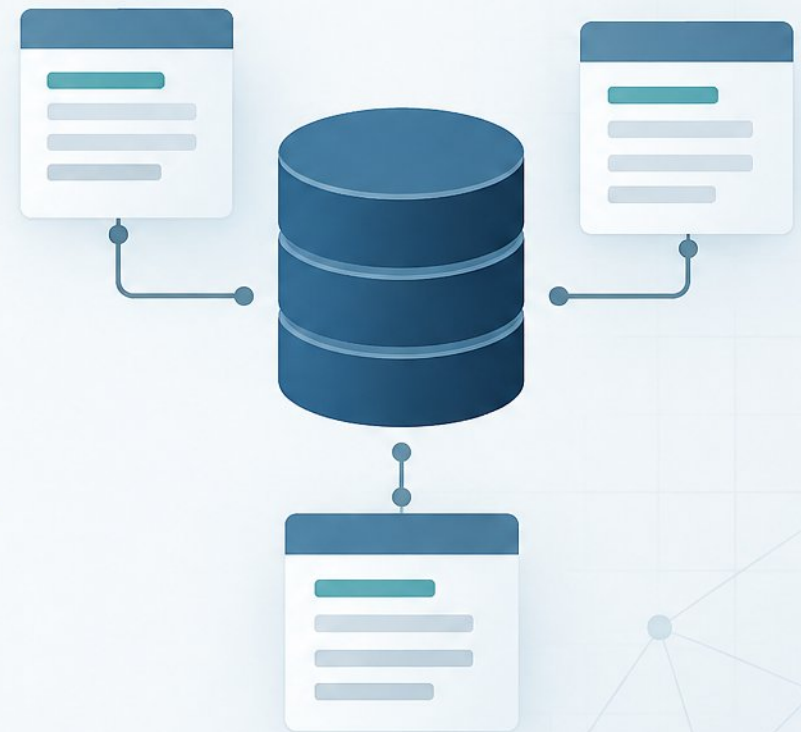


Introduction to SQL Database Design Practices

- **Schema quality** drives correctness, query speed, and long-term maintainability
- **Audience:** developers, junior designers, data engineers, analysts, students
- **Roadmap:** requirements, ER modeling, normalization, keys, indexing, migrations
- **Core vocabulary:** schema, relation, tuple, attribute, functional dependency
- **Workflow:** requirements → conceptual → logical → physical → validation → evolution
- **2026 default:** normalize first, enforce integrity in the database, plan migrations early



Course Roadmap and Design Lifecycle

FROM REQUIREMENTS TO A RELIABLE, REAL-WORLD DATABASE



Four-stage lifecycle: requirements, conceptual, logical, physical — with feedback loops



Skipping a stage compounds fragility: design debt grows with every release



Abstraction narrows from conceptual ER models to engine-specific physical choices



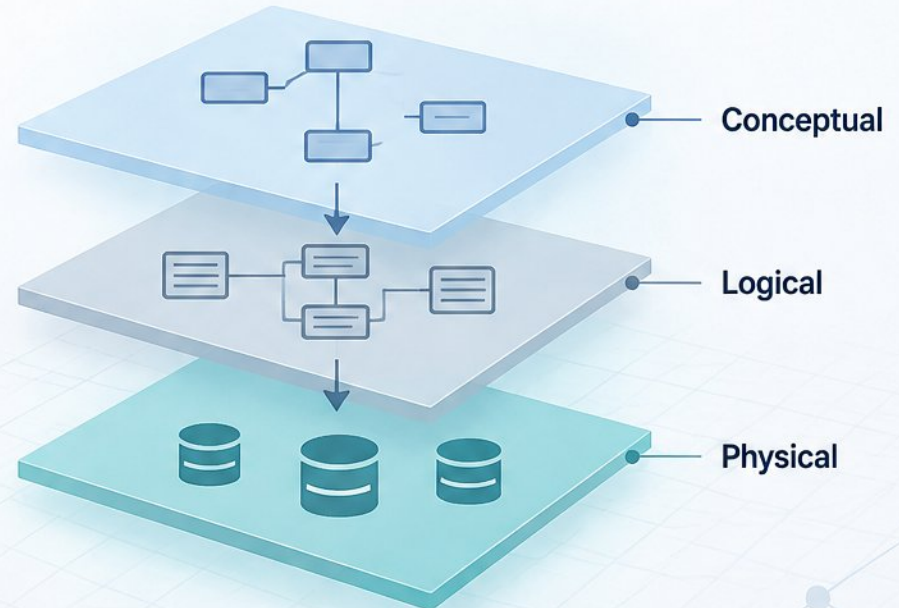
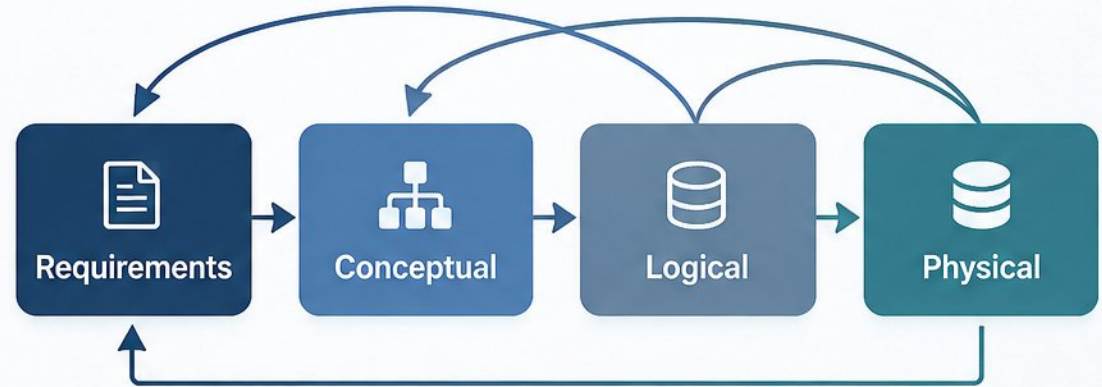
Roles: stakeholders, data architects, developers, and DBAs each own a stage



