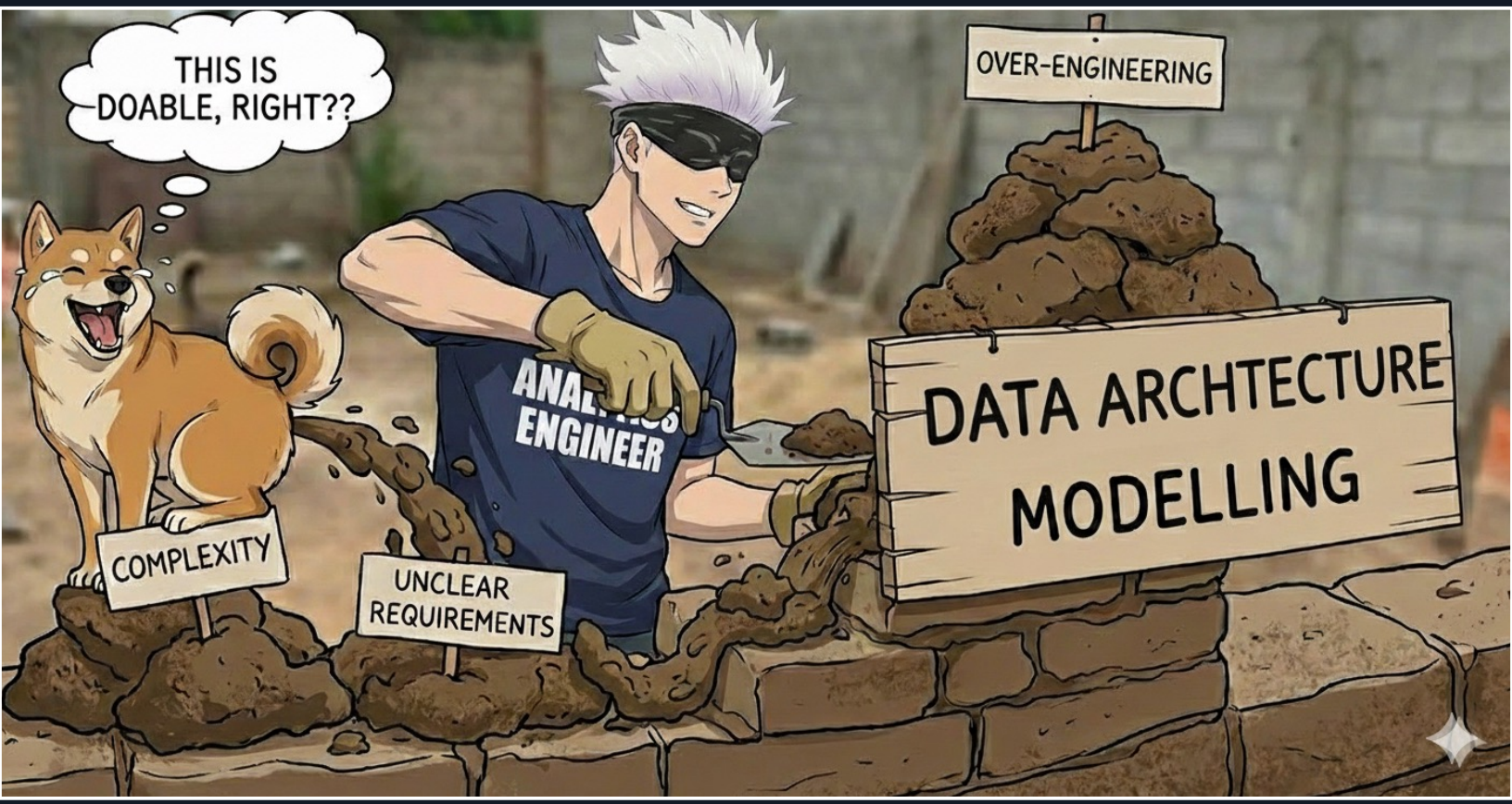


# The Complete Data Modelling Master Guide

Everything They Forgot to Teach You About Building Data as a Product

Snowflake | BigQuery | dbt | Data Vault 2.0

**Akash Sharma**

First Edition, 2026

*For every analytics engineer who's been told*
*"just put it all in one big table"*

*and for the dog who started it all.*

# Credits & Attribution

This book draws heavily on the ideas, methodologies, and published research of the authors listed below. Their original works are the foundations upon which every chapter stands. Nothing in this book claims to replace their contributions — rather, it attempts to synthesize, apply, and contextualize them for modern cloud data platforms. All errors in interpretation are mine.

**Ralph Kimball & Margy Ross** — *The Data Warehouse Toolkit, 3rd Edition (Wiley, 2013)*

Used in: Dimensional modelling methodology, star schemas, conformed dimensions, bus matrix — Parts I, II, VI, VII

**Dan Linstedt & Michael Olschimke** — *Building a Scalable Data Warehouse with Data Vault 2.0 (Morgan Kaufmann, 2015)*

Used in: Data Vault 2.0 methodology, hub/link/satellite patterns, hash keys — Parts III, III-B, III-C

**Hans Hultgren** — *Modeling the Agile Data Warehouse with Data Vault (Brighton Jones, 2012)*

Used in: Practical Data Vault patterns, bridge tables, PIT tables — Parts III-B, III-C

**Joe Celko** — *Analytics and OLAP in SQL; Trees and Hierarchies in SQL (Morgan Kaufmann)*

Used in: Advanced SQL patterns, hierarchy modelling, window functions — Parts XIV, XV

**Steve Hoberman** — *Data Modeling Made Simple; Data Modeling for the Business; Data Model Scorecard (Technics Publications)*

Used in: Data modelling process frameworks, naming conventions, quality scoring — Parts I, IV

**David C. Hay** — *Enterprise Model Patterns: Describing the World (Technics Publications, 2011)*

Used in: Universal data model patterns for parties, contracts, activities — Part IV

**Len Silverston** — *The Data Model Resource Book, Volumes 1-3 (Wiley)*

Used in: Industry data model patterns for healthcare, finance, retail, telecom — Parts VI-B, VII-B, XII

**Kent Gershkovich** — *Data Modeling with Snowflake: A Practical Guide (Technics Publications, 2023)*

Used in: Snowflake-specific modelling patterns, micro-partitions, clustering — Parts VI, VI-B

**Luca Zagni** — *Data Engineering with dbt (Packt, 2023)*

Used in: dbt project structure, materializations, testing patterns — Parts VIII, IX, X, XI

**Various Authors** — *Unlocking dbt, 2nd Edition (O'Reilly, 2024)*

Used in: dbt CI/CD, Semantic Layer, multi-project patterns — Parts X, XI

**Mark Needham, Michael Hunger & Michael Simons** — *DuckDB in Action (Manning, 2024)*

Used in: DuckDB architecture, vectorized execution, morsel-driven parallelism — Part XVI

**Jason Lee** — *DuckDB: Up and Running (O'Reilly, 2024)*

Used in: DuckDB extensions, MotherDuck, practical workflows — Part XVI

**Simon Aubury & Ned Letcher** — *Getting Started with DuckDB (O'Reilly, 2024)*

Used in: DuckDB fundamentals, Parquet integration — Part XVI

**Aditya Bhargava et al.** — *Vector Databases (O'Reilly, 2024)*

Used in: Vector database architecture, HNSW, IVF, PQ algorithms — Part XVII

**Chip Huyen** — *AI Engineering (O'Reilly, 2025)*

Used in: Embedding models, RAG architecture, multi-modal search patterns — Part XVII

## Official Documentation

- Snowflake Documentation (docs.snowflake.com)
- Google Cloud BigQuery Documentation (cloud.google.com/bigquery/docs)
- dbt Documentation (docs.getdbt.com)
- automate-dv / datavault4dbt Documentation
- DuckDB Documentation (duckdb.org/docs)
- MotherDuck Documentation (motherduck.com/docs)
- Pinecone, Weaviate, Milvus, Qdrant, Chroma, pgvector Documentation



Full acknowledgments with extended commentary on each source appear at the end of the book.

# About the Author

I'm Akash Sharma, from Delhi, India. I work with data for a living, which mostly means I write SQL, argue about naming conventions, and explain to people why you can't just JOIN everything together and call it a day.

This is my first book. I didn't set out to write something this long — it was supposed to be a short reference guide. But here's what happened: I kept running into the same problem. There are great books on Kimball. Great books on Data Vault. Solid docs on Snowflake and BigQuery. Decent stuff on dbt. But nothing that stitches it all together and shows you how these things actually work in practice, on real cloud platforms, with real messy data. So I started writing that book, and it... didn't stay short.

The thing that bugs me about most data modelling books is they live in this perfect world where data arrives clean, business rules don't change mid-sprint, and nobody asks you to "just quickly add a column" to a table that's already in production. I wanted to write something that acknowledges the mess. Every SQL example in here actually runs. Every dbt model follows patterns I've used or seen used in real projects. When I say "don't do this," it's usually because I've done it and regretted it.

Outside of data, I watch way too much anime — I'm fully convinced Talk no Jutsu would solve most stakeholder alignment meetings. I also read manhwa, mostly isekai stuff where some random guy wakes up as a level-1 nobody and ends up running an empire. Feels weirdly relatable to going from "what's a star schema" to designing enterprise data platforms.

