraction. Where finitude is present, it is exploitable in ways worth naming. Exhaustive case analysis becomes a compiler service (pattern matching over a closed sum). Universal quantification becomes a loop, or a model check: for a protocol whose state space is the product of small per-component universes, "no reachable state violates the invariant" is a finite reachability computation (fixpoint iteration, Section 7.5) run to completion, which is how model checkers turn Part II's transfinite mathematics into a button. Complementation becomes legitimate (Definition 2.4's warning dissolves when $\mathcal{U}$ is an explicit enum). Cardinality becomes an interface consideration: an API returning "all matching records" is sound over a universe of bounded size and a liability over an unbounded one. Pagination is the API designer's acknowledgment that the client's memory holds a finite set, and streaming is the acknowledgment that even enumeration must be lazy. Streaming is stronger than countability (Section 6.2): the result set must not only be countable but effectively enumerable by the server, one item at a time, within the client's patience. The chapter's summary

discipline: *declare the universe; know whether it is bounded and how large; design to both.*

11.6 Summary

Library sets enforce no-duplicates relative to an application-supplied equivalence whose lawfulness (equivalence axioms, hash/order consistency, stability under mutation) the library cannot check and the application must guarantee. Each violated law is a named bug family. Mutable collections are time-indexed set values, and immutability restores extensional reasoning. Maps are finite partial functions with Chapter 4's anatomy as API. Records and unions are products and coproducts, and the adjacency isomorphism underwrites a standard schema migration. Types denote value sets to excellent first approximation (subtyping as inclusion, checking as conservative membership, refinement types as compile-time separation), with calibrated fine print at recursion, effects, and foundations. Interfaces denote behavior sets, test suites approximate membership (completely only over closed, small contracts), and refinement is inclusion. Where programming's universes are finite by design, enumeration, complementation, and exhaustive verification become engineering options; where they are unbounded, bounds or abstractions must be supplied first.

Chapter 12. Formal Specification: Z, B, and Alloy

Dependencies: Chapters 1–5 and 9–11. Formal specification languages are set theory shipped as engineering notation. A system's state is a set-and-function structure, its operations are relations on states, and its correctness conditions are inclusions. In this chapter, we work one specification through in Z-style mathematical notation, state the proof obligations, develop refinement as the master relation between specification and implementation, and survey Alloy's bounded-checking alternative: the trade between proof and search that Chapter 9 priced.

12.1 Why specify in set theory

A specification answers *what*, deferring *how*. The claim this chapter substantiates is that Part I's vocabulary (sets, partial functions, relations, inclusions) is not merely adequate for the answer. With light syntactic sugar, it is the notation the major specification traditions (Z, B, VDM, TLA$^+$ in its state-predicate aspect) actually use. Two properties make it right for the job. First of all, it is *abstract*: a partial function `accounts : AccountId ⇸ Balance` commits to lookup semantics (functionality, definedness on a domain) while committing to nothing about hash maps, B-trees, or replication (Definition 4.1 without Section 3.4). Moreover, it is

analyzable: the conjecture "no withdrawal drives a balance negative" becomes a set-theoretic implication, checkable by proof (B's discharge obligations, Isabelle/HOL embeddings) or by exhaustive bounded search (Alloy). English carries neither property, and code carries the second only after the first is gone.

12.2 A worked specification: an access-badge system

State. We fix basic sets (given types): $PERSON$, $DOOR$, $ZONE$. Doors in this building are *one-way*: turnstiles, exit-only fire doors, and badge readers mounted on one side only. The state comprises

- $badged : PERSON \rightharpoonup ZONE$, a *partial function*: each badged person is in exactly one zone; unbadged persons are outside the system;
- $doors : DOOR \to ZONE \times ZONE$, a total function: $doors(d) = \big(from(d), to(d)\big)$ says that d leads from the first zone into the second (doors are physical, and the pair is ordered);
- $may\_open \subseteq PERSON \times DOOR$, a relation: authorization, with no functionality assumed either way.

Two further items are fixed data rather than state: a designated $occupiable \subseteq ZONE$ (the zones a person

may be located in; a service duct is not one), and a relation $cleared \subseteq PERSON \times DOOR$ derived from role data. We also fix one *axiom* about the building, $from(d) \in occupiable$ and $to(d) \in occupiable$ for every door d, and we return below to why it is needed.

Invariant Inv: the conjunction of

1. ran$badged \subseteq occupiable$ (no one is located in a service duct);
2. nobody is stranded: for every $p \in$ dom$badged$ there is a door d with $(p, d) \in may\_open$ and $from(d) = badged(p)$, that is, a door leading *out* of p's zone that p may open;
3. $may\_open \subseteq cleared$: authorization never exceeds clearance ($\subseteq$ as policy, Example 1.2).

Initial states. $Init$: $badged = \emptyset$ (nobody is inside) and $may\_open \subseteq cleared$. This is a *set* of states, since may_open is left open within the stated bound, and two things must be checked about it. It is nonempty (take $may\_open = \emptyset$), and every state in it satisfies Inv: clause 1 holds because ran$\emptyset = \emptyset$, clause 2 holds vacuously because dom$badged$ is empty, and clause 3 is part of the definition of $Init$. In set terms, $Init \neq \emptyset$ and $Init \subseteq Inv$, reading Inv as the set of states satisfying it.

An operation. $Pass(p, d)$: person p transits door d.

- *Precondition*: $p \in \mathrm{dom} badged$, $(p, d) \in may\_open$, and $from(d) = badged(p)$;
- *Effect*: $badged' = badged \oplus \{p \mapsto to(d)\}$, everything else unchanged,

where $\oplus$ is *functional override* ($f \oplus g$ agrees with g on dom$g$ and with f elsewhere; this operator is the specification literature's single most-used one, definable directly from Part I as $\{(x, y) \in f : x \notin \mathrm{dom} g\} \cup g$). Primed variables denote the post-state. An *operation is a relation between states*, exactly Chapter 3, and a deterministic operation is a partial function on states, exactly Chapter 4.

Now the proof obligations become mechanical in form.

1. **Invariant preservation**: $Inv \wedge pre \wedge effect \Rightarrow Inv'$, for each operation. We discharge it for $Pass$ clause by clause. *Clause 1.* The new range is contained in the old range together with $to(d)$; the old range lies in $occupiable$ by Inv, and $to(d) \in occupiable$ by the building axiom. Without that axiom the clause is *not* provable, and the attempt discovers the missing assumption, which is the routine and considerable value of discharging these things. *Clause 3.* may_open is unchanged. *Clause 2.* For every person other than p, location and authorization are unchanged, so their exit door is still there. For p the

clause now demands a door d' with $(p, d') \in may\_open$ and $from(d') = to(d)$, and *nothing stated implies that one exists*. The following state is a counterexample. Take $PERSON = \{p\}$, $ZONE = \{lobby, lab\}$, $DOOR = \{d_1\}$ with $doors(d_1) = (lobby, lab)$, $occupiable = ZONE$, $may\_open = cleared = \{(p, d_1)\}$, and $badged = \{p \mapsto lobby\}$. All three clauses hold, and $Pass(p, d_1)$ is enabled. After it, $badged' = \{p \mapsto lab\}$, and no door leads out of lab at all. Clause 2 fails, and the person is trapped. The obligation, attempted, has found a real design bug, and the repair is a design decision rather than a proof step: strengthen the precondition of $Pass$ to require an authorized exit from $to(d)$, or add a static requirement on the building and the authorization data (every zone a person may enter has a door they may leave by). Either choice changes the specified system, which is why the obligation could not be discharged without one.

2. **Precondition adequacy**: the specified precondition implies that the effect is well-defined (all function applications within their domains: Chapter 4's partiality discipline made a proof duty). Here $badged(p)$ is applied only under $p \in$ dom$badged$, and $doors$ is total.

3. **Initialization**: as checked above, $Init \neq \emptyset$ and $Init \subseteq Inv$. Both halves matter. A specification whose invariant is unsatisfiable specifies the impossible system, vacuously "correct" and never buildable (Chapter 1's vacuous truth as a lifecycle hazard), and a specification with a declared initial state that violates the invariant can start a run outside the region the preservation argument protects.

Note. The obligations establish that the specification is internally coherent, not that it matches the building, the security policy, or the law. Validation of the model against the world remains empirical. Verification within the model is mathematical. Keeping the two words distinct is specification culture's first discipline. The trap above, for instance, is a failure of the *model* as written, exhibited by a state and a transition of that model; whether the real building has a one-way door into a dead end is a separate, empirical question.

12.3 Refinement: the master inclusion

A specification permits; an implementation chooses. The relation between them is the following.

Definition 12.1 (Refinement, operation form). Let $A \subseteq AS \times AS$ be an abstract operation on abstract states and $C \subseteq CS \times CS$ a concrete operation on concrete states, each a relation between pre-state

and post-state whose precondition is its domain, dom$A$ and dom$C$. Let $r \subseteq CS \times AS$ be a *retrieve relation*, relating each concrete state to the abstract states it represents. Then C **refines** A through r iff both of the following hold.
(i) *Applicability*: for all $(c, a) \in r$, if $a \in \text{dom}A$ then $c \in \text{dom}C$; as an inclusion, $r^{-1}[\text{dom}A] \subseteq \text{dom}C$. The implementation refuses no demanded case.
(ii) *Correctness*: for all $(c, a) \in r$ with $a \in \text{dom}A$ and all $(c, c') \in C$, there is an a' with $(a, a') \in A$ and $(c', a') \in r$. As an inclusion of relations from AS to CS, using Chapter 3's composition,

$$(C \circ r^{-1}) \cap (\text{dom}A \times CS) \subseteq r^{-1} \circ A,$$

that is, every concrete step taken from a represented abstract state within A's precondition is matched by an abstract step to a state that the concrete result represents. The implementation invents no forbidden behavior.

For a whole system, one adds *initialization*: every concrete initial state represents some abstract initial state, $CInit \subseteq r^{-1}[AInit]$. Informally and memorably: *fewer refusals, no surprises, and a lawful start*. This is the downward-simulation form of data refinement (Spivey 1992, Chapter 5; Abrial 1996, Chapter 11).

Refinement is reflexive and transitive: a preorder on specifications-and-implementations (Chapter 5). *Stepwise refinement* is the transitivity clause worn as

method: abstract spec, refined toward the concrete in verified steps, each small. The B method industrializes exactly this, with tool-discharged obligations per step. Its flagship deployments in rail interlocking (Paris Métro Line 14) are the standard demonstration that the discipline scales to safety-critical production. Three of its consequences organize practical reasoning.

Nondeterminism is specification freedom. An abstract operation may permit many outcomes ("return *some* free slot"), and refinement may narrow to one. Within the operation model of Definition 12.1, narrowing is sound whenever enabledness is kept: with $CS = AS$ and $r = \mathrm{id}$, any $C \subseteq A$ with $\mathrm{dom}C = \mathrm{dom}A$ refines A, and *widening* (adding a transition from a state in $\mathrm{dom}A$ that A forbids) never does. Two boundaries of the license must be stated. First of all, Definition 12.1 constrains C only on calls that satisfy the abstract precondition; what C does from states outside $\mathrm{dom}A$ is unconstrained by it, which is the usual contract reading (a demanded case must be honored, an undemanded one is the implementation's business) and is not a claim about whole-system behavior. Then, the model is one of partial correctness. In a trace model with liveness or fairness requirements, an arbitrary deterministic choice may violate them (always returning the *same* free slot may starve a waiting requester), so determinization does not transfer to full behavioral refinement without conditions on the choices made. Consequently, observed behavior of an

implementation underdetermines the specification. Two correct implementations may differ, and clients who depend on one's incidental choices (iteration order of a hash set, Chapter 11) are depending on non-contract. Refinement names precisely what they have done wrong, and "Hyrum's law" describes the sociology of it.