Model technology-agnostic first; physical decisions last, driven by the target engine

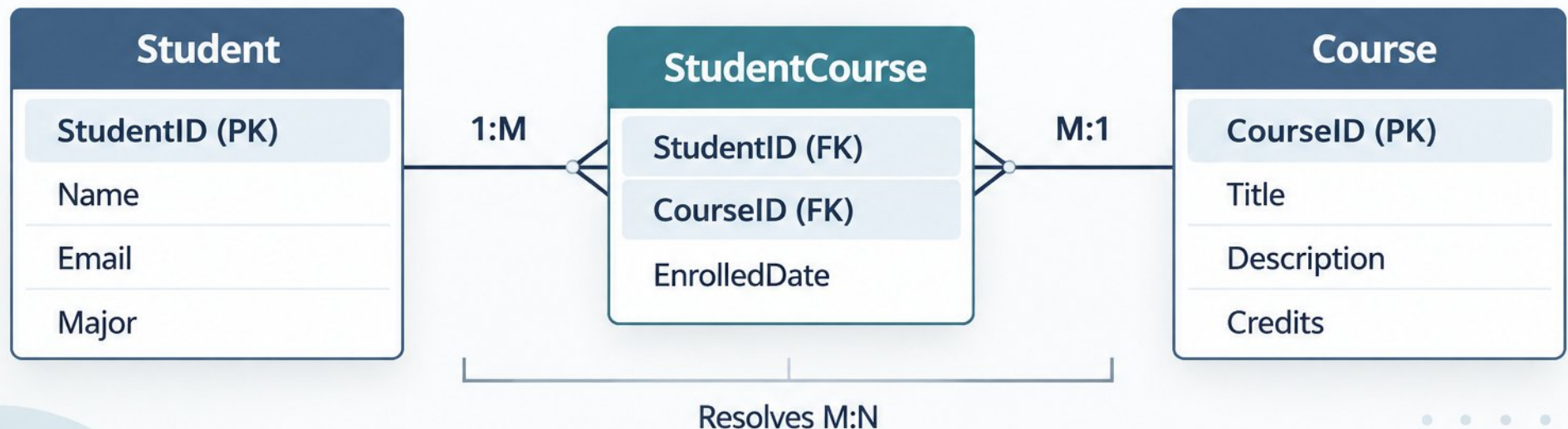


Validate with representative data and real queries before deployment — non-negotiable