If you find errors in this book (and you will — it's 600 pages written by one person), I'd genuinely appreciate hearing about them. If you find the book useful, that's even better. That's all I was going for.

**Akash Sharma**

*Delhi, 2026*

# Preface

Let me be upfront about something: this book started as notes. I had a messy Notion doc where I'd dump stuff I kept looking up — SCD types, Data Vault loading patterns, how CLUSTER BY works differently on Snowflake vs BigQuery. It grew over a couple of years and at some point I thought, maybe I should clean this up and share it.

"Clean it up" turned into "rewrite everything from scratch with proper examples," which turned into "well, I should cover dbt too since everyone's using it now," which turned into "might as well add case studies," and now here we are with something that's closer to an encyclopedia than a quick reference. Sorry about that. Or you're welcome. Depends on your perspective.

Here's what I tried to do differently from other books on this topic:

- **Everything runs.** Every SQL block, every dbt model, every DDL statement — I've tested them on actual Snowflake and BigQuery instances. If something doesn't work, it's a bug, not a "left as an exercise for the reader" situation.
- **Theory gets applied immediately.** I don't spend 30 pages on theory and then leave you to figure out the implementation. Every concept comes with a working example on a real platform within a page or two.
- **I cover the awkward stuff.** What do you do when your Data Vault has 200 satellites and queries take forever? How do you handle a slowly changing dimension that changes 50 times a day? When is it actually fine to use One Big Table? I don't dodge the uncomfortable questions.

The book is split into self-contained parts. You don't have to read them in order. If you already know Kimball, skip Part II. If you're a Snowflake shop, skip the BigQuery chapters. If you just need to figure out dbt materializations, go straight to Part VIII or IX. Treat it like a reference — that's how I use my own copy.

One last thing: I drew on ideas from about ten books (listed in the References at the end). This book doesn't replace any of them — Kimball's Toolkit is still the bible for dimensional modelling, Linstedt's book is still the definitive Data Vault reference. What this book does is put all of it in one place, translated into modern cloud SQL, and wrapped in enough context that you can actually go build something with it on Monday morning.

# Who This Book Is For

If you've ever stared at a Kimball star schema diagram and thought "this is elegant" and then stared at your company's actual data warehouse and thought "what happened here" — yeah, this book is for you.

**Analytics Engineers** — You write dbt models. You've got that one staging model that's 400 lines of Jinja and nobody wants to touch it. You know there's a better way to handle that dimension that changes every 5 minutes but you haven't had time to figure it out. Chapter 9 covers SCD Type 6. You're welcome.

**Data Engineers** — Someone told you to "build a data warehouse" like it's a weekend project. Now you're choosing between Data Vault, Kimball, a big denormalized table, or updating your LinkedIn. This book covers the first three. The fourth one's on you.

**Data Architects** — You need a single reference instead of 14 open browser tabs. This goes from 3NF theory to BigQuery ARRAY<STRUCT> patterns, with enough depth that you can settle arguments in design reviews.

**Students and career switchers** — If you can write a SELECT and know what a PRIMARY KEY is, you've got enough background. If you don't know what a PRIMARY KEY is, start at Chapter 1. No judgment — everyone starts somewhere.

**Senior folks and tech leads** — Useful as a desk reference for when someone drops "bridge table" in a meeting and you need a quick refresher. We've all been there.

This book probably **isn't** for you if:

- You're looking for a 20-page quick-start guide. This is 600+ pages. I don't do "quick."
- You want a vendor-neutral, platform-agnostic overview. This book gets its hands dirty with Snowflake and BigQuery SQL.
- You think data modelling is a solved problem. It isn't, and this book won't pretend it is.

**How to read this book:** Don't read it cover to cover. I mean, you can, but that's a lot. Each Part stands on its own. Start with whatever you need most. Skip what you already know. Come back to the rest when it becomes relevant. That's how books like this are meant to work.

# Table of Contents

# PART I: FOUNDATIONS OF DATA MODELLING

## 1. What Is Data Modelling and Why It Matters

Imagine an architect standing in front of an empty lot. Before heavy machinery arrives, before the first concrete is poured, the architect draws detailed blueprints. These blueprints spell out exactly where walls go, how they'll connect, what materials to use, and how the building will function. Every door, window, electrical outlet, and water pipe is precisely planned. Because a missing detail means costly rework later. And a fundamental mistake early on means you're demolishing walls that are already built. Fun times.

Data modelling is exactly this: the blueprint for your data system. Before you write a single line of SQL, before you create a database, before you build an application that depends on that database, you need to draw the blueprint. And no, I don't care if you think it's slow or bureaucratic. This blueprint answers: 'What things do we need to track? How do they connect? Which attributes matter? How'll we find stuff? How do we prevent chaos?' Get these questions right now, and you've saved yourself from 2am debugging sessions in six months.

### What Is Data Modelling?

#### The Three Levels of Data Modelling

There are three levels of data modelling, each aimed at a different crowd and solving different problems. If you mix them up, you'll spend a lot of time arguing with non-technical people about column names. So don't.

**Conceptual Data Model**

The conceptual model speaks the language of business stakeholders and executives. It shows what things the business needs to track (entities) and how they relate. No technical details. No mention of columns, tables, or databases. A CEO, product manager, or business analyst should understand it. Example: 'A Customer places Orders. Each Order contains Products. Products are categorized into Categories.' That's conceptual.

**Logical Data Model**

The logical model is for technical people but still database-agnostic. It shows tables (called 'relations' or 'entities'), columns with their types and constraints, primary keys, foreign keys, and relationships. It's specific enough to reveal design choices but not tied to a particular database system. A data architect and analyst use this extensively. SQL can be mapped from it, but there's nothing about storage mechanisms or indexing.

**Physical Data Model**

The physical model is what the DBA and database optimizer care about. It includes implementation details: specific SQL types, indexes, partitions, table spaces, performance tuning, replication strategy. The same logical model might have different physical implementations for PostgreSQL vs Oracle vs DuckDB depending on each system's strengths.

#### A Brief History of Data Modelling

| Era | Model Type | Key Figure(s) | Year(s) | Key Characteristics |
|---|---|---|---|---|
| Hierarchical | Hierarchical | IBM, Charles Bachman | 1960s-1970s | Tree structure, one parent per child |
| Network | Network (CODASYL) | Charles Bachman, others | 1970s-1980s | Graphs allowed, pointers, complex navigation |
| Relational | Relational | E.F. Codd | 1970s-present | Tables, SQL, normalized, powerful theory |

| Era | Model Type | Key Figure(s) | Year(s) | Key Characteristics |
|---|---|---|---|---|
| Dimensional | Star Schema, Snowflake | Ralph Kimball, others | 1990s-present | Fact tables, dimensions, analytics focus |
| Object-Oriented | OO-DB | Various authors | 1990s-2000s | Classes, inheritance, less adopted |
| NoSQL/Document | Document, Graph, Key-Value | Google, Amazon, MongoDB | 2000s-present | Flexible schema, horizontal scale, speed |
| Cloud/Modern | Data Vault, Cloud-native | Dan Linstedt, cloud providers | 2010s-present | Integration, auditability, cloud optimization |

Each era emerged because of new business needs. Hierarchical models were fast on disk but rigid. Network models were more flexible but hard to query. E.F. Codd revolutionized everything with the relational model—simple rules, powerful math, SQL queries. The rise of data warehousing brought dimensional modelling (Kimball). The internet explosion brought semi-structured data and NoSQL. Today we use multiple models together: relational for transactions, dimensional for analytics, document for flexible schemas, graph for relationships.

## The ANSI/SPARC Three-Schema Architecture

In 1975, the American National Standards Institute (ANSI) and Standards Planning and Requirements Committee (SPARC) proposed a framework that's still taught in universities today. It separates concerns into three independent schemas.

**External Schema (View Layer)**

What users and applications see. Different users may see different views of the same data. A customer sees only their own orders; a manager sees their team's data; an analyst sees aggregated summaries. Views provide security, simplicity, and abstraction. Each user gets exactly what they need.

**Conceptual Schema (Enterprise View)**

The single integrated data model describing the entire enterprise data. One table called Customer, one definition of what a Customer is. If Sales says 'customer_id' and Engineering says 'cust_id', the conceptual schema resolves this to one definition. This is where consistency lives.

**Internal Schema (Storage/Physical Layer)**

How data is actually stored on disk. File formats, indexes, partitions, whether a column is stored compressed. Users don't see this layer. The database engine handles the translation from requests at the conceptual layer to actual storage at the internal layer.

> *The power of ANSI/SPARC: if you change storage (migrate from spinning disk to SSD, or from MySQL to PostgreSQL), you only change the internal schema. The conceptual and external schemas stay the same, so all applications continue working.*

## Why Data Modelling Matters: Disaster Stories

### Case 1: The E-commerce Time Bomb

A startup built an e-commerce platform without proper data modelling. They stored everything flatly: each order and all its line items and prices in a single row. When they needed to update a product price, they faced a terrifying question: do we update historical orders (changing what customers paid) or leave old data and lose consistency? Within 3 years, their database was unmaintainable. Rebuilding from scratch cost 18 months and millions of dollars. A proper entity-relationship model would have separated Products from Orders from Prices, with time-stamped references.