The retrieve relation is the representation. Linking `Map`-of-`Set` adjacency to edge-set relation (Section 11.2's isomorphism) or a sorted-array set to the abstract set it denotes: the retrieve relation is Chapter 4's abstraction function generalized to relations, and data refinement's obligations are the commuting conditions along it. A "the cache is a materialized view of the source of truth" architecture statement is a retrieve-relation claim, and cache-coherence bugs are its violated commuting conditions.

Behavior sets nest. Where systems are modeled by their sets of traces (TLA$^+$, process calculi), refinement is literal inclusion: traces($Impl$) $\subseteq$ traces($Spec$). The entire correctness claim of a distributed protocol implementation against its abstract specification is one $\subseteq$ between two typically infinite sets. It is not established by enumeration: an arbitrary infinite trace set need not be effectively enumerable, and the set of infinite traces is in general uncountable (Section 6.2), so enumeration is not even available in principle. It is established by invariant-and-simulation arguments of the same

family as Definition 12.1, but with stronger obligations, and the difference matters. Definition 12.1 constrains only the steps taken from states within the abstract precondition. Take one state space $\{0,1,2\}$ at both levels, $r = \mathrm{id}$, $A = \{(0,0)\}$, $C = \{(0,0), (1,2), (2,2)\}$, and initial state 1 at both levels: applicability, correctness, and initialization all hold, yet the concrete trace 1,2,2, … has no abstract counterpart, since A permits no step from 1. Whole-system trace refinement therefore asks for more: matching initial states, simulation of *every reachable* concrete step (with the stuttering and observation conventions of the trace model fixed in advance), and preservation of whatever liveness or fairness the specification requires (Lamport 2002, Sections 5.8 and 8.9.4; Abadi and Lamport 1991). Operation refinement is the per-call special case of this discipline, not a substitute for it. Inclusion of trace sets is this book's opening claim (set theory as the working language of exact system description) at its fullest applied stretch.

12.4 Alloy: bounded relational analysis

Proof obligations demand provers or people. **Alloy** trades certainty for automation, and what follows describes its core mode, *bounded relational analysis* (Jackson 2012, Chapters 4–5). An Alloy model declares signatures (sets), fields (relations), facts (axioms: invariants), and assertions (conjectures). The analyzer grounds everything over user-chosen finite bounds ("scope": at most k atoms per

signature), translates to SAT (Section 9.4's re-propositionalization, verbatim), and searches *exhaustively within scope* for counterexamples. The output is one of two: a counterexample (a concrete small structure violating the assertion: a bug, rendered as a diagram), or "no counterexample within scope," which establishes the assertion for every structure within the declared bounds and says nothing about larger universes. It is *not* a proof, and the tool's culture is scrupulous about saying so. The methodological wager is the **small scope hypothesis**: most real design flaws already appear in tiny instances. Section 12.2's trap needs one person, two zones, and one one-way door. The hypothesis is empirical, well supported across published case studies, and philosophically congenial to this book. It claims that design errors are usually *structural* (wrong quantifier, missing frame condition, unclosed relation), and structural errors, living in the shape of the set-theoretic model rather than in magnitudes, manifest at small cardinalities. Where it fails (errors that require large or specifically sized instances: off-by-one at capacity boundaries, modular arithmetic collisions), bounded search is silently blind, and the practitioner supplements with targeted bounds or proof.

Note (bounds on atoms versus bounds on time). The account above is version-independent only for the relational core. Alloy 6 adds mutable state and a temporal logic, and its analyzer can check temporal assertions over an *unbounded* time horizon by

complete model checking while the signature scopes remain finite (Alloy Tools 2021). Two different bounds are therefore in play in a modern Alloy analysis, atoms and steps, and a completed check is to be read against both.

Alloy versus B is thus Chapter 9's boundary made into a tooling decision: decidable exhaustion within bounds versus undecidable-in-general proof with human effort. Mature practice uses the first to debug the model cheaply and the second where the stakes demand certificates.

12.5 Specification in the vernacular

The full formal method is one end of a spectrum, and the chapter's concepts pay rent at every point of it. Preconditions and postconditions in API documentation are $pre/effect$ clauses in prose. Property-based tests (Section 11.1) are invariant-preservation obligations sampled rather than proved. Database constraints (Chapter 10) are invariants a DBMS discharges continuously. Idempotency keys promise $op \circ op = op$ (an algebraic law, checkable). "Backwards compatible change" is refinement's applicability clause applied to an API's demanded cases. Schema evolution rules ("add optional fields only") are sufficient syntactic conditions for a semantic inclusion between message sets. Teams using no formal notation still make every one of these claims. The notation's marginal offer is exactness at the moments

(concurrency, failure, migration) when prose answers collapse under case analysis. The applied skill is not fluency in Z's schema calculus. It is the reflex of asking: *what is the state space, what is the invariant, what does each operation require and change, what may the client observe?* These four questions have answers that are sets, an inclusion, relations, and a refinement claim respectively, whatever concrete syntax they are written in.

12.6 Summary

Specifications model state as sets-with-functions, operations as relations on states, correctness as invariants. Proof obligations (preservation, precondition adequacy, initialization with $Init \neq \emptyset$ and $Init \subseteq Inv$) mechanize coherence checking and, routinely, surface missing assumptions and real design flaws, as the one-way door did. Refinement, a preorder of inclusions (fewer refusals, no surprises, a lawful start) through a retrieve relation, is the exact content of "correctly implements." Within the operation model it makes enabledness-preserving determinization sound and widening within the abstract precondition unsound; whole-system trace inclusion needs the stronger simulation obligations on every reachable step, and liveness adds conditions of its own. Alloy's bounded relational analysis grounds models over small scopes into SAT for exhaustive counterexample search under the small scope hypothesis: automation purchased at the price of "within scope," with Alloy 6 separating

bounds on atoms from bounds on time. The B tradition discharges obligations by proof and has carried the discipline into safety-critical production. The four specification questions, asked in any notation, are the chapter's portable core.

Chapter 13. Counting in Practice: Combinatorics and Inclusion–Exclusion

Dependencies: Chapters 2, 4–6. Enumerative combinatorics is the arithmetic of finite set constructions: every counting rule is a cardinality statement about a sum, product, power set, function space, or quotient built in Part I. In this chapter, we derive the rules from the constructions (so they transfer instead of being memorized), state inclusion–exclusion exactly, quantify the pigeonhole principle into the birthday bound, and work the engineering estimates (key spaces, test matrices, collision risks, capacity) that counting exists to serve.

13.1 The four bijective rules

All elementary counting rests on Chapter 6's definition ($|A| = n$ means $A \approx [n]$) plus four facts about constructions, each grounded in a bijection or a partition.

Theorem 13.1 (Sum, product, power, quotient). For finite sets:
(i) *Sum rule:* $A \cap B = \emptyset \Rightarrow |A \cup B| = |A| + |B|$; generally, a partition's blocks' sizes sum to the whole (Theorem 5.1 with addition).
(ii) *Product rule:* $|A \times B| = |A| \cdot |B|$; iterated, sequential independent choices multiply.
(iii) *Power rules:* $|B^A| = |B|^{|A|}$ (Definition 4.6) and

$|\mathcal{P}(A)| = 2^{|A|}$ (Theorem 2.2).
(iv) *Quotient rule (counting by symmetry):* if an equivalence on a finite set S has all classes of size exactly k, where $k \geq 1$, then $|S/\sim| = |S|/k$.

The quotient rule deserves its unfamiliar billing. It is the *license to divide*, and every "divide by symmetry" argument is an appeal to it. The appeal is valid exactly when the classes are equal-sized, which is the hypothesis to check before dividing. Its failure is why "necklace" countings are subtler than naive division, and why Burnside's lemma exists. We flag and defer that refinement.

Theorem 13.2 (The classical counts). For an n-set: (i) sequences of length k with repetition: n^k; (ii) without repetition (injections $[k] \to [n]$): $n^{\underline{k}} = n(n-1)\cdots(n-k+1)$; (iii) permutations: $n!$; (iv) k-subsets: $\binom{n}{k} = n^{\underline{k}}/k!$.

The derivation of $\binom{n}{k}$ *is* the method: count an easier set (ordered selections), recognize the target as its quotient, verify equal class sizes, divide. Sequences-versus-sets (Chapter 1's order-and-multiplicity distinctions) thus become arithmetic. The four combinations of (ordered?, repetition?) give n^k, $n^{\underline{k}}$, $\binom{n}{k}$, and $\binom{n+k-1}{k}$ (multisets, Chapter 16's counting face; derived, for $n \geq 1$, by the stars-and-bars bijection: a multiset of size k over $[n]$ is exactly a string of k stars and $n-1$ bars; Stanley 2011, Section 1.2).

Example 13.1 (Key spaces and entitlement matrices). A password of exactly 12 characters over a 94-symbol printable alphabet inhabits a set of size $94^{12} \approx 4.8 \times 10^{23}$ (power rule). The same rule prices every keyspace an attacker enumerates: 6-digit PINs at 10^6, UUID v4 at 2^{122}, a 64-bit hash at 2^{64}. An RBAC deployment with 40 roles assigns each user a role *set*: 2^{40} possible assignments (power set). This is why "list all realized entitlement combinations" is an audit of observed rows rather than an enumeration of the possibility space. A test matrix over 7 boolean flags and 3 environments has $2^7 \cdot 3 = 384$ cells (product rule). *Pairwise* coverage (every pair of parameter values co-occurring somewhere) collapses it to a few dozen cases, the combinatorial-design trade Chapter 2 promised: exhaustiveness priced by the product rule, bought down by accepting coverage of projections (Chapter 4's images) instead of the full product.

13.2 Inclusion–exclusion

The sum rule demands disjointness. Inclusion–exclusion repairs the overlap exactly.

Theorem 13.3 (Inclusion–exclusion). For finite sets $A_1, \ldots, A_n$:

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{\emptyset \neq S \subseteq [n]} (-1)^{|S|+1} \left|\bigcap_{i \in S} A_i\right|.$$

Inclusion–exclusion is not an approximation refined by corrections but an exact identity: the signed terms cancel so that each element of the union is counted exactly once. Its truncations bracket the truth (Bonferroni inequalities: odd-length prefixes overcount, even-length undercount; Feller 1968, Chapter IV). The truncated form is the applied form in which the identity usually earns money, since high-order intersections are the expensive ones to measure.

Example 13.2 (Deduplicated reach). Three alerting channels page overlapping on-call sets: $|E| = 40$, $|S| = 35$, $|P| = 30$; pairwise overlaps 18,12,15; triple overlap 9. Distinct people paged: $40 + 35 + 30 - 18 - 12 - 15 + 9 = 69$, not 105. The sum rule's disjointness hypothesis fails, and by how much is precisely what the correction terms record. The same computation with rates instead of counts is the "unioned conversion" and "deduplicated audience" arithmetic of every analytics platform. Its silent failure mode is a data pipeline that can measure the pairwise overlaps but not the triple. What can then be said depends on which information is used. The first two Bonferroni truncations alone give $105 - 45 = 60$ as a lower bound and 105 as an upper bound. The measured pairwise overlaps sharpen the bracket considerably: writing t for the unknown triple overlap, the union has exactly $60 + t$ people, and t is constrained by $0 \le t \le 12$, the smallest pairwise overlap, since the triple region lies inside each pairwise one. Both

extremes are feasible (at $t = 12$ the seven regions of the Venn diagram can hold 22,14,15,6,0,3,12 people and reproduce every stated count), so the honest bracket is *between* 60 *and* 72. The known answer 69 is not available as an upper bound in the scenario where the input that determines it is missing.