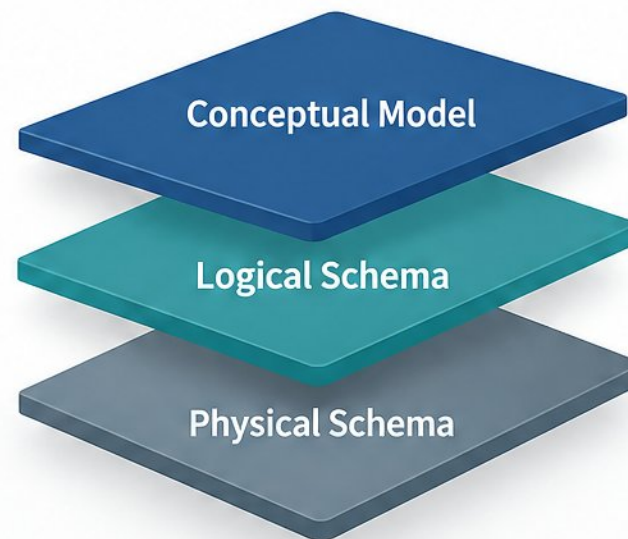
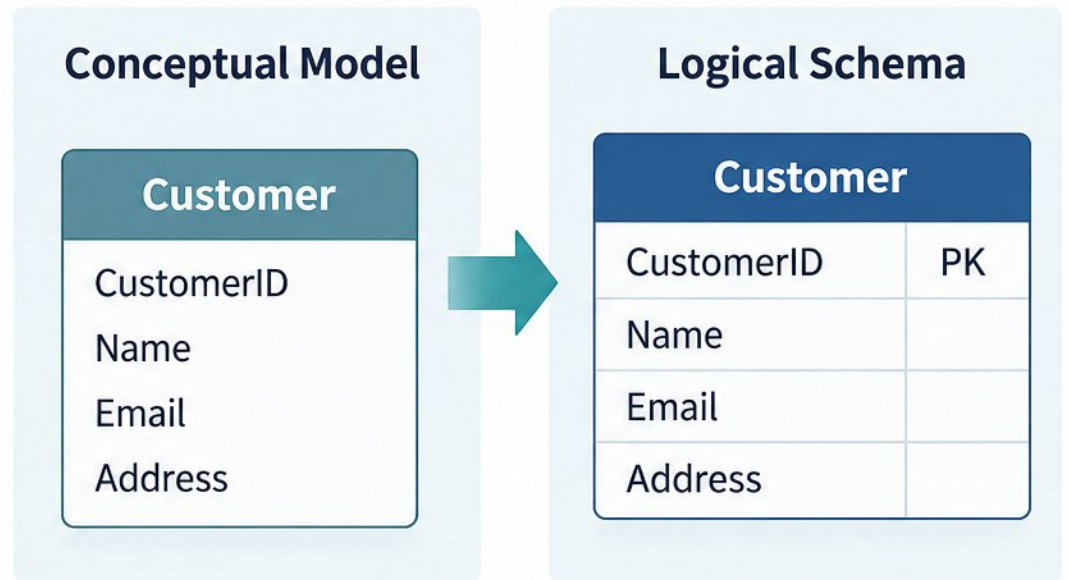
Requirements Analysis and Conceptual Modeling

- Translate business requirements into entities, relationships, attributes, and rules
- Build an ERD before any DDL — whiteboard fixes are cheap, migrations are not
- Model cardinality: 1:1, 1:M, M:N — plus participation and optionality constraints
- Decide what is an entity, an attribute, or a relationship
- Resolve M:N with junction tables, never list or array columns
- Validate the model with stakeholders and sample business questions