### Case 2: The Healthcare HIPAA Disaster

A hospital system never formally modelled their patient data. Different departments created their own systems. Patient John Smith in Cardiology was 'John M. Smith' in Oncology and 'John Michael Smith' in Lab. Three separate patient records. A data quality issue missed a critical cancer diagnosis in one system. Proper data modelling with a Master Patient Index would have unified them. Cost: a lawsuit worth $2.2 million and the DBA's job.

### Case 3: The Supply Chain Cascade

A manufacturing company's data model conflated suppliers with delivery locations (one table). When they added a second warehouse for the same supplier, the model couldn't represent it without creating duplicate supplier records. They patched it with workarounds. Later, inventory queries gave wrong answers because the same supplier appeared twice. A proper M:N relationship (many suppliers can serve many locations) would have prevented this. The workaround software was eventually retired, costing $800K and six months.

> *These aren't hypothetical. They're documented in case studies. A bad data model early is far more expensive to fix than taking time to model correctly before building.*

## Data Modelling Roles and Responsibilities

| Role | Primary Responsibility | Key Activities | Tools | Audience |
|---|---|---|---|---|
| Data Architect | Enterprise-wide data strategy and governance | Conceptual/logical models, standards, integration | ERwin, ER/Studio, visio | C-suite, all teams |
| Data Engineer | Build and maintain data pipelines and warehouses | Logical/physical models, ETL, performance | dbt, Airflow, Python, SQL | Data scientists, analysts |
| Analytics Engineer | Bridge between engineering and analysis | Dimensional models, dbt, transformation logic | dbt, SQL, BI tools | Data analysts, stakeholders |
| DBA (Database Admin) | Database performance, security, backups, availability | Physical models, indexes, tuning, replication | SQL Server, Oracle, PostgreSQL | All applications |
| Business Analyst | Requirements gathering, business rule documentation | Conceptual models, glossaries, validation | Visio, Confluence, spreadsheets | Business users, architects |

## The Data Modelling Lifecycle

Here's the thing nobody tells you: data modelling doesn't end. It's not a one-time event. It's a cycle that runs for the entire life of your system. Business changes, requirements shift, someone asks for something new. Your model evolves with it.

**1. Requirements Gathering**

Interview stakeholders. What do they need to track? What questions must the system answer? What regulations apply? What are current pain points? This is the most important step and often the most skipped. Rushing here guarantees failures later.

**2. Conceptual Modelling**

Build a business-language model showing entities and relationships. No columns yet. Validate against requirements. Show it to business people and get agreement.

**3. Logical Modelling**

Normalize the model. Define attributes. Add keys. Resolve many-to-many relationships. This is precise enough that SQL can be generated from it.

**4. Physical Modelling**

Optimize for the specific database system. Add indexes, partitions, compression. Consider performance trade-offs. Choose data types.

#### 5. Implementation/Deployment

Run the DDL scripts. Load data. Validate data quality. Train users. Go live.

#### 6. Maintenance and Evolution

Monitor performance. Adjust indexes. Handle new business requirements. The model evolves as the business evolves. A data model is never truly 'done'—it's a living thing.

## Types of Data Models

Different business problems need different data models. Here's a comparison of the major types in use today.

| Model Type | Primary Purpose | Structure | Tools/DBs | When to Use | Example |
|---|---|---|---|---|---|
| Operational (OLTP) | Daily transactions | Normalized, 3NF+ | PostgreSQL, MySQL, Oracle | Order entry, banking, HR | E-commerce database |
| Analytical (OLAP) | Business intelligence | Denormalized star schema | Snowflake, BigQuery, Redshift | Reporting, dashboards, trends | Sales warehouse |
| Data Vault | Integration, compliance | Hub-link-satellite structure | Snowflake, Teradata | Enterprise data integration | ETL with full audit trail |
| Graph | Relationship queries | Nodes and edges | Neo4j, ArangoDB | Social networks, recommendations | LinkedIn connections |
| Document | Flexible schemas | JSON/BSON documents | MongoDB, CouchDB | User profiles, content | CMS, logs, events |
| Time-Series | Time-ordered events | Timestamps + measurements | InfluxDB, TimescaleDB | Metrics, IoT, monitoring | CPU usage over time |

## Forward Engineering vs Reverse Engineering

#### Forward Engineering

You start with a business problem and design the data model from scratch. This is the ideal approach for new systems. You think before you build. Most of this book is about forward engineering.

#### Reverse Engineering

You inherit an existing database and extract its data model. This is critical when joining a company with legacy systems, or when you need to understand what was built without documentation. Tools can read your database schema and generate an ER diagram.

> *Most real-world careers involve 70% reverse engineering (understanding existing systems) and 30% forward engineering (building new). Start with understanding what you have.*

## Data Modelling Tools

Professional tools help you build, document, and maintain data models. They generate SQL DDL from your diagrams and vice versa.

```
# Popular data modelling tools (not code, just reference)
# ERwin Data Modeler - enterprise standard, expensive
# ER/Studio - oracle-focused, powerful
# SqlDBM (www.sqldbm.com) - free online, browser-based, good for learning
# lucidchart.com - visual diagrams, integrates with docs
# draw.io - free, open-source, simple
```

```
# dbt (data build tool) - modern, code-as-model, git-friendly
# LookML (Looker) - model-as-code for analytics
```

For this book, we'll draw diagrams in text and SQL, which is actually closer to how real data engineers work (you live in version control, not proprietary tools).

## Learning Path for This Book

> *This chapter gave you the '30,000-foot view' of data modelling. The next chapters zoom in. Chapter 2 teaches you to think in entities and relationships. Chapter 3 teaches normalization—the rules that prevent data quality disasters. Chapters 4-5 show you the processes and standards used by real companies. By the end of Part I, you'll know how to design databases that are correct, maintainable, and scale.*

# 2. Entity-Relationship Modelling — A Deep Dive

The entity-relationship (ER) model is the lingua franca of data modelling. Peter Chen invented it in 1976, and yes, people still use it because it actually works. Here's the core idea: the world's made of things (entities) and the connections between them (relationships). Your job is to draw that world in a model. Sounds simple, right? It mostly is. You'll probably mess up a few times, but that's how you learn.

## Entities: The Things You Model

An entity is a 'thing' you want to store data about. Could be concrete (Customer, Product, Invoice) or abstract (Warranty, Permission, Contract). Here's the test: can you finish this sentence? 'We need to track ___.' If you can, that's probably an entity. If you can't, you don't need it in your model.

### Strong vs Weak Entities

#### Strong Entities

An entity that stands alone. It has its own primary key independent of any other entity. A Customer is strong; it exists whether or not it has ever placed an Order. A Product is strong. A Category is strong. Most entities are strong.

#### Weak Entities

An entity that depends on another for its identity. It doesn't have a meaningful primary key on its own. Example: an OrderLine (the line item on an invoice). There's no such thing as 'OrderLine 5' without an Order. The key must include the parent Order's ID: (order_id, line_number).

Another example: a RoomReservation for a Hotel. The key must include the Room and the Date Range, because without knowing which room and which dates, 'Reservation 1' is meaningless.

> *Weak entities exist, but they're often red flags. If you have many weak entities, ask whether your model structure is right. Sometimes an OrderLine should just be data in a junction table, not its own entity.*

## Attributes: What You Know About Entities

An attribute is just a property or characteristic of an entity. Customers have names, emails, phone numbers, addresses. Products have SKUs, prices, weights. And they're not all the same type—some are simple, some are complex, some are a nightmare. Let's break down what we're dealing with.

### Simple Attributes

A single, atomic value. customer_id (integer), email (string), birth_date (date). Can't be broken down further into business components (though technically a string is characters; we don't model at that level).

### Composite Attributes

An attribute that's made of multiple simple attributes. Customer's address could be: street, city, state, zipcode. Person's name could be: first_name, middle_name, last_name. Should you store them separately or together? Depends on whether you'll ever query them separately. If you only ever show full_name, keep it simple (one column). If you search by last_name, split it.

### Multi-Valued Attributes

An attribute that can have multiple values for a single entity instance. A Customer can have multiple phone numbers. An Employee can have multiple certifications. An Author can have multiple book titles.

In ER diagrams, you'd show this with double ellipses. In relational databases (Chapter 3), multi-valued attributes become separate tables or junction tables to maintain first normal form.

### Derived Attributes

An attribute calculated from other attributes. Age is derived from birth_date. Total_price is derived from qty × unit_price. In ER diagrams, show it with a dashed line. In databases, you usually DON'T store derived attributes; you compute them in queries (to avoid inconsistency) or cache them for performance.