Example 13.3 (Derangements and the structure of "no fixed point"). Permutations of $[n]$ with no fixed point number $D_n = n!\sum_{k=0}^{n}(-1)^k/k!$: inclusion–exclusion over the sets A_i = "permutations fixing i". Since $\sum(-1)^k/k! \to e^{-1}$, the proportion of derangements tends to $1/e \approx 0.368$. Shuffle gift tags at any large party, and the chance nobody gets their own settles near 36.8%, essentially independent of n. This is a first instance of a running Part III theme: exact combinatorics stabilizing into the probabilistic limits of Chapter 14 (where $1/e$ returns as the Poisson zero-class and as the balls-in-bins empty-bin fraction, the same phenomenon three ways).

13.3 Pigeonhole, quantified: collisions and the birthday bound

Chapter 6's pigeonhole principle guarantees a collision past $|B|$ items. Applications need to know *how soon collisions become likely*, which is a counting question with a famous answer.

Theorem 13.4 (Birthday bound). Draw k items independently and uniformly from an n-set. The probability that all are distinct is

$$p_{\text{distinct}}(k, n) = \prod_{i=0}^{k-1} \left(1 - \frac{i}{n}\right) \leq e^{-k(k-1)/(2n)},$$

so collisions reach probability $1/2$ near $k \approx 1.18\sqrt{n}$, and $\sqrt{n}$ (not n) is the scale at which uniform random identifiers start colliding (Feller 1968, Section II.3; Mitzenmacher and Upfal 2017, Section 5.1).

Note. Probabilistic language here is a convenience. With all outcomes equally likely, every probability in sight is a ratio of two Part I cardinalities: $n^{\underline{k}}$ injective outcome sequences among n^k in all. Chapter 14 supplies the general framework.

The square-root scale is a first-class engineering constant, always under the theorem's hypotheses of independent, uniform draws. A 32-bit hash ($n = 2^{32}$) sees even-odds collision near 77,000 items, which is why 32 bits is not an identifier space. 64 bits moves the threshold to $\approx 5 \times 10^9$: fine for many systems, marginal for global scale. A version-4 UUID has 128 bits but only 122 random ones, the rest being fixed version and variant fields (RFC 9562, Section 5.4), so $n = 2^{122}$ as in Example 13.1 and the even-odds threshold is $k_{1/2} \approx 1.18 \cdot 2^{61} \approx 2.7 \times 10^{18}$ identifiers (the figure 2.2×10^{19} would apply to 128 independent uniform bits). The right engineering statement is relative rather than absolute. For k identifiers the collision probability is about $k^2/2^{123}$, so a system that will ever mint 10^{12} of them faces roughly 10^{-13}, which is why the failure

mode is usually dismissed, and a system minting 10^{17} faces about 10^{-3}, which is not dismissible. The identical arithmetic prices hash-partition skew onset, git short-hash ambiguity (7 hex digits: $n = 2^{28}$, collisions among a few tens of thousands of objects; this is why git lengthens abbreviations as repositories grow), and cryptographic birthday attacks, where the defender must budget for adversarial rather than accidental collision-seeking and thus doubles digest lengths (2^{128} work against a 256-bit hash is the design point, not 2^{256}).

13.4 Double counting and combinatorial identities

Counting one set two ways establishes identities without algebra. The technique is Part I's extensionality operationalized: one set, two enumerations, equal cardinalities.

Theorem 13.5 (Specimens). (i) $\sum_{k=0}^{n}\binom{n}{k} = 2^n$: both sides count $\mathcal{P}([n])$, by subset size (partition + sum rule) and by the power rule. (ii) $\binom{n}{k} = \binom{n}{n-k}$: complementation $X \mapsto [n] \setminus X$ is a bijection (an involution) between k-subsets and $(n-k)$-subsets. (iii) Pascal's rule $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$: partition the k-subsets by whether they contain the element n. (iv) Handshake lemma: in any finite simple graph (no loops, no multiple edges), $\sum_v \deg(v) = 2|E|$: count incidences (edge, endpoint) by edge and by

vertex; hence the number of odd-degree vertices is even.

Double counting via an incidence relation (build $I \subseteq A \times B$, count fibers both ways, equate) is among the most transferable reasoning patterns in applied mathematics. It verifies aggregate consistency in data systems (total order-lines summed by order equals summed by product; a mismatch *proves* a data defect). It bounds average degrees in dependency graphs. It underlies the amortized analyses of algorithms: "charge each expensive step to many cheap ones" is a fiber-counting argument about an incidence set of charges.

13.5 Estimation discipline

Applied counting's deliverable is usually a *magnitude with a purpose*: feasible or not, safe or not. Four habits keep the magnitudes honest. **Bound both sides**: for $1 \le k \le n$, $\left(\frac{n}{k}\right)^k \le \binom{n}{k} \le \frac{n^k}{k!} \le \left(\frac{en}{k}\right)^k$ brackets binomials without evaluation, and Stirling's approximation $n! \sim \sqrt{2\pi n}\,(n/e)^n$ prices factorials (Graham, Knuth, and Patashnik 1994, Chapter 9). **Work on the log scale first**: $\log_2 52! \approx 226$ bits is why a shuffled deck's order cannot be stored in a 64-bit word, and why "seed a shuffle from a 32-bit RNG state" provably reaches a vanishing fraction of permutations, an audit-grade conclusion from counting alone. **Name the model**: the exact cardinality identities above assume nothing

probabilistic, but every *estimate* of collision or likelihood assumed uniformity or independence somewhere. Identifier-collision estimates are wrong for identifiers that are *not* uniform (timestamps embedded in "random" IDs collapse the effective n), and stating the assumed set construction is what makes an estimate checkable. **Respect the exponent**: the recurring applied failure is not miscounting by a factor but mistaking a polynomial for an exponential. The power-set and power-rule constructions are the exponential factories, and recognizing one at design time (per-flag configs, per-subset caching, per-permutation testing) is worth more than any downstream optimization.

13.6 Summary

Finite counting is cardinality arithmetic over Part I's constructions: sums over partitions, products over tuples, powers over function spaces and subsets, quotients where symmetry classes are equal-sized. The classical formulas are derived, not memorized, and the ordered/repetition fourfold is its multiplication table. Inclusion–exclusion exactly corrects the sum rule's disjointness debt, with Bonferroni truncations bracketing when high-order overlaps are unmeasured. The birthday bound quantifies pigeonhole at the $\sqrt{n}$ scale that governs identifier and digest sizing. Double counting of incidence sets establishes identities and audits aggregates. Estimation discipline (two-sided bounds,

named models, exponent vigilance) turns counts into engineering judgments.

Chapter 14. From Sets to Chance: Sample Spaces, σ-Algebras, and Measure

Dependencies: Chapters 2, 4, 6; Chapter 13 for the finite case. Probability theory is set theory with a measure attached. Events are sets, logical combinations of events are Boolean operations, random variables are functions, and conditioning is universe restriction. The celebrated subtleties (why σ-algebras, why "almost surely," why some sets have no probability) are cardinality and choice phenomena from Part II wearing applied dress. In this chapter, we build the framework in Kolmogorov's axioms and read each applied convention back to its set-theoretic ground.

14.1 Finite probability is weighted counting

Definition 14.1 (Finite probability space). A finite *sample space* is a finite set Ω of *outcomes*; an **event** is any subset $E \subseteq \Omega$; a *probability distribution* is $p: \Omega \to [0,1]$ with $\sum_{\omega} p(\omega) = 1$, extended to events by $\Pr[E] = \sum_{\omega \in E} p(\omega)$. Uniform p makes $\Pr[E] = |E|/|\Omega|$: Chapter 13's ratios, officially.

The dictionary of Chapter 9 specializes wholesale. "Not E" is E^c with $\Pr[E^c] = 1 - \Pr[E]$. Combinations of events are $\cap$ and $\cup$, with inclusion–exclusion

(Theorem 13.3) transferring verbatim to probabilities (the same signed cancellation, with per-element counts becoming per-outcome weights). Implication becomes $E \subseteq F \Rightarrow \Pr[E] \leq \Pr[F]$ (*monotonicity*), whose everyday use is bounding a complicated event by a simpler superset. A second, separate consequence of additivity is *subadditivity*, the **union bound** $\Pr[\cup E_i] \leq \sum \Pr[E_i]$, and the two are often used together: enclose the bad event in a union of simple ones, then sum. The union bound is the single most-used inequality in randomized systems analysis: crude, assumption-free (no independence needed; it is Boole's inequality, Bonferroni's first truncation), and routinely sharp enough to size retry budgets, error rates, and failure-tolerance margins.

Definition 14.2 (Conditioning, independence). For $\Pr[F] > 0$: $\Pr[E \mid F] = \Pr[E \cap F]/\Pr[F]$: probability with the universe cut down to F and weights renormalized, Chapter 2's relative complement discipline as inference. Events are *independent* iff $\Pr[E \cap F] = \Pr[E]\Pr[F]$.

Conditioning is where set discipline pays most visibly. The universe-shift errors of Section 2.1 return as *base-rate neglect* (comparing $\Pr[E \mid F]$ with $\Pr[F \mid E]$: different conditioning universes, related only through Bayes' rule $\Pr[F \mid E] = \Pr[E \mid F]\Pr[F]/\Pr[E]$) and as *selection effects* (computing frequencies over the subset that survived a filter while narrating them as facts about

the whole; the set F is the analyst's silent choice, and naming it is the repair). Independence, meanwhile, is a *hypothesis about the measure*, never about the sets. Disjoint events of positive probability are maximally *dependent*: one occurring forbids the other. The persistence of this confusion in practice justifies the definitional pedantry.

14.2 Why infinite probability needs σ-algebras

Infinite sample spaces force the question Part II equipped us to answer: *which subsets get probabilities?* The natural demand (all of $\mathcal{P}(\Omega)$) fails for the spaces applications most need.

The obstruction. Consider $\Omega = [0,1)$ and ask for a measure μ with four properties: it is defined on *every* subset of Ω; it is countably additive; it is normalized, $\mu([0,1)) = 1$; and it is translation-invariant (shifting a set mod 1 preserves its measure). *No such μ exists.* Vitali's construction shows why (Billingsley 1995, Section 3). Partition $[0,1)$ by the equivalence $x \sim y$ iff $x - y \in \mathbb{Q}$ (Chapter 5). Use the Axiom of Choice to select one representative per class into a set V. Put $Q_0 = \mathbb{Q} \cap [0,1)$ and, for $q \in Q_0$, let $V_q = \{(v + q) \bmod 1 : v \in V\}$, the translate of V by q modulo 1 (indexing by all of $\mathbb{Q}$ would list each translate infinitely often, since q and $q + 1$ give the same set). The family $(V_q)_{q \in Q_0}$ is pairwise disjoint (a point in two translates would

put two representatives in one class) and covers $[0,1)$ (each x differs from its class's representative by a rational, reduced modulo 1 into Q_0). Translation invariance gives $\mu(V_q) = \mu(V)$, so countable additivity forces $1 = \sum_{q \in Q_0} \mu\,(V_q) = \sum_{q \in Q_0} \mu\,(V)$, which is impossible whether $\mu(V)$ is zero (the sum is 0) or positive (the sum diverges). Every ingredient is Part I–II stock: a quotient, a choice function, countable additivity meeting $\aleph_0$. All four hypotheses are needed. Infinity alone is not the obstruction: an infinite Ω carries perfectly good measures on its full power set, for example the *Dirac measure* concentrated at one point ($\mu(E) = 1$ iff $\omega_0 \in E$), which is countably additive and normalized but not translation-invariant. The conclusion is not that probability breaks but that **total measurability was not on offer for the uniform-like measure**. Some sets must fall outside the system. The applied reassurance must be stated with equal precision. Intervals and all their countable Boolean combinations are measurable (they are the Borel sets of Definition 14.3), and completing the measure admits further sets (the Lebesgue measurable ones). Beyond that, a set given by a formula or a program is not thereby a countable combination of intervals, and its measurability has to be established rather than presumed; whether a non-measurable set can be defined by a formula without parameters depends on axioms beyond ZFC (Section 8.3). For everyday modeling the reassurance holds in practice, because the sets one specifies are built

from intervals by countably many Boolean operations and limits, and that is the operational statement.