From Conceptual Model to Logical Schema

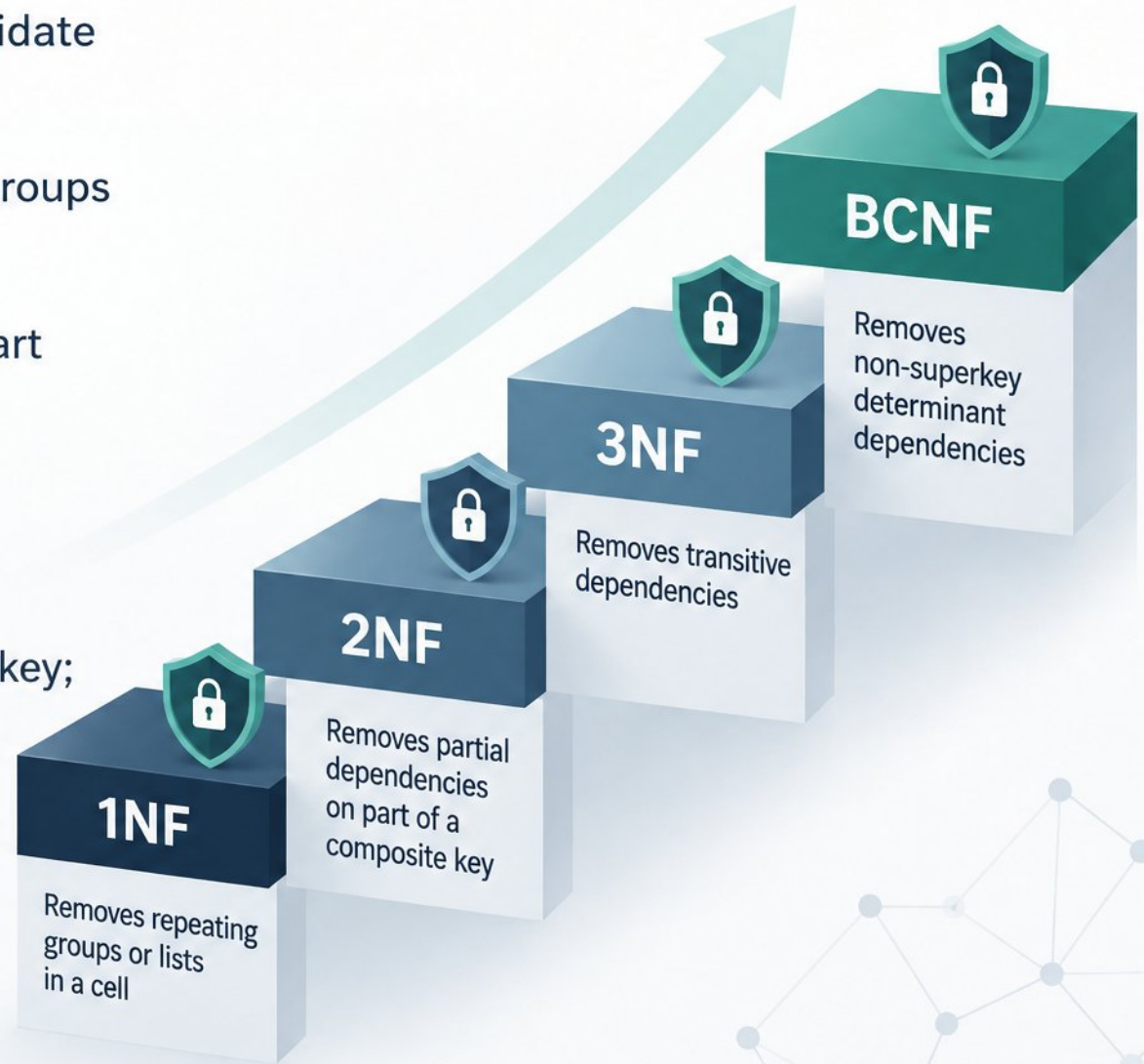
- Convert each entity into a table; each attribute becomes a column
- Assign primary keys and migrate foreign keys across relationships
- Keep the logical model technology-agnostic; no DBMS specifics yet
- Resolve many-to-many links with associative tables, not list columns
- Model subtypes and optional relationships explicitly, not nullable catch-alls
- Follow disciplined steps to prevent semantic drift from the conceptual model



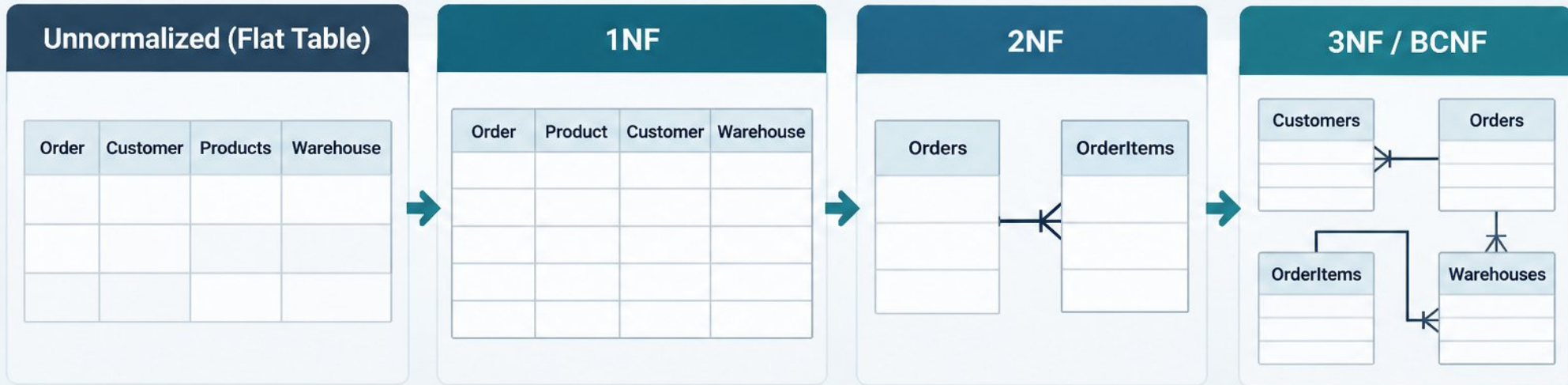
A disciplined path from concept to implementation