Example: a Customer entity with all attribute types:

```
# Example: Customer entity attributes
CUSTOMER
  Simple: customer_id, email, birth_date
  Composite: address (street, city, state, zipcode)
  Multi-valued: phone_numbers (mobile, home, work)
  Derived: age (from birth_date), vip_status (from purchase_history)
```

## Keys: The Identifiers

A key is just one or more attributes that uniquely identify an entity instance. Without keys, you've got chaos—how do you tell one Customer from another? One Order from another? Keys are how databases stay sane. They're foundational.

| Key Type | Definition | Uniqueness | Can Be NULL? | Example |
|---|---|---|---|---|
| Candidate | Any attribute(s) that could uniquely identify | Yes | No | email, ssn, phone |
| Primary | The chosen candidate key used in database | Yes | No | customer_id (surrogate) |
| Alternate | Candidate key not chosen as primary | Yes | No | email (if customer_id is PK) |
| Foreign | PK from another table, creates relationship | No (in sense of ref integrity) | Yes | customer_id in Orders table |
| Composite | Key made of 2+ columns | Yes (combination) | Depends | order_id + line_number |
| Surrogate | Artificial key, no business meaning | Yes | No | auto-increment id, UUID |
| Natural | Key from business attributes | Yes | No | ssn, email, sku |

### Natural Keys vs Surrogate Keys

One of the great debates in database design. Natural keys use business data; surrogate keys are artificial.

#### Natural Key Example: Email

```
CREATE TABLE customer_natural (
  email VARCHAR(255) PRIMARY KEY,
  first_name VARCHAR(100),
  last_name VARCHAR(100),
  phone VARCHAR(20)
);
```

#### Surrogate Key Example: Auto-incrementing ID

```
CREATE TABLE customer_surrogate (
  customer_id BIGINT PRIMARY KEY AUTO_INCREMENT,
  email VARCHAR(255) UNIQUE NOT NULL,
  first_name VARCHAR(100),
  last_name VARCHAR(100),
  phone VARCHAR(20)
```

```
);
```

**Natural Keys: Pros and Cons**

Pros:

- Smaller primary key (email is shorter than integer, less storage)
- No artificial column cluttering the table
- Business meaning is clear (you KNOW what email is)
- Can't accidentally use the same ID for two different emails

Cons:

- If business attribute changes, foreign keys cascade everywhere (if email is PK and someone needs two emails...)
- Performance impact if PK is string and referenced in many foreign keys
- Complex composite keys (order_date + customer_id + product_id) are hard to type and join on
- Privacy: if email is visible in logs as a PK, it's exposed

**Surrogate Keys: Pros and Cons**

Pros:

- Small, integer, efficient for joining and indexing
- No business meaning, so business changes don't force key changes
- Can change natural attributes without affecting relationships
- Privacy friendly (logs show ID numbers, not emails)

Cons:

- Extra column that has no semantic meaning
- You still need unique constraints on business attributes (like email UNIQUE)
- Can lose business-level constraints accidentally (nothing prevents two customers with same email)

> *Modern practice: use surrogate keys as primary keys (for performance and safety) AND add unique constraints on natural key attributes (for business correctness). Best of both worlds.*

## Relationships: How Entities Connect

A relationship is just how one entity connects to another. Customers place Orders. Employees work in Departments. Students enroll in Courses. These connections are as important as the entities themselves—if you miss them, your model falls apart.

### Unary Relationships (Self-References)

An entity related to itself. Example: an Employee has a Manager (who is also an Employee). A Category can have a Parent_Category. A Person can have Siblings (other Persons).

```
-- Unary relationship: Employee reporting structure
CREATE TABLE employee (
  employee_id INT PRIMARY KEY,
  name VARCHAR(100),
  manager_id INT,
  FOREIGN KEY (manager_id) REFERENCES employee(employee_id)
);

-- Example data:
```

```
INSERT INTO employee VALUES
  (1, 'Alice', NULL),        -- CEO, no manager
  (2, 'Bob', 1),             -- Bob reports to Alice
  (3, 'Charlie', 1),         -- Charlie reports to Alice
  (4, 'Diana', 2);           -- Diana reports to Bob
```

## Binary Relationships

Two entities involved. This is the most common type. Customer and Order, Product and Category, Employee and Department.

## Ternary and N-ary Relationships

Three or more entities involved. A Supplier supplies a Part to a Project (all three needed). A Teacher teaches a Course in a Semester to a Room. These are less common and often can be broken into binary relationships, but sometimes they're necessary.

## Cardinality: The Numbers

Cardinality describes how many instances of one entity relate to instances of another. There are four basic types:

**1:1 (One-to-One)**

One Customer has exactly one Primary_Contact_Person. One Person has exactly one Passport (in most countries). One Driver has one Driver's License. These are relatively rare; most relationships are 1:M or M:N. Use 1:1 only when the relationship is truly exclusive.

```
-- 1:1 relationship
CREATE TABLE person (
  person_id INT PRIMARY KEY,
  name VARCHAR(100)
);

CREATE TABLE passport (
  passport_id INT PRIMARY KEY,
  person_id INT UNIQUE NOT NULL,  -- UNIQUE makes it 1:1, not 1:M
  country VARCHAR(50),
  expiry_date DATE,
  FOREIGN KEY (person_id) REFERENCES person(person_id)
);
```

**1:M (One-to-Many)**

One Customer has many Orders. One Order has many OrderLines. One Category has many Products. Most relationships in a real database are 1:M. One side is the 'parent'; many side is the 'child'.

```
-- 1:M relationship
CREATE TABLE customer (
  customer_id INT PRIMARY KEY,
  name VARCHAR(100)
);

CREATE TABLE orders (
  order_id INT PRIMARY KEY,
  customer_id INT NOT NULL,  -- FK, many orders per customer
  order_date DATE,
  FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
);
```

### M:N (Many-to-Many)

Many Students enroll in many Courses. Many Authors write many Books. Many Employees join many Projects. To represent M:N in relational databases, you create a junction table (also called associative table or bridge table) that holds foreign keys to both sides.

```
-- M:N relationship via junction table
CREATE TABLE student (
  student_id INT PRIMARY KEY,
  name VARCHAR(100)
);

CREATE TABLE course (
  course_id INT PRIMARY KEY,
  title VARCHAR(100)
);

-- Junction table
CREATE TABLE enrollment (
  student_id INT NOT NULL,
  course_id INT NOT NULL,
  enroll_date DATE,
  grade CHAR(1),
  PRIMARY KEY (student_id, course_id),
  FOREIGN KEY (student_id) REFERENCES student(student_id),
  FOREIGN KEY (course_id) REFERENCES course(course_id)
);

-- Example:
-- Student 1 enrolls in Course 1 and Course 2
-- Student 2 enrolls in Course 1 and Course 3
```

## Optionality (Participation): Must It Exist?

Cardinality tells you the count (1 or many). Optionality tells you whether the relationship is required.

### Total Participation (Mandatory)

Every instance of one entity MUST participate in the relationship. Every OrderLine MUST belong to an Order (no orphan line items). This is shown with a thick/bold line or double line in ER diagrams.

### Partial Participation (Optional)

Not every instance needs the relationship. A Customer may or may not have returned an Order (some orders are never returned). An Employee may or may not manage anyone (some employees don't have reports). Shown with a single line in ER diagrams.

```
-- Optional FK (partial participation)
CREATE TABLE employee (
  employee_id INT PRIMARY KEY,
  name VARCHAR(100),
  manager_id INT,  -- Can be NULL, not every employee has a manager
  FOREIGN KEY (manager_id) REFERENCES employee(employee_id)
);

-- Mandatory FK (total participation)
CREATE TABLE order_line (
  order_id INT NOT NULL,  -- NOT NULL = must have order
  line_number INT NOT NULL,
  order_date DATE,
  PRIMARY KEY (order_id, line_number),
  FOREIGN KEY (order_id) REFERENCES orders(order_id)
```

```
);
```

### Identifying vs Non-Identifying Relationships

An identifying relationship is one where the parent's key becomes part of the child's primary key. A non-identifying relationship is where the child has its own independent key.

```
-- IDENTIFYING relationship: OrderLine is weak; its key includes Order
CREATE TABLE orders (
  order_id INT PRIMARY KEY,
  customer_id INT,
  order_date DATE
);

CREATE TABLE order_line (
  order_id INT NOT NULL,
  line_number INT NOT NULL,
  product_id INT,
  quantity INT,
  PRIMARY KEY (order_id, line_number),  -- order_id IS PART OF PK
  FOREIGN KEY (order_id) REFERENCES orders(order_id)
);

-- NON-IDENTIFYING relationship: Product is strong; it has its own key
CREATE TABLE product (
  product_id INT PRIMARY KEY,
  name VARCHAR(100),
  category_id INT,
  FOREIGN KEY (category_id) REFERENCES category(category_id)
);
```

## ER Notation Systems

There are multiple ways to draw ER diagrams. They all show the same ideas but with different symbols—it's like speaking the same language with different accents. I'll show you all the major ones using the same example so you can see how they compare.

### Chen's Original Notation (Academic)

Entities are rectangles, attributes are ovals, relationships are diamonds. Connecting lines show relationships. This notation is the most comprehensive (can show all attribute types) but rarely used in industry anymore.

Library example in Chen notation (textual description): Rectangle 'Author' with ovals for {author_id, name, birthdate}. Rectangle 'Book' with ovals for {isbn, title, publication_year}. Diamond 'writes' connecting them (1 Author writes many Books, M:N because Authors can co-author). Rectangle 'Patron' with {patron_id, name, address}. Diamond 'borrows' connecting Patron to Book (many Patrons borrow many Books).