Definition 14.3 (σ-algebra, measure, probability space). A **σ-algebra** on Ω is a family $\mathcal{F} \subseteq \mathcal{P}(\Omega)$ containing Ω and closed under complement and *countable* union (hence countable intersection, by De Morgan). A **probability measure** is $\Pr: \mathcal{F} \to [0,1]$ with $\Pr[\Omega] = 1$ and countable additivity: for pairwise disjoint $E_1, E_2, \ldots \in \mathcal{F}$, $\Pr[\bigcup_i E_i] = \sum_i \Pr[E_i]$. The triple $(\Omega, \mathcal{F}, \Pr)$ is a *probability space* (Kolmogorov's axioms; Billingsley 1995, Section 2). The σ-algebra *generated* by a family $\mathcal{G}$ is the least σ-algebra containing it. Its existence follows by the intersect-all-candidates argument of Theorem 3.2(i), the third time this book has built a least closed structure by that route. On $\mathbb{R}$, the σ-algebra generated by the intervals is the **Borel** σ-algebra. It contains every set reachable from intervals by countably many Boolean operations and limits, which covers the sets ordinary modeling specifies, though not every set a formula can define (Billingsley 1995, Section 2).

Why *countable* closure, not finite, not arbitrary? The calibration is Chapter 6's. Finite additivity is too weak to price limit events. "The retry loop eventually succeeds" is a countable union of stage successes, and its probability should be the limit of the partial ones: that *is* countable additivity, in the form of continuity from below. Closure under

arbitrary unions would readmit every subset (any E is a union of its singletons) and reconvene Vitali. Countability is the exact width of the needle's eye: big enough for limits, asymptotics, and "eventually"; small enough to dodge the non-measurable. The reader should hear $\aleph_0$ load-bearing in both clauses.

Definition 14.4 (Almost surely). An event $E \in \mathcal{F}$ holds *almost surely* (a.s.) iff $\Pr[E] = 1$.

Almost-sure is not sure. A uniform draw from $[0,1)$ misses any fixed point with probability 1 yet lands somewhere. Continuous models thus systematically assign probability zero to every exact outcome: the measure-zero sliver of Chapter 6 (countable and even some uncountable sets weigh nothing). Applied fluency includes not over-reading a.s. statements. "The estimator converges almost surely" quantifies over probability, not over adversarial or out-of-model inputs, and the zero-probability exception set is where numerical edge cases, adversarial examples, and "can't happen" incidents habitually live. A discipline as small as pronouncing "with probability 1" instead of "always" keeps the epistemics straight.

14.3 Random variables are functions; distributions are pushforwards

Definition 14.5 (Random variable, distribution). A **random variable** on $(\Omega, \mathcal{F}, \Pr)$ with values in a measurable space $(S, \mathcal{S})$ is a function $X: \Omega \to S$ that is

measurable: $X^{-1}[B] \in \mathcal{F}$ for every $B \in \mathcal{S}$. Its **distribution** is the pushforward measure $\Pr_X[B] = \Pr\left[X^{-1}[B]\right]$ on S.

No randomness resides in X. It is a plain function (Chapter 4); the chance lives in the measure on Ω, and X merely transports it. Measurability is exactly the demand that questions about outputs pull back to priced events, and its good behavior is Theorem 4.3 cashing in: *preimage commutes with all Boolean operations*. This is why $\{X \in B_1 \cup B_2\}$, $\{X \notin B\}$, and their countable elaborations are automatically events, why the pullback $\{X^{-1}[B]: B \in \mathcal{S}\}$ is itself a σ-algebra (the *σ-algebra generated by X*: the formal object behind "the information revealed by observing X"), and why composing measurable maps stays measurable while *images* enjoy no such guarantees. Chapter 4's asymmetry between image and preimage, introduced as a data-pipeline curiosity, turns out to be the load-bearing wall of the entire measure-theoretic edifice.

Example 14.1 (The modeling stack, assembled). A load balancer sprays k requests uniformly and independently over n servers. Sample space: $\Omega = [n]^{[k]}$, the function space (Definition 4.6), $|\Omega| = n^k$, uniform measure ("independently" *is* the product structure). Random variables: $L_j = |f^{-1}[\{j\}]|$, the j-th server's load, a preimage cardinality, Chapter 4 verbatim. Then: $\Pr[\text{server } j \text{ idle}] = (1 - 1/n)^k \to e^{-k/n}$ (the $1/e$ of Example 13.3 at $k = n$); expected maximum load at $k = n$ grows as $\Theta(\log n/\log\log n)$

(stated, not derived here; Mitzenmacher and Upfal 2017, Section 5.2); and the birthday computation of Theorem 13.4 is $\Pr[\text{all } L_j \leq 1]$. One paragraph, and every object in it (function space, product, preimage, cardinality ratio) is Part I machinery. The probabilistic layer added only the measure and the limit theorems. This is the chapter's thesis in miniature: *probability models are set constructions plus weights*, and mastery of the constructions is most of model literacy.

Expectation assembles a random variable's values against its measure: $\mathbb{E}[X] = \sum_{\omega} X(\omega)\, p(\omega)$ in the finite case, with **linearity** ($\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$, unconditionally, independence not required) as the workhorse. Linearity plus indicator variables ($\mathbf{1}_E$, the characteristic functions of Chapter 2, with $\mathbb{E}[\mathbf{1}_E] = \Pr[E]$) mechanizes otherwise painful counts. For example, let C be the number of colliding unordered pairs among k draws $X_1, \ldots, X_k$ from $[n]$. By linearity alone, $\mathbb{E}[C] = \sum_{i<j} \Pr[X_i = X_j]$, with no assumption about the joint draws. Pairwise independence together with uniformity makes each term $1/n$, giving $\mathbb{E}[C] = \binom{k}{2}/n$. Uniform marginals alone do not fix the terms: two draws that are always equal are each uniform and collide with certainty. Nor is independence necessary: for uniform X on $[n]$ and $Y = f(X)$ with f a permutation of $[n]$ fixing exactly one value, Y is uniform and completely determined by X, yet $\Pr[X = Y] = 1/n$ exactly. Independence is a sufficient condition for

the $1/n$ terms, and the one that applications can usually justify. (Three equal values count as three colliding pairs, and as two duplicates beyond the first; the two counts answer different questions.) The expected pair count becomes of order one when k is of order $\sqrt{n}$. That is the birthday *scale* recovered in one line, and under mutually independent uniform draws Theorem 13.4 shows that it is also the scale at which *some* collision has substantial probability. The second conclusion does not follow from pairwise independence alone. For a prime q, choose $2 \le k \le q$ distinct elements $t_1, \dots, t_k$ of the field $\mathbb{F}_q$, let A, B be independent and uniform on that field, and put $X_i = At_i + B$ with arithmetic modulo q. The outputs are pairwise independent and uniform (two prescribed outputs at distinct inputs determine A and B uniquely). They all coincide when $A = 0$ and are all distinct otherwise, since $X_i = X_j$ iff $A(t_i - t_j) = 0$, so $\Pr[C > 0] = 1/q$ for every k in this range, while $\mathbb{E}[C] = \binom{k}{2}/q$ grows with k. Expected pair counts price collision *workload* (hash-bucket overflow, duplicate-detection load, shard skew) without a single distributional calculation. The birthday and maximum-load results cited here (Theorem 13.4, Example 14.1) assume mutually independent uniform draws; other joint laws require their own probability and load analysis, as the affine example shows. The indicator method (*decompose the count into a sum of* $\{0,1\}$*-valued variables over a set you can enumerate*) is Chapter 13's incidence-set double counting with a measure

attached, and it is the single most transferable computation pattern in applied probability.

14.4 Independence, product spaces, and the perils of the joint

Independent components are modeled by **product spaces** $(\Omega_1 \times \Omega_2,\ \mathcal{F}_1 \otimes \mathcal{F}_2,\ \Pr_1 \otimes \Pr_2)$. The product σ-algebra is generated by the measurable rectangles,

$$\mathcal{F}_1 \otimes \mathcal{F}_2 = \sigma(\{E_1 \times E_2 : E_1 \in \mathcal{F}_1,\ E_2 \in \mathcal{F}_2\})$$

(not by $\mathcal{F}_1 \times \mathcal{F}_2$ itself, which under Definition 2.9 is a set of *pairs of sets*), and the product measure is determined on those rectangles by $\Pr[E_1 \times E_2] = \Pr_1[E_1]\Pr_2[E_2]$: Chapter 2's products carrying the multiplication that "independent" promises. Everything real about the model lives in the choice of *joint* space. Marginals constrain it without, in general, determining it: the same two fair-coin marginals arise from independent flips, from one coin copied, and from one coin anti-copied (three joints, one pair of marginals), and only degenerate marginals pin the joint down. What the marginals do give is a dependence-free bracket. For failure events A and B with marginal probabilities p and q,

$$\max(p, q) \le \Pr[A \cup B] \le \min(1,\ p + q),$$

the upper bound being the union bound of Section 14.1 and the lower bound monotonicity. In applied terms, this is why correlated failures defeat

redundancy calculations premised on independence ("five nines from three replicas" assumes the product measure, under which the joint failure probability is the product p^3; a shared power bus, deploy pipeline, or certificate authority installs a very different joint, under which it may be as large as p), and why portfolio and capacity models stand or fall on dependence structure. "We validated each component" therefore bounds the system in the weak sense of the display and no more: it justifies the bracket, not the product. The set-theoretic frame makes the error legible. A reliability claim is a statement about a measure on the *product* space, and naming which measure (not just which marginals) is the modeling act.

Note (interpretation). Everything above is mathematics: theorems about measures on set structures. What probability *means* (long-run frequency, degree of belief, propensity) is an interpretive layer the mathematics serenely underdetermines (compare Section 8.4: one calculus, several semantics). Applied practice mixes interpretations daily (a frequentist SLO beside a Bayesian prior beside a load-test's empirical distribution), usually harmlessly, occasionally not. The mixture is worth noticing when "the probability of this specific outage" is demanded of a framework that prices only ensembles, or when a belief update is audited as if it were a frequency claim. The set-theoretic core is the shared, uncontested stratum.

14.5 Summary

Finite probability is weighted counting. Events are subsets, inclusion–exclusion and monotonicity transfer, the union bound prices compound failure without independence assumptions, and conditioning is universe restriction with renormalization, with base-rate and selection errors being universe-shift errors. A uniform-like measure on an interval cannot price every subset (Vitali: quotient + choice + countable additivity + translation invariance), so probability lives on σ-algebras, closed under exactly *countable* operations, the calibrated width that admits limit events while excluding the non-measurable. Borel sets cover the countable interval combinations that ordinary modeling specifies, measurability beyond them is established rather than presumed, and "almost surely" flags the measure-zero remainder where edge cases dwell. Random variables are measurable functions transporting the measure by preimage (Theorem 4.3 as foundation), distributions are pushforwards, information is a generated σ-algebra, and linearity plus indicators converts Part I set decompositions into expectations. Independence is product measure: a property of the joint that marginals constrain but in general do not determine, which is where redundancy arithmetic and validation claims quietly break.