Normalization Fundamentals: 1NF through BCNF

- Functional dependencies and candidate keys define every normal form
- **1NF**: atomic values, no repeating groups or lists in a cell
- **2NF**: no partial dependencies on part of a composite key
- **3NF**: no transitive dependencies; the practical OLTP target
- **BCNF**: every determinant is a superkey; catches overlapping candidate keys
- Each form removes one class of update, insert, or delete anomaly



Practical Normalization: Worked Example and Denormalization Trade-offs



- One flat order table: lists in cells, partial and transitive dependencies
- 1NF: atomic values, no repeating groups — split product lists into rows
- 2NF: remove partial dependencies on part of the composite key
- 3NF: extract customers and warehouses — no transitive dependencies
- BCNF: split when a non-key column determines another column
- Normal forms check structure, not design: missing tables still pass
- Denormalize only for measured reads, one column at a time, with sync and monitoring
- Caching layers and materialized views often remove the need entirely



Keys, Constraints, and Data Integrity

A stronger data model leads to safer, more reliable applications.

- 1 BIGSERIAL identity default; UUID v7 when IDs span databases
- 2 Surrogate keys over natural; never expose SSN or email as PK
- 3 Choose referential actions deliberately: CASCADE, RESTRICT, SET NULL
- 4 NOT NULL, UNIQUE, CHECK, DEFAULT: the database as gatekeeper
- 5 Enforce rules in the database—frontends change, scripts bypass logic
- 6 Soft deletes with deleted_at plus partial unique indexes; audit columns

users

id	BIGSERIAL PRIMARY KEY
org_id	UUID NOT NULL
email	TEXT NOT NULL UNIQUE
name	TEXT NOT NULL
status	TEXT NOT NULL CHECK (status IN (...))
created_at	TIMESTAMPTZ NOT NULL DEFAULT ...
deleted_at	TIMESTAMPTZ

FOREIGN KEY (org_id) REFERENCES orgs(id)
ON DELETE CASCADE



**THE DATABASE
AS GATEKEEPER**

Physical Design: Data Types and Storage Decisions

RIGHT TYPES. FEWER PROBLEMS. A STRONGER FOUNDATION.



Never use FLOAT/DOUBLE for money — use NUMERIC(19,4) or integer cents



BIGSERIAL is 8 bytes vs UUID's 16; benchmarks show 20–40x faster reads



Use TIMESTAMPTZ, not TIMESTAMP; always add created_at and updated_at



VARCHAR(254) for email with a CHECK constraint; TEXT when length isn't a business rule



JSONB only for genuinely semi-structured data, with CHECK constraints and documented scope



Prefer BIGINT over INT for any growing identifier — the 4-byte saving isn't worth a migration

Key Comparisons

Choose the right type for the job.



FLOAT/DOUBLE
(not for money)

VS



NUMERIC(19,4)
or integer cents



BIGSERIAL
8 bytes

VS



UUID
16 bytes



TIMESTAMP

VS



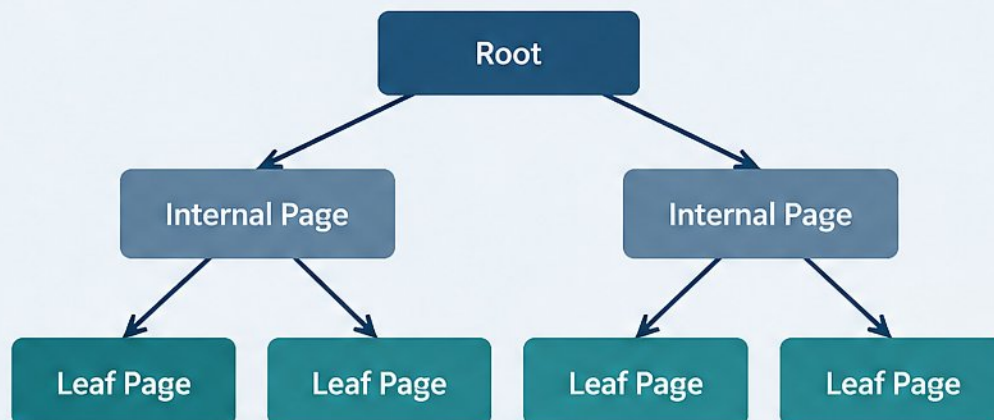
TIMESTAMPTZ



Indexing Strategy: B-tree, Composite, Covering, and Partial Indexes

- B-tree is the default for equality, range, and sort on scalars
- Use GIN for JSONB, arrays, and full-text search
- Composite order: equality columns first, then range/sort columns
- Covering indexes with INCLUDE enable index-only scans
- Partial indexes shrink size on status and soft-delete flags
- Every index slows writes: keep roughly six per OLTP table