### Crow's Foot (IE) Notation — Industry Standard

Most widely used in industry today. Entities are rectangles with attributes listed inside. Relationships are lines with symbols at the ends: crow's foot (many), single line (one), circle (optional). Clean and readable.

Library example in Crow's Foot: Rectangle 'Author' with PK author_id, name, birthdate. Rectangle 'Book' with PK isbn, title, publication_year, author_id (FK). Connecting line from Author to Book: one Author, many Books (crow's foot on Book end). Separate rectangle 'Patron' with PK patron_id, name, address. Rectangle 'Checkout' junction table with PK {patron_id, isbn, checkout_date}, showing M:N between Patron and Book.

### IDEF1X — US Government Standard

Used heavily in government and defense. Entities are boxes. Primary keys are at top. Foreign keys are at bottom. Relationships are lines with special symbols. Distinguishes identifying (bold line) from non-identifying (dashed) relationships.

### Barker's Notation — Oracle's Preference

Used by Oracle's tools. Similar to Crow's Foot but slightly different symbols. Ovals at relationship ends instead of crow's foot. If you're in Oracle shops, you might see this.

### UML Class Diagrams

Object-oriented notation. Classes are boxes with three sections: class name, attributes, methods. Less common for pure data modelling (those methods don't apply to databases) but increasingly used in modern shops.

| Notation | Entity Symbol | One-to-Many | Many-to-Many | Optional FK | Best For | Popularity |
|---|---|---|---|---|---|---|
| Chen | Rectangle | Diamond+line | Diamond+line | Not shown | Academic | Textbooks only |
| Crow's Foot | Rectangle | Single line to crow's foot | Junction table | Circle + line | Industry | Most common |
| IDEF1X | Box | Bold line | Junction+special | Different line | Government | Contracts |
| Barker | Rectangle | Oval + line | Junction | Different oval | Oracle systems | Legacy systems |
| UML | Class box | Multiplicity numbers | Multiplicity numbers | 0..1 notation | OO systems | Modern apps |

*Don't memorize all notation systems. Learn one (Crow's Foot) and use it consistently. When you see a different notation, just look up what the symbols mean.*

## Supertypes and Subtypes (Generalization/Specialization)

Here's a pattern you'll see: some entities are basically the same but with slight differences. Vehicle is your supertype—Car, Truck, Motorcycle are subtypes. They all have color, year, VIN. But Trucks have payload, Cars have door count, Motorcycles have sidecar options. How do you model that without losing your mind? Let's see.

### Exclusive vs Inclusive Subtypes

#### Exclusive (Disjoint)

A vehicle is EITHER a Car OR a Truck OR a Motorcycle, never more than one. Once you classify it, that's it.

#### Inclusive (Overlapping)

An Employee can be both a Manager AND an Engineer (overlapping roles). A Product can be both 'Digital' and 'Gift-wrapped'.

```
-- Implementation: Single table with type discriminator
CREATE TABLE vehicle (
  vehicle_id INT PRIMARY KEY,
  color VARCHAR(50),
  year INT,
  vin VARCHAR(20) UNIQUE,
  vehicle_type VARCHAR(20),  -- 'CAR', 'TRUCK', 'MOTORCYCLE'
  num_doors INT,              -- NULL for non-cars
  payload_capacity INT,       -- NULL for non-trucks
  has_sidecar BOOLEAN         -- NULL for non-motorcycles
);
```

```
-- Implementation: Separate tables (inheritance)
CREATE TABLE vehicle_base (
  vehicle_id INT PRIMARY KEY,
  color VARCHAR(50),
  year INT,
  vin VARCHAR(20) UNIQUE
);

CREATE TABLE car (
  vehicle_id INT PRIMARY KEY,
  num_doors INT,
  FOREIGN KEY (vehicle_id) REFERENCES vehicle_base(vehicle_id)
);

CREATE TABLE truck (
  vehicle_id INT PRIMARY KEY,
  payload_capacity INT,
  FOREIGN KEY (vehicle_id) REFERENCES vehicle_base(vehicle_id)
);
```

## Converting ER to Relational Tables

Time to turn that ER model into actual SQL tables. It's a mechanical process, but it's easy to miss details. Let me walk you through it, because getting this wrong is how you end up with garbage schemas that haunt you for years.

### Step 1: Create tables for strong entities

Each strong entity becomes a table. Each simple attribute becomes a column. Include the primary key.

```
-- Author entity -> Table
CREATE TABLE author (
  author_id INT PRIMARY KEY,
  first_name VARCHAR(100),
  last_name VARCHAR(100),
  birth_date DATE
);
```

### Step 2: Handle 1:M relationships

Place the foreign key on the 'many' side. A Book is written by one Author, so author_id goes in the Book table.

```
CREATE TABLE book (
  isbn VARCHAR(20) PRIMARY KEY,
  title VARCHAR(200),
  publication_year INT,
  author_id INT NOT NULL,
  FOREIGN KEY (author_id) REFERENCES author(author_id)
);
```

### Step 3: Handle M:N relationships

Create a junction table. Its primary key is the composite of both foreign keys. Any attributes specific to the relationship go here.

```
-- Patron and Book have M:N (many patrons borrow many books)
CREATE TABLE patron (
  patron_id INT PRIMARY KEY,
  name VARCHAR(100),
  email VARCHAR(100)
```

```
);

CREATE TABLE checkout (
  patron_id INT NOT NULL,
  isbn VARCHAR(20) NOT NULL,
  checkout_date DATE NOT NULL,
  due_date DATE,
  return_date DATE,
  PRIMARY KEY (patron_id, isbn, checkout_date),
  FOREIGN KEY (patron_id) REFERENCES patron(patron_id),
  FOREIGN KEY (isbn) REFERENCES book(isbn)
);
```

### Step 4: Handle weak entities

Weak entities include their parent's key as part of their own primary key, and include a foreign key reference.

```
-- Example: AccountTransaction is weak (depends on Account)
CREATE TABLE account (
  account_id INT PRIMARY KEY,
  account_type VARCHAR(20)
);

CREATE TABLE account_transaction (
  account_id INT NOT NULL,
  transaction_number INT NOT NULL,
  amount DECIMAL(10, 2),
  transaction_date DATE,
  PRIMARY KEY (account_id, transaction_number),
  FOREIGN KEY (account_id) REFERENCES account(account_id)
);
```

### Step 5: Handle supertypes/subtypes

Two main options: single-table (with a type discriminator), or separate tables for each subtype with FK to supertype. See earlier code example above.

## Common ER Modelling Mistakes

I've seen these errors so many times. They always seem like good ideas at 3pm on Friday, and they always bite you on Monday morning. Let me show you what to watch for.

| Mistake | Example | Problem | Fix |
|---|---|---|---|
| Storing lists in one column | phone_numbers VARCHAR(500) with '555-1234,555-5678' | Can't query individual phones, anomalies | Create phone_number table |
| Hybrid primary key | PK is (customer_id, email), but should be one | Can't guarantee either is unique alone | Choose ONE candidate key as PK |
| Missing NOT NULL on FK | customer_id INT FOREIGN KEY, can be NULL | Orphan records, quality issues | customer_id INT NOT NULL FK |
| M:N without junction table | Storing both IDs in one table | Can't represent the relationship properly | Create junction/bridge table |
| Storing derived data | Storing age AND birth_date (redundant) | Age becomes wrong if not updated | Compute age in SELECT, store only birth_date |
| Overusing weak entities | Making everything depend on parent | Tight coupling, hard to reuse | Use strong entities where possible |
| Composite keys with meaning loss | Order key is (date, customer, product) | Unclear what uniquely identifies order | Use surrogate key (order_id) |

| Mistake | Example | Problem | Fix |
| --- | --- | --- | --- |
| Forgetting relationships in model | Order table but no FK to Customer | Can't actually link orders to customers | Add FK columns and define constraints |

# 3. Normalization — From First to Sixth Normal Form

Normalization is the formal process that turns a messy spreadsheet into a database that doesn't make you want to scream. It's based on E.F. Codd's relational model—the guy who figured out how to make data reliable. You'll either love it or hate it, but trust me, you'll appreciate it the first time you don't have to fix a data corruption bug at 3am.

## Why Normalize? The Update Anomalies

Here's the real reason we normalize: to avoid three terrible things—insert anomalies, update anomalies, and delete anomalies. They're not theoretical. I've seen companies lose millions to these. Let me show you all three at once, and you'll understand why this matters.

```
-- BAD: Denormalized employee_project table
CREATE TABLE employee_project (
  emp_id INT,
  emp_name VARCHAR(100),
  emp_phone VARCHAR(20),
  project_id INT,
  project_name VARCHAR(100),
  project_budget DECIMAL(10, 2)
);

-- Example data:
-- emp_id | emp_name | emp_phone   | project_id | project_name | budget
-- 101    | Alice    | 555-0001    | 1          | ProjectX     | 100000
-- 101    | Alice    | 555-0001    | 2          | ProjectY     | 150000
-- 102    | Bob      | 555-0002    | 1          | ProjectX     | 100000
```

### INSERT Anomaly

You want to hire a new employee, Charlie, but don't assign them to a project yet. You CAN'T insert Charlie's record because project_id and project_name are not nullable in many designs. You're forced to assign them to a dummy project. This is nonsensical.