Chapter 15. Graphs and Networks as Set Systems

Dependencies: Chapters 3–5 (relations, functions, orders), 7 (fixpoints, well-foundedness), 13 (counting). A graph is a relation drawn as a picture, and a network model is a set system with structure worth naming. In this chapter, we ground graph vocabulary in the relational definitions already established, develop reachability and connectivity as closure and equivalence, DAGs as the order-theoretic case, trees as well-founded recursion's natural habitat, and set systems (hypergraphs) as the general form, with the applied translations (dependency analysis, network partitioning, schema shapes, cascade reasoning) carried throughout.

15.1 Graphs are relations, curated

Definition 15.1 (Directed graph). A *directed graph* (digraph) is a pair $G = (V, E)$ with V a finite set of *vertices* and $E \subseteq V \times V$ a binary relation of *edges*. An **undirected graph** takes E to be a set of two-element subsets $\{u, v\}$ of V with $u \neq v$ (equivalently, a symmetric irreflexive relation quotiented by the pairing, Definition 5.2's identification). Undirected graphs in this chapter are therefore *simple*: no loops and no multiple edges. Digraphs may carry loops (v, v), and where a loop matters we say so.

Nothing here is new mathematics. A digraph *is* Chapter 3's relation-on-a-set, and the three representations of Section 3.4 (tuple set, boolean matrix, adjacency function) are the three storage layouts every graph library offers. What graph theory adds is a curated vocabulary for the relation's *emergent* structure: paths, components, cycles, cuts, that is, the properties that live not in single tuples but in their compositions and closures. The applied justification for the vocabulary is that systems are graphs at every scale: call graphs, dependency graphs, data lineage, network topologies, social and organizational structures, state machines (Chapter 12's operations as edge relations on state spaces), and the entity-relationship shape of any database (Chapter 10's foreign keys as edges between row sets).

Definition 15.2 (Walk, path, cycle, reachability). In a digraph, a *walk* from u to v is a finite sequence $u = x_0, x_1, \dots, x_k = v$ with each $(x_i, x_{i+1}) \in E$; its *length* is k, and vertices may repeat. In an undirected graph the condition reads $\{x_i, x_{i+1}\} \in E$, and the relational formulas below are applied to the *adjacency relation* $R_E = \{(u, v) \in V \times V : \{u, v\} \in E\}$, which is symmetric and irreflexive; we write E for R_E in them when no confusion can arise. A **path** is a walk with no repeated vertex. A *cycle* in a digraph is a closed walk $x_0, \dots, x_k = x_0$ of length $k \geq 1$ whose vertices $x_0, \dots, x_{k-1}$ are distinct (so a loop is a cycle of length 1); in a simple undirected graph a cycle has $k \geq 3$, since $k = 1$ would need a loop and $k = 2$

would traverse one edge back and forth, which repeats the edge rather than closing a cycle. Vertex v is **reachable** from u iff some walk connects them, that is, iff uE^*v (in the undirected case uR_E^*v), the reflexive–transitive closure of Definition 3.6. Since a walk that repeats a vertex can be shortened by cutting out the repeated stretch, reachability by walks and reachability by paths coincide (Diestel 2017, Chapter 1).

Reachability *is* closure: Theorem 3.2's chain characterization says so verbatim (its chains are walks), and its two characterizations are the two algorithm families. The intersection-of-closed-supersets view justifies invariant-based reasoning ("the reachable set is the least set containing the roots and closed under edges": Section 7.5's least fixpoint, computed by every BFS/DFS/worklist algorithm and by every garbage collector's mark phase). The chain view gives path-finding its objects. The **distance** $d(u, v)$ (least path length) refines reachability with a cost, and BFS computes it because path length is exactly closure-iteration stage: v has distance $\leq k$ iff $v \in$ the k-th iterate. This equation identifies an algorithm's loop counter with a mathematical stratification (Chapter 7's rank, in miniature).

15.2 Connectivity as equivalence; components as quotient

Definition 15.3 (Connectivity). In an undirected graph, $u \sim v$ iff a path joins them; the equivalence classes (Theorem 5.1) are the **connected components**. In a digraph, $u \approx v$ iff each reaches the other ($uE^*v \wedge vE^*u$); the classes are the **strongly connected components** (SCCs).

Both relations are equivalences by construction (reflexivity from the empty walk, transitivity from walk concatenation, symmetry by hypothesis or by the biconditional). Therefore, the full Chapter 5 apparatus applies with no further argument: components partition the graph, "number of components" is a quotient cardinality, and component labels are the canonical projection. Two applied dividends follow immediately. First, the **condensation**: collapsing each SCC to a single vertex. Its vertex set is the quotient $V/\approx$ and its edge set is

$$E_{\text{cond}} = \{\big(\pi(u), \pi(v)\big) : (u, v) \in E,\ \pi(u) \neq \pi(v)\}.$$

The restriction $\pi(u) \neq \pi(v)$ is essential: the plain image of E under $\pi \times \pi$ would turn every edge inside a component into a self-loop on the quotient vertex (the two-cycle $a \to b \to a$ has one component, and its image would be a loop on it). With loops discarded, the condensation is always a DAG, since a cycle among distinct components would merge

them. Therefore, every cyclic system has a canonical two-level description (acyclic architecture among clusters of mutual dependence), and "break the cycles" refactoring is the demand that every SCC be a singleton *and* that no vertex carry a self-loop, since a self-loop is a cycle on a singleton component. Second, *incremental connectivity* is the union–find data structure maintaining a partition under class-merging: Theorem 5.1 as a mutable object, with path compression as memoized projection. Its near-constant amortized cost is why "are these two in the same cluster yet?" is cheap at scale (network partition detection, percolation simulation, Kruskal's spanning-tree algorithm).

Example 15.1 (Partitions of the network kind). A distributed system's "network partition" is literally Definition 15.3: the reachability equivalence on live nodes fractures into multiple classes, and every consistency claim must now specify *per class* (which side holds quorum: a cardinality comparison between a class and $|V|/2$). The CAP-theorem conversation is conducted entirely in this chapter's nouns. A partition is a quotient with ≥ 2 blocks; "availability during partition" quantifies over blocks; healing is the classes re-merging. Precision about which equivalence relation (physical link liveness, application-level reachability, quorum visibility) generates the partition dissolves a standard class of postmortem ambiguity.

15.3 DAGs, orders, and the shapes of hierarchy

Theorem 15.1. A digraph is acyclic (it has no cycle, equivalently no closed walk of length ≥ 1) iff E^+ is irreflexive, in which case E^* is a partial order on V. Conversely, every finite poset is the reflexive–transitive closure of its covering relation (Definition 5.10), and of any *irreflexive* relation generating the same order, and the digraph of such a relation is acyclic. Irreflexivity is needed: on $V = \{v\}$ the loop $E = \{(v, v)\}$ has E^* the one-element poset while its digraph is a cycle of length one, so loops must be discarded from a generating relation before its graph is called a DAG.

Therefore, DAG-hood is not a graph-flavored curiosity but the *presentation of a partial order by generators*: the stored edges generate, and the semantic order is the closure. All of Section 5.3 applies as algorithmics: topological sort (Theorem 5.4) as build/task scheduling with cycle detection as the acyclicity audit; chains as critical paths and antichains as parallelism budgets (Example 5.3); transitive reduction as the minimal equivalent dependency declaration (deduplicating a manifest without changing the closure). One further applied identification merits its own sentence: **version histories are DAGs, not trees**. A merge commit has two parents, making the history a DAG that *presents* the ancestry partial order (not necessarily its Hasse

diagram, since a parent edge may be implied by a longer path and git does not remove such edges). A "common ancestor" query asks for lower bounds of two commits in that order. The *meet* (Definition 5.7), when it exists, is the unique greatest lower bound; git's merge base is the set of *maximal* common ancestors, and when there are several of them the order has no meet for that pair at all. That is the poset fact (maximal ≠ greatest) behind git's criss-cross merge cases, and its documentation says as much (Git Project 2026, `git-merge-base`).

Definition 15.4 (Tree, forest). A *tree* is a nonempty connected acyclic undirected graph (simple, as all undirected graphs here are); a *forest* is an undirected graph all of whose components are trees, so the empty graph is a forest and not a tree. For a nonempty finite simple undirected graph the following are equivalent: it is a tree; it is connected with $|E| = |V| - 1$; any two of its vertices are joined by exactly one path, in the strict sense of Definition 15.2 (the empty graph satisfies the last clause vacuously and fails the second, which is why nonemptiness is required) (Diestel 2017, Theorem 1.5.1 and Corollary 1.5.3; walks would be of no use in this characterization, since even the two-vertex tree has infinitely many walks between its endpoints, obtained by traversing its edge back and forth). A **rooted tree** is a tree with a distinguished vertex r, the *root*. Orienting every edge away from r gives the tree's *outward* digraph, in which every vertex is reachable from r by exactly one directed

walk, which is also its only directed path. The unique-directed-path condition alone does *not* define rooted trees: the digraph $r \to a \to b \to r$ has exactly one directed path from r to each vertex and is a cycle. Acyclicity of the underlying undirected graph is what excludes it.

Unique-path is the tree's essence and its applied warrant. Hierarchies (file systems, org charts, DOM and JSON documents, decision trees) are exactly the relations where *context is unambiguous*: each node has one parent, one path to the root, one address. In the outward digraph every vertex other than r has exactly one incoming edge, and the *parent* function $p\colon V \setminus \{r\} \to V$ that it defines is well-founded in Chapter 7's sense (following parents from any vertex reaches r after finitely many steps, since the tree has no cycle). Therefore, structural induction and recursion (Theorem 7.3, Theorem 7.2's pattern) are sound on trees with no further argument. This is the formal reason tree-walking programs are written recursively and verified by induction on depth, and the reason cyclic "trees" (Section 7.1's mutation-made cycles) break both the program and the argument at once. When hierarchy fails (a node needing two parents, a document shared by two folders), the honest structure is a DAG with the attendant costs (multiple addresses, meet ambiguity) or a graph with full generality. Much schema anguish is the attempt to store a DAG in a tree-shaped container, with duplicated nodes, symbolic links, and label systems being the standard

workarounds, each trading away a different piece of path uniqueness.

15.4 Degrees, counting, and graph magnitude estimates

The handshake lemma (Theorem 13.5(iv): $\sum_v \deg(v) = 2|E|$) and its directed sibling ($\sum_v \deg^+(v) = \sum_v \deg^-(v) = |E|$) are Chapter 13's incidence double-counting specialized to graphs, and they anchor magnitude reasoning about networks. Average degree $2|E|/|V|$ (for $|V| > 0$) prices traversal costs and memory layouts. A claimed edge list whose degree sums are odd, or whose in- and out-totals disagree, is *provably* corrupt before any semantic check (Section 13.4's aggregate audit applied to topology data). Sparse-versus-dense ($|E|$ near $|V|$ versus near $|V|^2$, the extremes of Example 3.1) selects between adjacency-list and matrix representations, which is Section 3.4's trade quantified. Beyond bookkeeping, extremal facts of the same double-counting genus do real design work. A DAG's edges are at most $\binom{|V|}{2}$; a tree's are exactly $|V| - 1$; so "how far from a tree" measures a dependency structure's entanglement. For an undirected graph with k connected components (we write k here because c is taken), the measure is the **cyclomatic number** $|E| - |V| + k$: the number of edges that must be removed to leave a forest, equivalently the dimension of the graph's cycle space (Diestel 2017, Section 1.9). It is not the

number of cycles, which is generally much larger: K_4 has cyclomatic number $6 - 4 + 1 = 3$ and seven cycles. Probabilistic facts (Chapter 14 on the product space of random edges) calibrate expectations, and two thresholds must be kept apart. In the Erdős–Rényi model $G(n, p)$, with each of the $\binom{n}{2}$ possible edges present independently with probability p, a *giant component* containing a constant fraction of the vertices emerges abruptly as the mean degree $(n-1)p$ crosses 1; below it all components are small, above it one is large, and many vertices may still lie outside it. Global *connectivity* (every vertex in one component) arrives much later, at mean degree about $\ln n$ (Erdős and Rényi 1960; Diestel 2017, Section 11.4). "A few extra links" therefore separates fragmented from *mostly* connected, not from fully connected. We state these landmarks without derivation; cascade and epidemic reasoning consults them constantly.