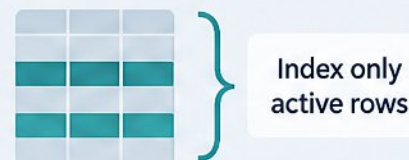
B-tree Index (Cross-section)



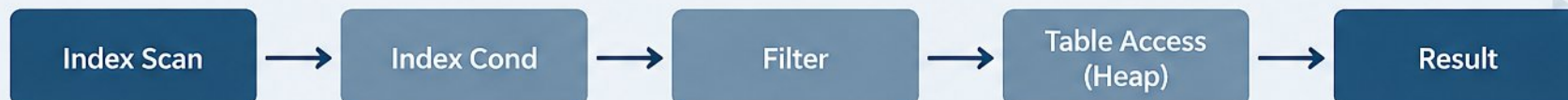
Composite Index (equality → range/sort)



Partial Index (only selected rows)



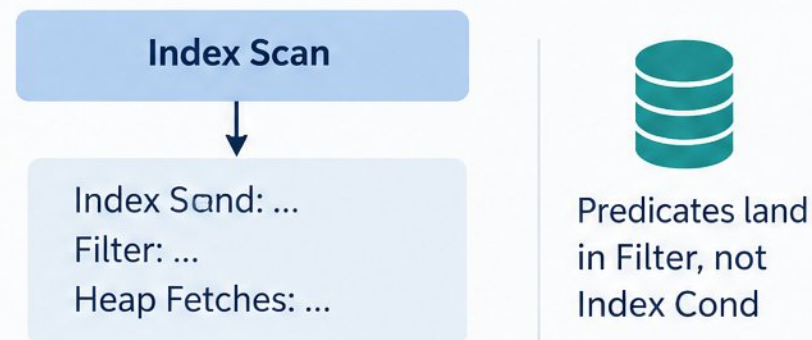
Query Plan (EXPLAIN)



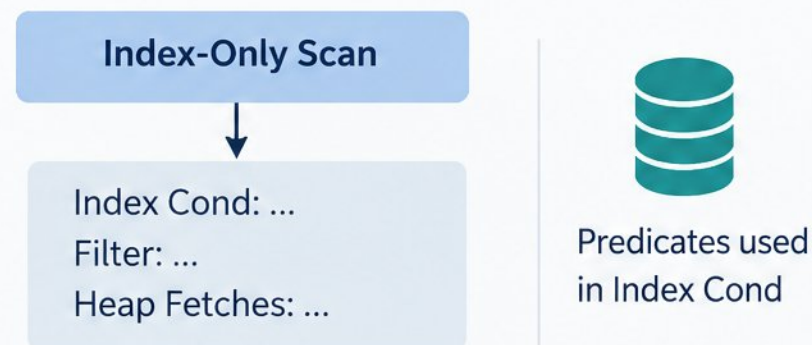
Validating Design Decisions with Query Plans

- Read EXPLAIN (ANALYZE, BUFFERS):
Index Cond, Filter, Sort, Heap Fetches
- Misordered composite index: predicates land in Filter, not Index Cond
- Index Scan vs. Index-Only Scan vs. Bitmap Heap Scan by row distribution
- Planner ignores indexes on low-cardinality uniform columns — it is correct
- Write top queries first; design indexes from access patterns
- Run VACUUM ANALYZE after bulk loads to refresh stats and visibility

Misordered Composite Index

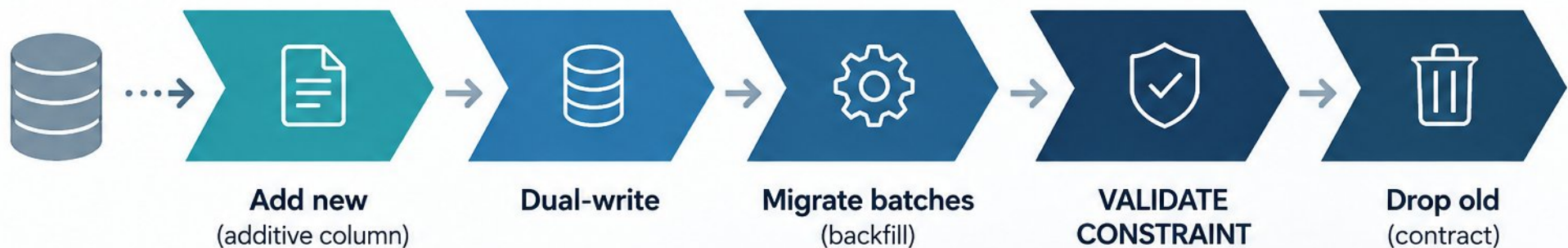


Well-Ordered Composite Index



Schema Evolution: Migrations, Version Control, and Expand-Contract

Evolve your schema safely with a disciplined, repeatable process.