### UPDATE Anomaly

Alice's phone number changes to 555-9999. But Alice is on two projects (two rows). If you only update one row, the database becomes inconsistent: Alice's phone is 555-0001 in one row and 555-9999 in another. Same employee, different phone. Bad.

### DELETE Anomaly

Bob is only on ProjectX. You delete his row because the project is cancelled. Now you've lost information about Bob the employee, not just Bob's assignment to the project. Oops. You wanted to delete the relationship, but you deleted the entity.

> *All three anomalies can be fixed by normalization. The general rule: store each fact only once.*

## Functional Dependencies

Functional dependency is the mathy concept behind normalization. It's simpler than it sounds: A → B just means 'if you know A, you automatically know B.' That's it. Your employee_id tells you their name. Their SSN tells you their entire identity. Let's see what that looks like.

```
-- Examples of functional dependencies:
employee_id → employee_name       -- Know emp_id, know their name
product_id → product_price        -- Know product_id, know its price
email → customer_id               -- Each email maps to exactly one customer
ssn → name, address, birth_date   -- Know SSN, know all personal info

-- NOT a functional dependency:
customer_id → product_id          -- One customer can buy many products
```

Here's the rule: a good table has non-key attributes that depend only on the key (and nothing but the key). If an attribute depends on something else, you've got anomalies waiting to happen. That's bad.

## UNNORMALIZED FORM (UNF) — Raw Data

This is data as it might come from a spreadsheet or form. It has repeating groups and non-atomic values.

```
-- UNNORMALIZED: Spreadsheet-style order data
Order_Data:
  order_id | customer | items
  1001     | John     | iPhone 14 ($999), AirPods ($199), USB Cable ($19)
  1002     | Mary     | MacBook Pro ($2500), AppleCare ($379)
  1003     | Jane     | iPad ($599), Apple Pencil ($129), Case ($49)

-- The 'items' column has repeating groups (product, price pairs)
-- A single cell contains multiple values
```

## FIRST NORMAL FORM (1NF) — Atomic Values

Rule: All attribute values must be atomic (indivisible). No repeating groups. No arrays or lists in cells.

```
-- FIRST NORMAL FORM: Same data, properly structured
CREATE TABLE customer (
  customer_id INT PRIMARY KEY,
  customer_name VARCHAR(100)
);

CREATE TABLE product (
  product_id INT PRIMARY KEY,
  product_name VARCHAR(100),
  price DECIMAL(10, 2)
);

CREATE TABLE orders (
  order_id INT PRIMARY KEY,
  customer_id INT NOT NULL,
  FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
);

CREATE TABLE order_item (
  order_id INT NOT NULL,
  product_id INT NOT NULL,
  quantity INT,
  unit_price DECIMAL(10, 2),
  PRIMARY KEY (order_id, product_id),
  FOREIGN KEY (order_id) REFERENCES orders(order_id),
  FOREIGN KEY (product_id) REFERENCES product(product_id)
);
```

Now each cell has one atomic value. The repeating group (products in an order) is represented by multiple rows in order_item, not multiple values in one cell.

## SECOND NORMAL FORM (2NF) — No Partial Dependencies

Rule: Must be in 1NF, AND no non-key attribute can depend on only part of a composite primary key.

```
-- NOT 2NF: order_item table violates this
CREATE TABLE order_item_bad (
  order_id INT NOT NULL,
  product_id INT NOT NULL,
  quantity INT,
  product_name VARCHAR(100),     -- Depends on product_id only, not (order_id, product_id)
  product_price DECIMAL(10, 2),  -- Same problem
  PRIMARY KEY (order_id, product_id)
);

-- SECOND NORMAL FORM: Fix by separating
-- Keep order_item small
CREATE TABLE order_item (
  order_id INT NOT NULL,
  product_id INT NOT NULL,
  quantity INT,
  PRIMARY KEY (order_id, product_id),
  FOREIGN KEY (order_id) REFERENCES orders(order_id),
  FOREIGN KEY (product_id) REFERENCES product(product_id)
);

-- product_name and product_price stay in product table
CREATE TABLE product (
  product_id INT PRIMARY KEY,
  product_name VARCHAR(100),
  product_price DECIMAL(10, 2)
);
```

> *2NF only matters if your primary key is composite. If your PK is a single surrogate key (like product_id), you're already in 2NF by definition (there's no 'part of the key').*

## THIRD NORMAL FORM (3NF) — No Transitive Dependencies

Rule: Must be in 2NF, AND no non-key attribute can depend on another non-key attribute (no transitive dependencies). Codd's famous quote: 'Every non-key attribute must depend on the key, the whole key, and nothing but the key.'

```
-- NOT 3NF: employee table has transitive dependency
CREATE TABLE employee_bad (
  emp_id INT PRIMARY KEY,
  emp_name VARCHAR(100),
  dept_id INT,
  dept_name VARCHAR(100),       -- Depends on dept_id, not emp_id!
  dept_location VARCHAR(100)    -- Same problem
);

-- If emp_id is 101 and 102 both in dept_id 5, and you store dept_name='Sales' twice,
-- then update one to 'Sales & Marketing', the database is inconsistent.

-- THIRD NORMAL FORM: Fix by separating
CREATE TABLE employee (
  emp_id INT PRIMARY KEY,
  emp_name VARCHAR(100),
  dept_id INT NOT NULL,
  FOREIGN KEY (dept_id) REFERENCES department(dept_id)
);
```

```
CREATE TABLE department (
  dept_id INT PRIMARY KEY,
  dept_name VARCHAR(100),
  dept_location VARCHAR(100)
);
```

3NF is the most common stopping point in practice. It eliminates most anomalies and maintains good query performance.

## BOYCE-CODD NORMAL FORM (BCNF) — Stricter than 3NF

Rule: Every determinant (attribute that determines another) must be a candidate key. BCNF is stricter than 3NF. Most tables that are 3NF are also BCNF, but not always.

```
-- Example where 3NF != BCNF: Professor-Course-Time
-- A professor teaches multiple courses, each at different times
-- A course is taught by only one professor
-- A time slot is used by only one professor

CREATE TABLE class_schedule_3nf (
  course_id INT,
  professor_id INT,
  time_slot VARCHAR(20),
  professor_name VARCHAR(100),
  PRIMARY KEY (course_id, time_slot)
);

-- Candidate keys: (course_id, time_slot), and (professor_id, time_slot)
-- But professor_id → professor_name
-- professor_id is NOT a candidate key, so this is 3NF but not BCNF

-- BCNF: Separate
CREATE TABLE class_schedule_bcnf (
  course_id INT PRIMARY KEY,
  professor_id INT,
  FOREIGN KEY (professor_id) REFERENCES professor(professor_id)
);

CREATE TABLE professor_schedule (
  professor_id INT PRIMARY KEY,
  time_slot VARCHAR(20)
);
```

*BCNF is theoretical perfection but can sometimes make queries harder. Usually 3NF is a better practical choice.*

## FOURTH NORMAL FORM (4NF) — Independent Multi-Valued Dependencies

Rule: No independent multivalued dependencies. This handles cases where one attribute has multiple independent values related to another attribute.

```
-- NOT 4NF: Employee with both skills and languages (independent)
CREATE TABLE employee_skill_language (
  emp_id INT,
  skill VARCHAR(50),
  language VARCHAR(50),
  PRIMARY KEY (emp_id, skill, language)
);

-- Example data:
```

```
-- emp_id=1: skill='Java', language='English'
-- emp_id=1: skill='Java', language='Spanish'
-- emp_id=1: skill='Python', language='English'
-- emp_id=1: skill='Python', language='Spanish'

-- If Bob speaks English and knows Java, does that mean he speaks English only with Java?
-- No, those are independent. This forces us to store redundant combinations.

-- FOURTH NORMAL FORM: Separate tables
CREATE TABLE employee_skill (
  emp_id INT,
  skill VARCHAR(50),
  PRIMARY KEY (emp_id, skill),
  FOREIGN KEY (emp_id) REFERENCES employee(emp_id)
);

CREATE TABLE employee_language (
  emp_id INT,
  language VARCHAR(50),
  PRIMARY KEY (emp_id, language),
  FOREIGN KEY (emp_id) REFERENCES employee(emp_id)
);
```

## FIFTH NORMAL FORM (5NF) — Lossless Decomposition

Rule: Can be decomposed into lossless join dependencies. This is rare and theoretical. It handles complex multi-way relationships.

```
-- Example: Supplier-Part-Project (all three needed to define a supply)
CREATE TABLE supply (
  supplier_id INT,
  part_id INT,
  project_id INT,
  PRIMARY KEY (supplier_id, part_id, project_id)
);

-- In 5NF, you'd decompose this into three tables
-- supplier_part, supplier_project, part_project
-- and reconstruct with joins. Rarely used in practice.
```

## SIXTH NORMAL FORM (6NF) — Temporal Data

Rule: Handles temporal (time-varying) data. Each attribute is stored in its own table with valid_from and valid_to timestamps. Related to Data Vault methodology.

```
-- Traditional table: Only current state
CREATE TABLE employee_current (
  emp_id INT PRIMARY KEY,
  name VARCHAR(100),
  salary DECIMAL(10, 2),
  department_id INT
);

-- SIXTH NORMAL FORM: Temporal with full history
CREATE TABLE employee_6nf_name (
  emp_id INT,
  name VARCHAR(100),
  valid_from TIMESTAMP,
  valid_to TIMESTAMP,
```

```
  PRIMARY KEY (emp_id, valid_from)
);

CREATE TABLE employee_6nf_salary (
  emp_id INT,
  salary DECIMAL(10, 2),
  valid_from TIMESTAMP,
  valid_to TIMESTAMP,
  PRIMARY KEY (emp_id, valid_from)
);

CREATE TABLE employee_6nf_department (
  emp_id INT,
  department_id INT,
  valid_from TIMESTAMP,
  valid_to TIMESTAMP,
  PRIMARY KEY (emp_id, valid_from)
);

-- Now you can query: What was Alice's salary in 2022?
-- SELECT salary FROM employee_6nf_salary
-- WHERE emp_id=123 AND valid_from <= '2022-01-01' AND valid_to > '2022-01-01';
```

6NF is related to Data Vault architecture (satellites are essentially 6NF). Used in systems needing full audit trails.