15.5 Hypergraphs: set systems in general

Definition 15.5 (Hypergraph). A *hypergraph* is a pair $(V, \mathcal{E})$ with $\mathcal{E} \subseteq \mathcal{P}(V)$: edges are arbitrary vertex *sets*, not pairs. When distinct edges may have the same vertex set, one takes instead an *indexed family* $(e_i)_{i \in I}$ of subsets of V (Definition 2.5), a hypergraph *with multiple edges* (Berge 1989, Chapter 1).

Hypergraphs drop the arity-2 restriction and thereby name the ubiquitous many-to-many shape: a tag system (V = items, each tag's extension an edge),

a chat platform's channels, a curriculum's courses-with-enrollments, a transaction touching several accounts. An n-ary database relation (Chapter 10) is also one, provided the vertices keep their attribute identity: the vertex set must be the set of (attribute, value) pairs, each row becoming the edge $\{(A, t(A)) : A \in H\}$, since untagged value sets would identify the rows $(A = 1, B = 2)$ and $(A = 2, B = 1)$. The **incidence relation** $I \subseteq V \times \mathcal{E}$ $(v \in e)$ recovers the bipartite-graph view. Every hypergraph is a bipartite graph with designated sides, and the converse holds exactly for the indexed version: a bipartite graph with sides V and I is the hypergraph $(e_i)_{i \in I}$ with e_i the neighborhood of i, and two right-side vertices with the same neighborhood give two edges with the same vertex set, which Definition 15.5's simple form would collapse into one. This is the adjacency isomorphism of Section 11.2, one level up. Standard hypergraph problems are recognizable as Part I constructions under stress, and each comes in a feasibility form and an optimization form, which differ sharply in difficulty. *Covering* asks for a set of edges whose union is V (monitoring probes covering all hosts), and the optimization problem is a cover with the fewest edges. *Packing* asks for pairwise-disjoint edges (non-conflicting transaction batches), optimized by taking as many as possible. A *transversal* is a vertex set meeting every edge (tap points hitting all data flows, assuming no edge is empty, since an empty edge can be hit by nothing). An *inclusion-minimal* transversal is easy to find: start

from all of V and delete any vertex whose removal still leaves every edge hit. For the edges $\{a, b\}$ and $\{a, c\}$, both $\{a\}$ and $\{b, c\}$ are inclusion-minimal. The hard problem is the *minimum-cardinality* transversal, or deciding whether one of size at most k exists. Then there is *coloring*, which comes in two strengths that applications must not confuse. *Weak* coloring partitions V so that no edge with at least two vertices is monochrome. *Strong* coloring gives the vertices of each edge pairwise distinct colors, which is the same as properly coloring the ordinary graph whose edges join every pair of vertices sharing a hyperedge (the *conflict graph*). Exam scheduling, where every course taken by one student needs a different time, is the strong problem: three courses colored red, red, blue satisfy the weak condition and still clash for that student. The optimization forms of all of these problems (fewest covering edges, most packed edges, smallest transversal, fewest colors) are NP-hard in general (Chapter 9's boundary, met in its combinatorial dress). Therefore, applied practice runs on approximation guarantees and structure exploitation, with the greedy cover's logarithmic ratio being the canonical positive result, and the reason "greedy, then audit the gap" is a defensible operational posture.

Note. Modeling grants degrees of freedom. The same domain can be cast as a graph (pairwise projections of the true n-ary structure) or a hypergraph (the structure itself), and the projection *loses*

information: Chapter 4's image asymmetry at schema scale, since knowing all pairwise co-enrollments does not determine the courses. Choosing the pairwise cast is often right (tooling, algorithms), but it is a lossy compression to be chosen knowingly, not a neutral notation change. The recurring data-modeling error it invites is answering an n-ary question (which *triples* co-occur?) from pairwise storage that cannot contain the answer.

15.6 Summary

Graphs are curated relations. Walks compose edges, paths are walks without repetition, reachability is reflexive–transitive closure computed as a least fixpoint, and distance is closure stratification. Connectivity relations are equivalences, components their quotient, condensation the quotient graph with intra-component edges discarded (and therefore acyclic), and union–find the mutable partition, with network partitions being the concept literally. DAG equals partial-order-by-generators, importing scheduling, critical paths, parallelism bounds, and transitive reduction from Chapter 5, with version histories presenting an order whose merge bases are maximal lower bounds rather than meets. Trees are the unique-path case where well-founded recursion is native, and forcing DAG-shaped reality into tree-shaped storage is a named modeling failure. Degree identities are incidence double-counting serving as data audit and magnitude

estimation, the cyclomatic number measures distance from a forest, and random graphs grow a giant component at mean degree 1 long before they become connected. Hypergraphs are set systems proper (bipartite incidence in disguise, with indexed edges where multiplicity matters) whose optimization problems (minimum cover, maximum packing, minimum transversal, and weak or strong coloring with fewest colors) are the NP-hard workhorses of allocation, and whose pairwise projections lose exactly the higher-arity information many questions need.

Chapter 16. Beyond Crisp Sets: Fuzzy, Rough, and Multisets

Dependencies: Chapters 1–5; Chapter 9 (the algebra being generalized); Chapters 10 and 14 for contrasts. Classical sets are crisp: *membership is total, binary, and exact. Applications meet three recurring pressures against each clause (graded membership, indiscernible granularity, and significant multiplicity), and each pressure produced a disciplined generalization. In this chapter, we present fuzzy sets, rough sets, and multisets as controlled relaxations of specific classical clauses, with the mathematics that survives each relaxation, the mathematics that does not, and the applied judgment of when each model is the honest one.*

16.1 The method of deliberate generalization

The pattern, three times over: identify which axiom of the classical concept the application strains; relax exactly that axiom; recompute which theorems survive. The recomputation is the intellectual content. A generalization is characterized as much by its *casualties* (the classical laws that fail) as by its gains, and the applied error this chapter warns against throughout is transporting a classical law into a generalized setting where it has quietly died. Chapter 10 already exhibited the pattern's canonical instance: SQL's bags relax "no multiplicity," and

Theorem 10.1's identities had to be re-audited one by one. Here the pattern gets its systematic treatment.

16.2 Fuzzy sets: graded membership

Definition 16.1 (Fuzzy set; Zadeh 1965). A *fuzzy set* on a universe U is a function $\mu: U \to [0,1]$, the **membership function**; $\mu(x)$ is the *degree* of membership. The classical (crisp) sets embed as the $\{0,1\}$-valued functions, that is, characteristic functions (Chapter 2), with the embedding making "fuzzy set" a literal generalization of $\mathcal{P}(U) \cong \{0,1\}^U$ to $[0,1]^U$.

Definition 16.2 (Zadeh operations). $(\mu \cap \nu)(x) = \min\big(\mu(x), \nu(x)\big)$; $(\mu \cup \nu)(x) = \max\big(\mu(x), \nu(x)\big)$; $(\mu^c)(x) = 1 - \mu(x)$; $\mu \subseteq \nu$ iff $\mu(x) \le \nu(x)$ for all x. The **α-cut** of μ is the crisp set $\mu_{\ge\alpha} = \{x: \mu(x) \ge \alpha\}$.

What survives: min/max/complement satisfy commutativity, associativity, distributivity, De Morgan, idempotence. The lattice structure stands ($[0,1]^U$ is a complete distributive lattice under pointwise order; Definition 5.8 at work). What dies: **excluded middle and non-contradiction**. For $\mu(x) = 0.5$: $(\mu \cup \mu^c)(x) = 0.5 \neq 1$ and $(\mu \cap \mu^c)(x) = 0.5 \neq 0$. A fuzzy set and its complement overlap, and their union does not exhaust the universe. This is not a defect but the *point*: the model is built for predicates ("tall," "hot," "anomalous," "similar") whose boundary cases genuinely half-satisfy both a

description and its negation. However, the casualty list invalidates every classical inference that leans on partition-by-complement. Case analysis "either in A or in A^c" no longer covers with certainty weight, and Chapter 9's Boolean-identity toolkit must be re-checked identity by identity (the surviving structure is a *De Morgan algebra*, not a Boolean one).

Alternative connectives and the modeling license. Min/max is one choice. Product ($\mu\nu$) and Łukasiewicz ($\max(0, \mu + \nu - 1)$) conjunctions define rival systems (*t-norms*), each with matching duals, and the choice is consequential. Min is idempotent but insensitive to reinforcement (two weak pieces of evidence min to the weaker). Product compounds (independent-evidence intuition, Chapter 14 adjacent). Łukasiewicz saturates. Fuzzy control systems (the technology's industrial success story: appliance controllers, gearbox logic) and fuzzy retrieval scoring choose t-norms as engineering decisions, documented per system. The epistemic obligation this freedom creates: a fuzzy model must *state* its connectives, since "fuzzy AND" underdetermines the mathematics. This is a sharper form of Chapter 9's lesson that the algebra is fixed only once the operations are.

Fuzzy is not probability. $\mu_{\text{tall}}(\text{Kim}) = 0.7$ asserts partial satisfaction of a vague predicate by a fully known object. $\Pr[\text{tall}] = 0.7$ asserts uncertainty about a crisp fact. The distinction is operational, not philosophical hair-splitting. Degrees of vague

satisfaction need not obey additivity (Kim can be 0.7-tall and 0.6-heavy with no coherence constraint), conditioning has no canonical fuzzy analogue, and conflating the two mis-prices aggregations (averaging membership degrees across a population is meaningful; "the probability a random person is tall" requires the measure apparatus of Chapter 14 and answers a different question). Where both vagueness and uncertainty are present ("probably quite anomalous"), hybrid formalisms exist, but the applied baseline is to name which of the two each number encodes.

Example 16.1 (Thresholding as α-cut, and its instability). An anomaly scorer emits μ:events → [0,1]. Every alerting rule "page when score $\geq \alpha$" is an α-cut, and the cut family is nested: $\alpha \leq \beta \Rightarrow \mu_{\geq\beta} \subseteq \mu_{\geq\alpha}$, a chain in $(\mathcal{P}(U), \subseteq)$, Definition 5.3's granularity dial made continuous. The engineering fact the formalism surfaces: near-threshold mass makes cut membership unstable under score noise. Under a uniform upward shift of exactly ε, the newly included events are exactly those in the band $[\alpha - \varepsilon, \alpha)$, and nothing leaves. Under general noise of magnitude at most ε ($|\delta(x)| \leq \varepsilon$ per event), an event can change membership only if its score lies in the two-sided band $[\alpha - \varepsilon, \alpha + \varepsilon)$, since scores above it stay above α and scores below it stay below. Not every event in that band changes, so the band's mass is an upper bound on the churn, not the churn itself. Either way, alert-flapping is a *statement about the measure of a band*, measurable and monitorable,

rather than a mystery. Hysteresis (enter at α, exit at $\alpha' < \alpha$) is the standard repair, and in this vocabulary it is precisely the replacement of one cut by a pair of nested cuts with state.