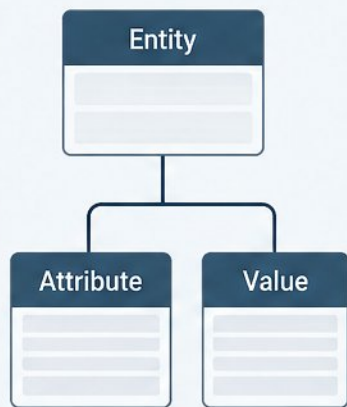


- Numbered, version-controlled migration files are the source of truth
- Backward-compatible by default: additive columns, nullable first, then constrain
- Expand → backfill → contract: add new, dual-write, migrate batches, drop old
- Zero-downtime DDL: CREATE INDEX CONCURRENTLY, NOT VALID, short lock_timeout
- N/N-1 rule: running code and current schema must always agree
- Contract is irreversible — gate on zero mismatches, no NULLs, and a snapshot

Common Antipatterns and How to Avoid Them

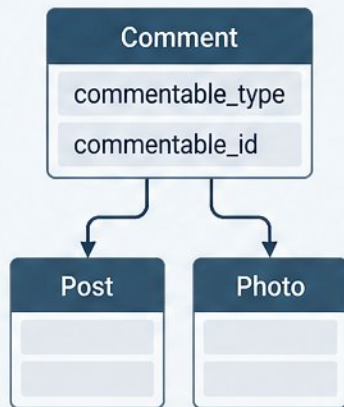
RECOGNIZE PROBLEMS. DESIGN BETTER SOLUTIONS.

EAV



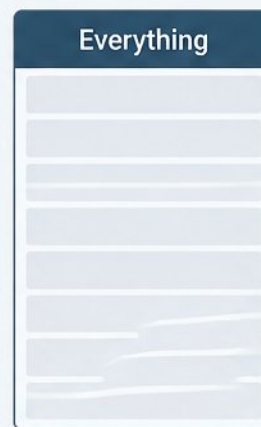
- EAV tables destroy type enforcement and spiral into join-heavy queries

Polymorphic Association



- Polymorphic type + id pairs block foreign keys; prefer separate tables or a shared parent

God Table



- God tables mix unrelated domains; split by domain and link with foreign keys

Delimited List Column



- Delimited lists belong in intersection tables; flags belong in a state enum

Boolean-Flag Soup



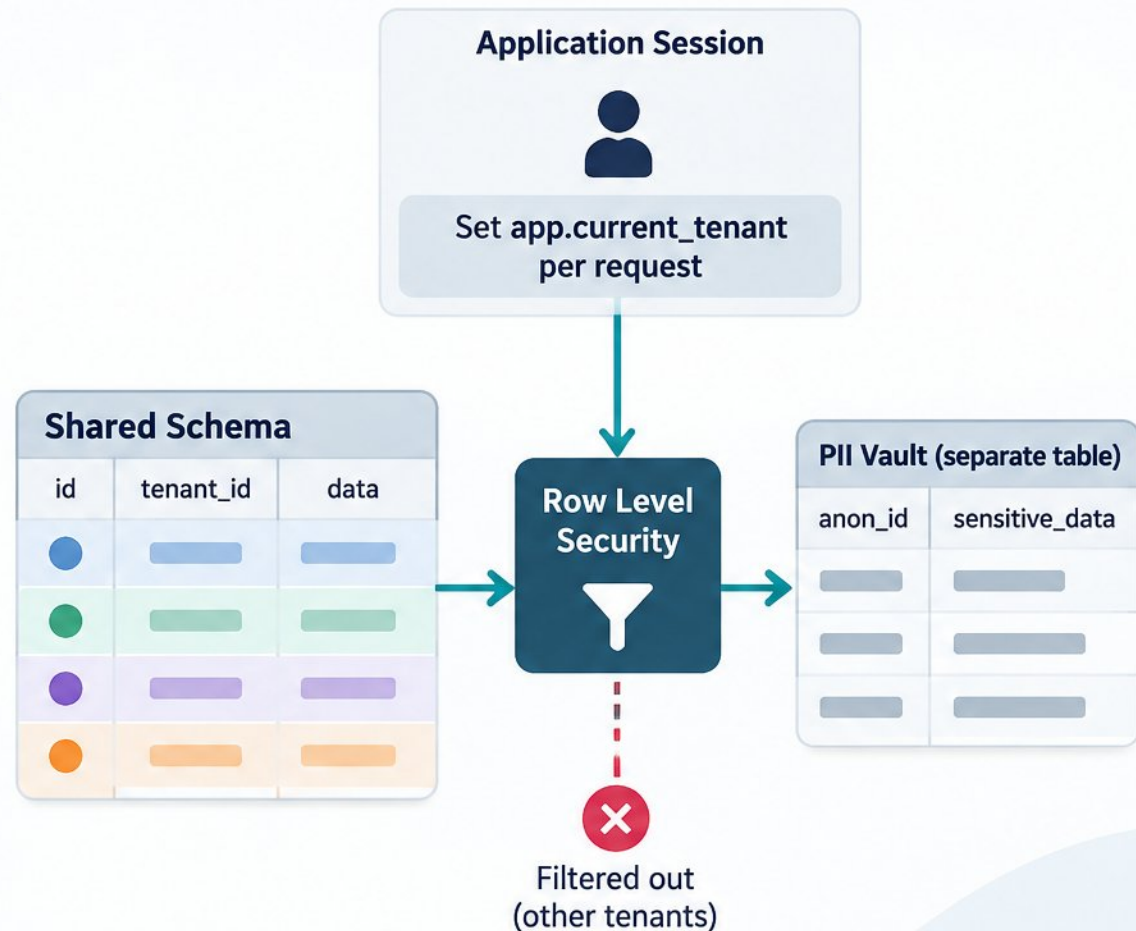
- Consistent snake_case naming; audit antipatterns with column-count and FK queries