## Normalization Comparison Table

| Normal Form | Rule | Violation Example | How to Fix | Typical Use Case |
|---|---|---|---|---|
| UNF | Repeating groups allowed | items: 'iPhone, AirPods' | Flatten to 1NF | Raw spreadsheets |
| 1NF | Atomic values only | Multi-valued cell | Separate into rows | Most databases |
| 2NF | No partial dependencies | Composite key with partial dependency | Separate tables | Tables with composite keys |
| 3NF | No transitive dependencies | emp_id → dept_id → dept_name | Separate into 3 tables | Operational databases |
| BCNF | Every determinant is candidate key | Non-candidate determines attribute | Remove or restructure | Perfect normalization |
| 4NF | No independent multi-valued deps | emp_id → skill AND language | Separate into 2 tables | Complex attributes |
| 5NF | Lossless join decomposition | Complex ternary relationships | Decompose carefully | Rare, theoretical |
| 6NF | Temporal/time-varying data | No history tracked | Add valid_from/valid_to | Audit trails, Data Vault |

## Normalization Decision Guide

Which normal form should you aim for in practice? It depends on your use case.

- OLTP (Order processing, Banking): 3NF or BCNF. Fast inserts/updates, no anomalies.
- Data Warehouse (Analytics): Denormalized. Star schema with fact and dimension tables. Optimized for queries.
- Audit/Compliance: 6NF with temporal dimensions. Need full history and change tracking.
- Master Data: 3NF+unique constraints. Prevent duplicates; ensure data quality.
- NoSQL/Document: Denormalized. Schema flexibility; duplication acceptable.

## Denormalization: When to Break the Rules

Sometimes you intentionally denormalize for performance. This is acceptable if done carefully.

### Pre-Joined Tables

Store employee and department together to avoid a join on every query. Trade: more storage, harder to update department.

### Summary Tables

Calculate totals once and store them instead of summing millions of rows each query. Trade: need to update summaries, data may become stale.

### Redundant Columns

Store product_price in the order_item table even though it's also in product. Why? Historical prices. If product price changes, old orders should show what the customer actually paid, not the new price.

| Denormalization Pattern | When to Use | Benefit | Cost | Example |
|---|---|---|---|---|
| Materialized view | Slow aggregation queries | Query speed 10-100x faster | Storage, refresh complexity | Daily sales summary |
| Redundant column | Audit/historical data needed | Data integrity, no joins | Storage, update logic | Order historical price |
| Pre-joined table | Frequent joins | Fewer joins, simpler queries | Storage, harder updates | emp_dept table |
| Array/JSON column | Variable attributes | Flexible schema, less normalization | Query complexity, filtering | Product tags array |

*Denormalization adds complexity. Only do it when measurements show it's necessary and the trade-offs are worth it.*

# 4. The Data Modelling Process — Hoberman's Methodology

Steve Hoberman's 'Data Modeling Made Simple' gave us a 10-step process that actually works. Thousands of organizations use it. Is it rigid? No. You'll iterate a lot. Will it get you from 'we need a database for this vague idea' to 'we have a working system'? Yes. So let's follow it.

## Hoberman's 10-Step Data Modelling Process

### Step 1: Determine Purpose and Scope

What problem are we solving? What's in scope, what's out? Example: 'Build an e-commerce database to support product catalog, shopping cart, order processing, and basic analytics. NOT including: accounting, inventory management, or warehouse systems.'

### Step 2: Identify Existing Resources

What's already there? Existing databases, reports, spreadsheets, documentation, systems? Can you leverage them? Do reverse engineering if needed.

### Step 3: Build the Conceptual Model

High-level entities and relationships. Business language. Show to stakeholders. Get agreement.

```
-- E-commerce conceptual model (text)
CUSTOMER - places - ORDERS
ORDERS - contains - PRODUCTS
PRODUCTS - belong to - CATEGORIES
CUSTOMER - have - ADDRESSES
ORDERS - processed by - PAYMENT
```

### Step 4: Complete the Logical Model

Tables, columns, keys, normalization, relationships. Database-agnostic but precise.

```
-- E-commerce logical model (tables)
CREATE TABLE customer (
  customer_id INT PRIMARY KEY,
  email VARCHAR(100) UNIQUE,
  first_name VARCHAR(100),
  last_name VARCHAR(100),
  created_at DATE
);

CREATE TABLE address (
  address_id INT PRIMARY KEY,
  customer_id INT NOT NULL,
  street VARCHAR(200),
  city VARCHAR(50),
  state VARCHAR(2),
  zipcode VARCHAR(10),
  FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
);

CREATE TABLE product (
  product_id INT PRIMARY KEY,
  sku VARCHAR(50) UNIQUE,
  name VARCHAR(200),
  category_id INT NOT NULL,
```

```
  price DECIMAL(10, 2),
  FOREIGN KEY (category_id) REFERENCES category(category_id)
);

CREATE TABLE category (
  category_id INT PRIMARY KEY,
  category_name VARCHAR(100),
  description TEXT
);

CREATE TABLE orders (
  order_id INT PRIMARY KEY,
  customer_id INT NOT NULL,
  order_date TIMESTAMP,
  shipping_address_id INT NOT NULL,
  billing_address_id INT NOT NULL,
  order_status VARCHAR(20),
  FOREIGN KEY (customer_id) REFERENCES customer(customer_id),
  FOREIGN KEY (shipping_address_id) REFERENCES address(address_id),
  FOREIGN KEY (billing_address_id) REFERENCES address(address_id)
);

CREATE TABLE order_item (
  order_id INT NOT NULL,
  product_id INT NOT NULL,
  quantity INT,
  unit_price DECIMAL(10, 2),
  PRIMARY KEY (order_id, product_id),
  FOREIGN KEY (order_id) REFERENCES orders(order_id),
  FOREIGN KEY (product_id) REFERENCES product(product_id)
);

CREATE TABLE payment (
  payment_id INT PRIMARY KEY,
  order_id INT NOT NULL,
  payment_method VARCHAR(50),
  amount DECIMAL(10, 2),
  payment_date TIMESTAMP,
  status VARCHAR(20),
  FOREIGN KEY (order_id) REFERENCES orders(order_id)
);
```

### Step 5: Complete the Physical Model

Optimize for your specific database. Indexes, partitioning, data types. If you're using PostgreSQL, leverage JSONB and arrays. If Oracle, consider partitioning strategies.

```
-- E-commerce physical model (PostgreSQL-specific)
CREATE TABLE customer (
  customer_id BIGSERIAL PRIMARY KEY,
  email VARCHAR(100) UNIQUE NOT NULL,
  first_name VARCHAR(100),
  last_name VARCHAR(100),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
CREATE INDEX idx_customer_email ON customer(email);
CREATE INDEX idx_customer_created_at ON customer(created_at);

CREATE TABLE orders (
  order_id BIGSERIAL PRIMARY KEY,
  customer_id BIGINT NOT NULL REFERENCES customer(customer_id),
```

```
  order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  order_status VARCHAR(20) NOT NULL DEFAULT 'pending',
  total_amount DECIMAL(10, 2),
  metadata JSONB  -- For flexible future fields
);
CREATE INDEX idx_orders_customer_id ON orders(customer_id);
CREATE INDEX idx_orders_order_date ON orders(order_date);
CREATE INDEX idx_orders_status ON orders(order_status);
```

### Step 6: Resolve Enterprise Issues

If this is part of a larger system, how does it integrate? Naming standards? Existing master data? Governance?

### Step 7: Complete the Deployment Model

Migration scripts, loading procedures, testing plans, rollback procedures.

### Step 8: Manage the Model

Maintain documentation. Track changes. Handle new requirements. The model is living.

### Step 9: Assess Model Quality

Use Hoberman's Data Model Scorecard (see below). Are all data quality rules in place?

### Step 10: Market the Model

Make it easy to find and use. Good documentation. Training. Glossary. Make data discoverable so people use it.