16.3 Rough sets: indiscernibility and approximation

Fuzzy sets grade the *predicate*. Rough sets keep predicates crisp and admit that the *observer's resolution* is limited.

Definition 16.3 (Rough approximation; Pawlak 1991). Let $\sim$ be an equivalence on U (the **indiscernibility relation**: objects identical in all observed attributes, Chapter 5's kernel of the observation function, Definition 5.5). For a target $X \subseteq U$, the **lower approximation** $\underline{X} = \{x : [x] \subseteq X\}$ (the union of blocks wholly inside X) and the **upper approximation** $\overline{X} = \{x : [x] \cap X \neq \emptyset\}$ (the union of blocks meeting X). X is *definable* at this granularity iff $\underline{X} = \overline{X}$; otherwise X is **rough**, with *boundary* $\overline{X} \setminus \underline{X}$.

Always $\underline{X} \subseteq X \subseteq \overline{X}$. The lower approximation collects what the granularity *certainly* includes, the upper what it *possibly* includes, and the boundary is the region where the available attributes simply cannot decide, not for lack of data volume but for lack of *distinguishing* attributes. The formalism thus prices a question data practice usually leaves

implicit: *is the feature set sufficient to express the target concept?* Perfect classification of X from observed attributes is possible only if X is definable ($\underline{X} = \overline{X}$). A nonempty boundary demonstrates, prior to any model fitting, any choice of classifier, that errors on boundary blocks are unavoidable at this granularity, and their empirical misclassification floor is computable from block counts (Example 16.2 does the computation). Feature engineering, in this vocabulary, is refinement of $\sim$ (Definition 5.3) to shrink the boundary. Attribute *reduction* (finding minimal attribute subsets inducing the same $\sim$: "reducts") is the dual economy. The theory's applied home is there: feature selection, rule induction from decision tables, and audit questions of the form "could these records ever have been distinguished by the fields we collect?" In privacy contexts, the same question inverts into k-anonymity's demand that blocks be *large* (indistinguishability as protection, the same quotient mathematics with the opposite sign).

What survives and what dies: lower approximation distributes over intersection and upper over union ($\underline{X \cap Y} = \underline{X} \cap \underline{Y}, \overline{X \cup Y} = \overline{X} \cup \overline{Y}$), but only inclusions hold in the mixed cases (Pawlak 1991, Chapter 2). The operators are an interior/closure pair on the partition topology, and Chapter 4's image and preimage express them exactly, though not symmetrically. With $\pi: U \to U/\sim$ the projection, the upper approximation is the preimage of an image,

$$\overline{X} = \pi^{-1}\big[\pi[X]\big],$$

the union of every block that meets X. The lower approximation is *not* the same construction applied to X (for the single block $U = \{a, b\}$ and $X = \{a\}$, the preimage of the image is all of U while $\underline{X} = \emptyset$). It is obtained by passing through the complement,

$$\underline{X} = U \setminus \pi^{-1}\big[\pi[U \setminus X]\big],$$

everything not in a block that meets the complement, which is exactly the union of the blocks inside X. Reading the second display as an identity between operators gives the De Morgan duality

$$\big(\underline{X}\big)^{c} = \overline{X^{c}},$$

and dually $\big(\overline{X}\big)^{c} = \underline{X^{c}}$. Complement behaves; excluded middle holds per element, but *decidability* does not. The tripartition {certainly in, certainly out, boundary} is the structure: three-valued where Chapter 10's nulls were, but with the third value now carrying exact semantics (granularity failure) rather than SQL's ambiguous "unknown."

Example 16.2 (The confusion matrix's ceiling). Consider a fraud target X over a fully labeled table of N transactions, observed through attributes (amount band, merchant class, hour). The blocks C are attribute-combination cells (Chapter 13's product cells), $\underline{X}$ = cells with only fraud, $\overline{X}$ = cells with any fraud. Any classifier that sees only these

attributes is constant on each cell, so on a mixed cell it must misclassify either the fraudulent or the legitimate transactions, whichever it labels against. Its empirical error on this table is therefore at least

$$\frac{1}{N}\sum_{C} \min\left(|C \cap X|,\ |C \setminus X|\right),$$

the *minority mass* of the mixed cells, and the bound is attained by labeling each cell with its majority. The boundary mass $\left|\overline{X} \setminus \underline{X}\right|/N$ is a different and larger number. For example, among 1,000 transactions, suppose that all cells are pure except one containing 27 legitimate and 3 fraudulent transactions. The boundary mass is $30/1{,}000 = 3\%$, but the error floor is $3/1{,}000 = 0.3\%$: label the cell legitimate and only the three frauds are missed. The floor is an audit-grade figure produced by two `GROUP BY` queries (Chapter 10 computing Chapter 5's quotient), and its publication alongside model metrics separates "model underperforms" from "features underdetermine": two failures with different owners and different fixes. Two calibrations apply. The floor is *empirical*, a fact about this fixed table; a bound on future or population error requires distributional assumptions (Chapter 14) and is not supplied by the counts alone. And it is a floor for classifiers on *these* attributes; refining $\sim$ with a new attribute can split the mixed cell and lower it, which is what feature engineering is for.

16.4 Multisets: multiplicity restored

Definition 16.4 (Multiset). A *multiset* (bag) over U is a function $m: U \to \mathbb{N}$, the **multiplicity function**, with *support* $\{x: m(x) > 0\}$ (required finite for this chapter). Operations, pointwise: union max, intersection min, **sum** $m + n$ (additive union), difference $\max(0, m - n)$; inclusion $m \subseteq n$ iff $m(x) \leq n(x)$ everywhere; cardinality $|m| = \sum_x m\,(x)$.

Here is the three-member family portrait: crisp sets are {0,1}-valued, fuzzy sets [0,1]-valued, multisets $\mathbb{N}$-valued. Three codomains for one functional form, with the operation tables determined by the codomain's own lattice/arithmetic structure (a unification worth a margin note: min/max serve as intersection/union in all three, because all three codomains are chains). Multisets are the honest model wherever occurrences, not identities, carry the meaning: word counts (the "bag of words" of text retrieval; the name is the mathematics), inventory and account ledgers, resource allocations, prime factorizations ($60 = \{2,2,3,5\}$ as a multiset; the fundamental theorem of arithmetic is a statement about unique multiset decomposition), chemical equations, and SQL's intermediate results (Chapter 10's bags, now with their algebra stated rather than lamented).

The distinctive structural novelty is the *pair* of unions: max (join-like, idempotent) versus +

(aggregating, and not idempotent in general: $m + m = m$ only for the empty multiset), with no classical counterpart for the latter. Both are legitimate. They answer different questions ("combined catalog" versus "combined inventory"), and the bag-semantics bugs of Section 10.4 are, in this light, systematically *confusions of the two unions*. `UNION ALL` is $+$, `UNION` is max after support-collapse, and the double-counting join multiplies multiplicities where the query's intent was support. What dies with idempotence: the algebra is no longer a lattice under $+$; inclusion–exclusion needs restatement ($|m + n| = |m| + |n|$ holds unconditionally; the sum rule *simplifies*, disjointness no longer needed, which is why accountants prefer ledgers to sets); and extensionality refines. Two multisets are equal iff their multiplicity functions are, so "same support" is now a strictly coarser equivalence (Definition 5.3) whose quotient map ($m \mapsto$ support) is exactly `DISTINCT`: a projection that Chapter 5's well-definedness discipline says must be *placed*, not sprinkled.

Example 16.3 (Ledgers versus registries). A user-role *registry* is a set (holding a role twice is meaningless). A *ledger* of grants and revocations is a *sequence* of events (Definition 4.7), and three objects derived from it must be kept distinct. First, the cumulative counts: through time t, $G_t \colon U \to \mathbb{N}$ counts the grants of each role and R_t the revocations, two multisets. Then, the invariant: under reference-counting semantics, where a role is active while it

has been granted more often than revoked, the running difference must stay non-negative, $R_t \subseteq G_t$ as multiset inclusion at every prefix t, per-element counts and all. Finally, the registry, which is the *support of the net count*,

$$Registry_t = \mathrm{supp}(G_t - R_t),$$

and not the support of the ledger itself: one grant followed by one revocation leaves the role in the support of G_t and absent from the registry. If the intended policy is idempotent instead (a grant makes the role active, a revocation makes it inactive, however many times either is repeated), then the registry is defined by that transition rule applied along the sequence, and the multiset counts are the wrong intermediate object. Systems that store the registry but audit via the ledger are keeping both the quotient and the fiber data (Theorem 5.2's factorization as an architecture: the ledger, its projection to the registry, and the reconciliation equation between them). "Drift" incidents are failures of that equation, detected, once the *correct* equation is stated, by one comparison query.

16.5 Choosing the model: a closing discipline

The chapter's three generalizations, plus the classical base and Chapter 14's measures, form a decision surface the practitioner crosses on every schema and metric design. The closing discipline is to ask, per concept being modeled: *Is membership*

binary? (No: fuzzy, or a probability if the number encodes uncertainty about a crisp fact, Section 16.2's distinction.) *Is it observable at the working granularity?* (No: rough; report the boundary rather than paper over it.) *Do repetitions carry meaning?* (Yes: multiset, and decide which union each operation intends.) *Does order carry meaning?* (Yes: sequences, which require a different set-theoretic structure, the functions on an index set that Definition 4.7 already provides.) The questions are cheap. The defaults ("everything is a crisp set," or worse, "everything is whatever the ORM emits") are silently wrong on real data at a measurable rate, and every mismatch identified by the checklist has appeared in this book as a named bug family: threshold flapping, feature-floor denial, double counting, order dependence. Modeling is choosing the codomain. The rest (this book's recurring moral) is checking which laws survived the choice before reasoning with them.

16.6 Summary

Deliberate generalization relaxes one classical clause at a time and re-audits the laws. Fuzzy sets grade membership into [0,1]: lattice and De Morgan structure survive, excluded middle dies, t-norm choice is a stated modeling decision, α-cuts bridge back to crisp sets, and vagueness must not be booked as probability. Rough sets keep the target crisp but observe it through an indiscernibility quotient: the upper approximation is a preimage of

an image, the lower one passes through the complement, the two bracket every concept, the mixed cells' minority mass is an empirical error floor computable before any model, and refinement/reduction of the partition is the formal content of feature engineering, with k-anonymity the same mathematics run for privacy. Multisets restore multiplicity: two unions with different laws, simplified sum-rule arithmetic, support-projection as the bridge to sets, and the bag-set confusions of practice resolved by placing the projection deliberately, with a ledger's registry being the support of its net counts. Model choice is codomain choice plus law re-audit.

Appendix A. Notation Reference

Logic. $\wedge$ and; $\vee$ or; $\neg$ not; $\Rightarrow$ implies; $\Leftrightarrow$ iff; $\forall$ for all; $\exists$ there exists.

Sets and membership. $x \in A$ membership; $\emptyset$ empty set; $\{a, b\}$ enumeration; $\{x \in A : \varphi(x)\}$ separation (bounded set-builder); $A \subseteq B$ subset; $A \subsetneq B$ proper subset; $\mathcal{U}$ universe of discourse; $\mathcal{P}(A)$ power set; $|A|$ cardinality.

Operations. $A \cup B$, $A \cap B$ union, intersection; $A \setminus B$ difference; $A \triangle B$ symmetric difference; A^c complement (relative to the universe in scope); $\bigcup_{i \in I} A_i$, $\bigcap_{i \in I} A_i$ indexed union, intersection; $A \times B$ Cartesian product; (a, b) ordered pair; $A \sqcup B$ disjoint union; A^n n-fold product.