CLEANER MODELS. STRONGER DATA. FEWER SURPRISES.

Security, Privacy, and Multi-Tenancy in Schema Design

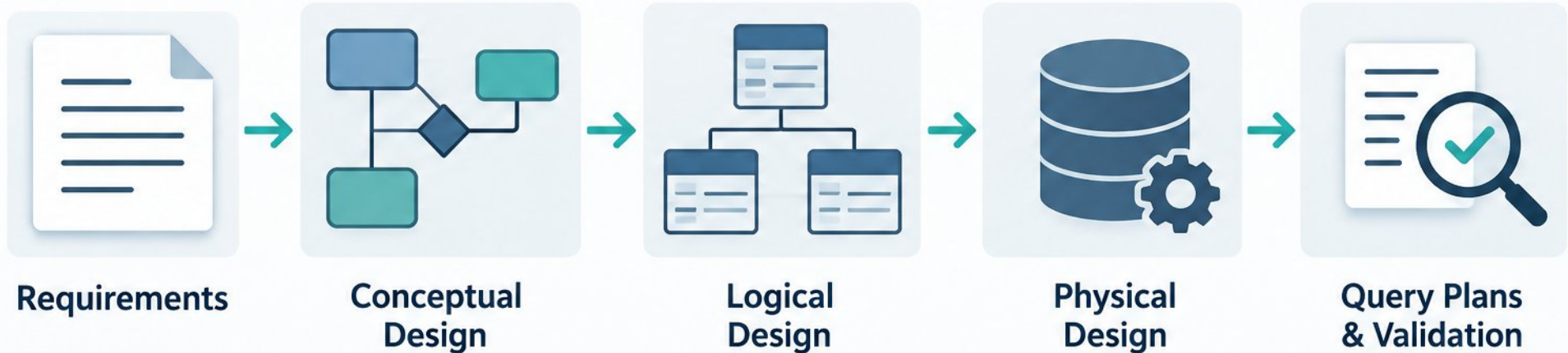
- **Shared schema + tenant_id column:** cheapest to operate, scales to 10,000+ tenants
- **Schema-per-tenant:** logical isolation auditors accept; 200–2,000 tenants per cluster
- **Database-per-tenant:** maximum isolation for regulated, \$100k+ contracts
- **Postgres RLS:** USING filters reads, WITH CHECK blocks cross-tenant writes
- **App role** must not have SUPERUSER or BYPASSRLS; FORCE RLS on owner
- Set **app.current_tenant** per request; SET LOCAL resets at transaction end
- **Never trust job payloads or cached keys** without tenant context



ISOLATE DATA.
BUILD WITH CONFIDENCE.

Hands-On Workshop: Designing a Relational Schema End to End

- Walk one scenario: requirements → conceptual → logical → physical design
- Apply normalization, keys, constraints, and indexes to a domain
- Run candidate queries and read plans to validate index choices
- Lint schema, generate test data, use review checklists; document with COMMENT ON
- Next: migration discipline, index hygiene, further reading



PersonWise.

PersonWise • AI digital-human course platform



Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/a01538a8/public