Relations. $R \subseteq A \times B$ binary relation; $a \mathrel{R} b$ for $(a, b) \in R$; $\mathrm{dom} R$, $\mathrm{ran} R$ domain, range; R^{-1} converse; $S \circ R$ composition (first R, then S); id_A identity; R^+, R^* transitive and reflexive–transitive closure; $[a]$ equivalence class; $A/\sim$ quotient set; $\ker f$ kernel of f.

Functions. $f : A \to B$ total function; $f : A \rightharpoonup B$ partial function; $f(a)$ application; $f[X]$ image; $f^{-1}[Y]$ preimage; $g \circ f$ composition; B^A function space; χ_X characteristic function; $f \restriction X$ restriction; $f \oplus g$ functional override; $\prod_{i \in I} A_i$ general product (choice functions).

Orders. $\leq$ partial order; $<$ strict order; $\vee$, $\wedge$ join, meet; $\bowtie$ natural join (Chapter 10).

Numbers and cardinals. $\mathbb{N}$ (with 0), $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$; $[n] = \{1, \ldots, n\}$; $[a, b]$, (a, b) intervals; $A \approx B$ equinumerous; $A \preccurlyeq B$ injects into; $\aleph_0$ cardinality of $\mathbb{N}$; $\mathfrak{c} = 2^{\aleph_0}$ the continuum; $n^{\underline{k}}$ falling factorial; $\binom{n}{k}$ binomial coefficient.

Ordinals and hierarchy. $S(x) = x \cup \{x\}$ successor; ω least infinite ordinal; α, β, γ ordinals; V_α cumulative hierarchy stage; Ord the class of ordinals.

Probability (Chapter 14). $(\Omega, \mathcal{F}, \Pr)$ probability space; $\Pr[E \mid F]$ conditional probability; $\mathbb{E}[X]$ expectation; $\mathbf{1}_E$ indicator.

Generalized sets (Chapter 16). $\mu: U \to [0,1]$ fuzzy membership; $\mu_{\geq\alpha}$ α-cut; $\underline{X}, \overline{X}$ lower and upper rough approximations; $m: U \to \mathbb{N}$ multiset multiplicity.

Bibliography

We organize the literature below by part, and we cite it in the text by author and year, with a chapter, section, or theorem locator wherever a stated result is demonstrated there. The editions cited are the ones we consulted for this book's statements, but any modern edition serves for the books.

Foundations (Parts I–II).

Halmos, P. R. *Naive Set Theory*. Springer, 1974. The classic short course in exactly the informal-rigorous register of Parts I–II.

Enderton, H. B. *Elements of Set Theory*. Academic Press, 1977. Careful constructions of pairs, functions, and numbers; Chapter 4 (natural numbers and the recursion theorem), Chapter 6 (Cantor and Cantor–Schröder–Bernstein), Chapter 7 (ordinals).

Jech, T. *Set Theory*, 3rd millennium edition. Springer, 2003. The comprehensive reference: Chapter 2 (ordinals), Chapter 5 (the Axiom of Choice and its equivalents), Chapter 6 (the cumulative hierarchy and rank), Chapters 13–14 (the constructible universe, forcing, and the independence results).

Kunen, K. *Set Theory: An Introduction to Independence Proofs*. North-Holland, 1980. The standard route into Gödel's *L* (Chapter VI) and Cohen forcing (Chapter VII).

Boolos, G. S., Burgess, J. P., and Jeffrey, R. C. *Computability and Logic*, 5th ed. Cambridge, 2007. Nonstandard models of first-order arithmetic, for Section 7.2's distinction.

Kirby, L., and Paris, J. "Accessible Independence Results for Peano Arithmetic." *Bulletin of the London Mathematical Society* 14(4), 285–293, 1982. DOI 10.1112/blms/14.4.285. Goodstein sequences and hydra games, Section 7.5.

Tarski, A. "A Lattice-Theoretical Fixpoint Theorem and Its Applications." *Pacific Journal of Mathematics* 5(2), 285–309, 1955. The fixpoint theorem behind Section 7.5's iteration.

Dilworth, R. P. "A Decomposition Theorem for Partially Ordered Sets." *Annals of Mathematics* 51(1), 161–166, 1950. DOI 10.2307/1969503. Example 5.3's antichain bound.

Devlin, K. *The Joy of Sets*, 2nd ed. Springer, 1993. Includes anti-foundation and non-well-founded alternatives.

Aczel, P. *Non-Well-Founded Sets*. CSLI, 1988. The AFA monograph behind Section 8.2's remark.

Logic and Boolean structure (Chapter 9).

Givant, S., and Halmos, P. *Introduction to Boolean Algebras*. Springer, 2009. Chapter 2 (the derived laws), Chapter 10 (regular open sets), Chapter 15 (finite Boolean algebras), Chapter 22 (the representation theorem), Chapter 34 (Boolean algebras and Boolean spaces).

Stone, M. H. "The Theory of Representations for Boolean Algebras." *Transactions of the American Mathematical Society* 40(1), 37–111, 1936. DOI 10.2307/1989664.

Huth, M., and Ryan, M. *Logic in Computer Science*, 2nd ed. Cambridge, 2004. Section 1.5 (normal forms and functional completeness), Section 2.5 (undecidability of predicate logic).

Cook, S. A. "The Complexity of Theorem-Proving Procedures." *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 151–158, 1971. DOI 10.1145/800157.805047. NP-completeness of SAT.

Levin, L. A. "Universal Sequential Search Problems." *Problems of Information Transmission* 9(3), 265–266, 1973. The independent half of Cook–Levin.

Impagliazzo, R., Paturi, R., and Zane, F. “Which Problems Have Strongly Exponential Complexity?” *Journal of Computer and System Sciences* 63(4), 512–530, 2001. DOI 10.1006/jcss.2001.1774. The exponential time hypothesis, Section 9.3.

Bryant, R. E. “Graph-Based Algorithms for Boolean Function Manipulation.” *IEEE Transactions on Computers* C-35(8), 677–691, 1986. DOI 10.1109/TC.1986.1676819. Reduced ordered BDDs and their canonicity (Section 2), the integer-multiplication lower bound (Section 3.3 and the appendix).

Biere, A., et al., eds. *Handbook of Satisfiability*, 2nd ed. IOS Press, 2021. SAT solving as industrial practice.

Databases (Chapter 10).

Codd, E. F. “A Relational Model of Data for Large Shared Data Banks.” *Communications of the ACM* 13(6), 377–387, 1970. DOI 10.1145/362384.362685. The founding paper; still readable, still sharp.

Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases*. Addison-Wesley, 1995. The relational theory reference: Chapter 8 (functional dependencies and Armstrong’s axioms), Chapter 11 (normal forms).

Date, C. J. *Database in Depth*. O'Reilly, 2005. The set-versus-bag and null critiques from inside the model's camp.

PostgreSQL Global Development Group. *PostgreSQL Documentation*, section "Subquery Expressions." https://www.postgresql.org/docs/current/functions-subquery.html, consulted 2026. The `NOT IN` rule of Section 10.4.

Programming and specification (Chapters 11–12).

Pierce, B. C. *Types and Programming Languages*. MIT Press, 2002. The types-as-sets reading, its formal refinements, and its limits; Chapter 15 (subtyping).

Okasaki, C. *Purely Functional Data Structures*. Cambridge, 1998. Immutability engineering for Section 11.1's value-semantics argument.

Oracle. *Java Platform, Standard Edition 25 API Specification*, interface `java.util.Set`. https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Set.html, 2025. The mutable-element and `equals`/`hashCode` contract of Section 11.1.

Python Software Foundation. *The Python Language Reference*, section "Data model," `object.__hash__`. https://docs.python.org/3/reference/datamodel.html, consulted 2026. The hash/equality contract of Section 11.1.

Spivey, J. M. *The Z Notation: A Reference Manual*, 2nd ed. Prentice Hall, 1992. Chapter 5 (sequential systems and data refinement).

Abrial, J.-R. *The B-Book: Assigning Programs to Meanings*. Cambridge, 1996. Refinement (Chapter 11) discharged at industrial scale.

Abadi, M., and Lamport, L. "The Existence of Refinement Mappings." *Theoretical Computer Science* 82(2), 253–284, 1991. DOI 10.1016/0304-3975(91)90224-P. Refinement as trace inclusion, Section 12.3.

Jackson, D. *Software Abstractions: Logic, Language, and Analysis*, revised ed. MIT Press, 2012. Alloy's relational core (Chapters 4–5) and the small scope hypothesis.

Alloy Tools. *Alloy 6 Documentation*. https://alloytools.org/alloy6.html, 2021. Mutable state, temporal logic, and unbounded-time checking, Section 12.4.

Lamport, L. *Specifying Systems*. Addison-Wesley, 2002. TLA$^+$; behaviors as trace sets; Section 5.8 (proving implementation by refinement mappings), Section 8.9.4 (refinement with liveness).

Combinatorics and probability (Chapters 13–14).

Graham, R., Knuth, D., and Patashnik, O. *Concrete Mathematics*, 2nd ed. Addison-Wesley, 1994. Chapter 9 (asymptotics, including Stirling's approximation).

Stanley, R. P. *Enumerative Combinatorics*, vol. 1, 2nd ed. Cambridge, 2011. Where the counting rules become a discipline; Section 1.2 (sets and multisets).

Feller, W. *An Introduction to Probability Theory and Its Applications*, vol. 1, 3rd ed. Wiley, 1968. Discrete probability with combinatorial soul; Section II.3 (the birthday problem), Chapter IV (inclusion–exclusion, Bonferroni inequalities, matches).

Davis, K. R., Peabody, B., and Leach, P. "Universally Unique IDentifiers (UUIDs)." RFC 9562, IETF, May 2024. DOI 10.17487/RFC9562. Section 5.4 (version 4, its 122 random bits).

Billingsley, P. *Probability and Measure*, 3rd ed. Wiley, 1995. The measure-theoretic apparatus of Chapter 14 in full; Section 2 (probability measures, Borel sets), Section 3 (a nonmeasurable set).

Mitzenmacher, M., and Upfal, E. *Probability and Computing*, 2nd ed. Cambridge, 2017. Balls-in-bins (Section 5.2, maximum load), Bloom filters, and the randomized-systems toolkit.

Graphs and generalized sets (Chapters 15–16).

Diestel, R. *Graph Theory*, 5th ed. Springer, 2017. Chapter 1 (walks, paths, trees, the cycle space), Section 11.4 (the evolution of random graphs).

Erdős, P., and Rényi, A. "On the Evolution of Random Graphs." *Publications of the Mathematical Institute of the Hungarian Academy of Sciences* 5, 17–61, 1960. The giant component and connectivity thresholds, Section 15.4.

Git Project. *git-merge-base* documentation. https://git-scm.com/docs/git-merge-base, consulted 2026. Merge bases as best common ancestors, Section 15.3.

Berge, C. *Hypergraphs: Combinatorics of Finite Sets*. North-Holland, 1989. Chapter 1 (hypergraphs, including multiple edges).

Zadeh, L. A. "Fuzzy Sets." *Information and Control* 8(3), 338–353, 1965. DOI 10.1016/S0019-9958(65)90241-X. The founding paper of Section 16.2.

Klir, G., and Yuan, B. *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Prentice Hall, 1995.

Pawlak, Z. *Rough Sets: Theoretical Aspects of Reasoning about Data*. Kluwer, 1991. The founding monograph of Section 16.3; Chapter 2 (approximations of sets).

Wider horizons.

Ferreirós, J. *Labyrinth of Thought: A History of Set Theory*, 2nd ed. Birkhäuser, 2007. How the subject actually developed.

Maddy, P. *Defending the Axioms*. Oxford, 2011. What axiom choice is, philosophically, for the reader Chapter 8 provoked.

Index

C

D

E

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z