## Hoberman's Data Model Scorecard

10 scoring categories to evaluate model quality. Max 10 points each.

| Category | Max Points | What It Measures | Questions to Ask |
|---|---|---|---|
| Completeness | 10 | All required entities/attributes present | Are all data requirements captured? |
| Normalization | 10 | Proper normal form for use case | Are there update anomalies? Dependencies correct? |
| Naming Convention | 10 | Consistent, meaningful names | Are names clear? Easy to understand? |
| Integrability | 10 | Can integrate with other systems | Can we connect to master data? Standards followed? |
| Implementation Feasibility | 10 | Can be built with available tech | Can it run on our database? Performance OK? |
| Data Quality | 10 | Rules prevent bad data entry | Are there constraints? Defaults? Null rules? |
| Documentation | 10 | Clear, accessible, up-to-date | Is there a data dictionary? Business rules documented? |
| Flexibility | 10 | Can handle future requirements | Room to grow? Extensible? |
| Efficiency | 10 | Performance optimized | Indexes in place? Denormalization where needed? |
| Visibility | 10 | Stakeholders understand/accept | Does business side understand? Buy-in? |

A perfect model scores 100. Most real models score 70-85; there are always trade-offs.

## Conceptual vs Logical vs Physical — Detailed Comparison

Here's how the same business scenario looks at each level.

### Conceptual Level (Business View)

```
A Customer can place multiple Orders.
Each Order contains one or more Items (Products).
Each Order is assigned a Status (pending, processing, shipped, delivered).
A Customer has one or more Addresses.
```

### Logical Level (Database Designer View)

```
CUSTOMER (customer_id, email, first_name, last_name, created_date)
ORDER (order_id, customer_id FK, order_date, status)
PRODUCT (product_id, sku, name, price, category_id FK)
ORDER_ITEM (order_id FK, product_id FK, quantity, unit_price)
ADDRESS (address_id, customer_id FK, street, city, state, zip)

Relationships:
  CUSTOMER (1) ---< (M) ORDER
  ORDER (1) ---< (M) ORDER_ITEM
  PRODUCT (1) ---< (M) ORDER_ITEM
  CUSTOMER (1) ---< (M) ADDRESS
```

### Physical Level (Implementation View)

```
-- PostgreSQL DDL
CREATE TABLE customer (
  customer_id BIGSERIAL PRIMARY KEY,
  email VARCHAR(255) UNIQUE NOT NULL,
  first_name VARCHAR(100) NOT NULL,
  last_name VARCHAR(100) NOT NULL,
  created_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
CREATE INDEX idx_customer_email ON customer(email);

CREATE TABLE product (
  product_id BIGSERIAL PRIMARY KEY,
  sku VARCHAR(50) UNIQUE NOT NULL,
  name VARCHAR(255) NOT NULL,
  price NUMERIC(12, 2) NOT NULL,
  category_id BIGINT NOT NULL REFERENCES category(category_id),
  created_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
CREATE INDEX idx_product_sku ON product(sku);
CREATE INDEX idx_product_category ON product(category_id);

CREATE TABLE orders (
  order_id BIGSERIAL PRIMARY KEY,
  customer_id BIGINT NOT NULL REFERENCES customer(customer_id),
  order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  status VARCHAR(20) NOT NULL DEFAULT 'pending',
  CHECK (status IN ('pending', 'processing', 'shipped', 'delivered'))
);
CREATE INDEX idx_orders_customer ON orders(customer_id);
CREATE INDEX idx_orders_status ON orders(status);
```

## Requirements Gathering

The foundation of good data modelling is understanding what the business actually needs. Ambiguity here leads to wrong models later. Take time here.

### Interview Stakeholders

Talk to the people who will use the system. Sales asks 'Can I see which customers bought what?' Marketing asks 'Can I segment by purchase history?' Finance asks 'Can I report on revenue by product category and region?'

### Analyze Existing Reports and Dashboards

If a report exists showing customers, orders, and total spend, those are entities/attributes your model needs. Reverse engineer from the output.

### Understand Business Rules

Can a customer have multiple addresses? Can an order have zero items? Can a product have no category? These are constraints your model must enforce.

### Document Non-Functional Requirements

How many customers? How many orders per day? What's the expected response time for queries? This affects physical design decisions.

## Iterative Modelling

You won't get the model right the first time. That's OK. Present your draft to stakeholders, get feedback, refine. Show a query that doesn't work ('I can't report on regional sales') and adjust the model to support it. This iterative approach often reveals hidden requirements.

# 5. Naming Conventions, Standards & Data Dictionary

A database with bad naming is like a house with no street signs. People get lost. They create duplicate tables because they don't know where the existing one is. They misspell column names. They use inconsistent types. Naming conventions seem boring, but they're the difference between a database six people understand and a database nobody can use.

## Why Naming Conventions Matter

Six months from now, someone new joins the team. They ask, 'Is it customer_id or cust_id? Is it customer_name or customer_full_name? Where's the email—in customer or in a separate contact table?' If you answer 'I dunno, go dig through the schema,' you've failed. Good naming conventions mean the answer is obvious without asking.

## Table Naming

### Singular vs Plural

Should it be 'customer' or 'customers'? Opinions vary. The relational model suggests singular (a row represents one entity instance). SQL doesn't care. Most modern standards use singular for consistency with ORM frameworks.

```
-- SINGULAR (recommended)
CREATE TABLE customer (...);
CREATE TABLE product (...);
CREATE TABLE category (...);

-- PLURAL (less common, but valid)
CREATE TABLE customers (...);
CREATE TABLE products (...);
```

### Prefixes and Suffixes

Some companies prefix table names: 'tbl_customer' (tbl = table). Others suffix with '_t': 'customer_t'. This seems helpful at first but adds clutter. Modern databases with schemas (postgres, sql server) don't need it.

### Abbreviations

Avoid abbreviations unless standardized (ID, SSN). 'cust' vs 'customer'? Full names are clearer. But 'id' for identifier is universal.

### Underscores vs Camel Case

snake_case is easier to read and more SQL-friendly. camelCase is common in programming. Pick one and stick to it. Most SQL shops use snake_case.

```
-- snake_case (recommended for SQL)
CREATE TABLE customer (
  customer_id BIGINT,
  email_address VARCHAR(255),
  phone_number VARCHAR(20),
  created_date DATE
);

-- camelCase (not recommended for SQL, more Java-like)
CREATE TABLE customer (
  customerId BIGINT,
```

```
  emailAddress VARCHAR(255)
);
```

| Convention Style | Table Name | Column Names | Best For | Pros | Cons |
|---|---|---|---|---|---|
| Singular + snake_case | customer | email_address, phone_number | SQL, PostgreSQL, dbt | Clear, readable, standard | Longer names |
| Plural + snake_case | customers | email_address, phone_number | Some legacy systems | Reads naturally | Non-standard |
| Singular + camelCase | customer | emailAddress, phoneNumber | Java/ORM apps | Compact | Hard to read in SQL |
| Prefixed | tbl_customer | col_email | Very old systems | Explicit type info | Verbose, redundant |

# Column Naming

## Primary and Foreign Keys

Consistent key naming makes relationships obvious. Common patterns:

- PK: 'id' (if only one) or 'table_name_id' (if many tables)
- FK: 'referenced_table_name_id' or 'fk_referenced_table'
- Example: customer.customer_id, order.customer_id (FK to customer)

```
-- Clear PK/FK naming
CREATE TABLE customer (
  customer_id BIGINT PRIMARY KEY,
  name VARCHAR(100)
);

CREATE TABLE orders (
  order_id BIGINT PRIMARY KEY,
  customer_id BIGINT NOT NULL,  -- FK, obvious by name
  order_date DATE,
  FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
);
```

## Boolean Columns

Name boolean columns to read naturally as a question. 'is_active', 'has_payment', 'requires_approval', not just 'active' or 'approval'.

```
-- Good: reads as yes/no question
CREATE TABLE customer (
  customer_id BIGINT,
  is_active BOOLEAN DEFAULT TRUE,
  is_vip BOOLEAN DEFAULT FALSE,
  has_purchased BOOLEAN,
  requires_verification BOOLEAN
);
```

## Date and Timestamp Columns

Name clearly to distinguish between dates, times, and timestamps. Add suffixes like '_date', '_time', '_at'.

```
CREATE TABLE orders (
  order_id BIGINT,
  order_date DATE,                    -- Just the date
  created_at TIMESTAMP,               -- Full timestamp
  shipped_at TIMESTAMP,
```

```
  expected_delivery_date DATE,
  ship_time TIME                        -- Just time of day
);
```

## Domains and Standard Attributes

A domain is a standard definition: 'An email always has this type and these constraints.' Define domains and reuse them.

| Domain Name | SQL Type | Constraints | Example | Use Case |
|---|---|---|---|---|
| email | VARCHAR(255) | UNIQUE, format check | user@example.com | Email contacts |
| phone | VARCHAR(20) | Format check optional | 555-1234 or +1-555-1234 | Phone numbers |
| money | DECIMAL(10,2) | NOT NULL usually | 99.99 | Prices, amounts |
| percentage | DECIMAL(5,2) | CHECK >= 0 AND <= 100 | 15.50 | Tax rate, discount |
| url | VARCHAR(2000) | URL format | https://example.com | Web links |
| ssn | VARCHAR(11) | Format pattern, private | 123-45-6789 | Social security |
| ipv4_address | VARCHAR(15) | IP format | 192.168.1.1 | Network address |
| uuid | UUID or VARCHAR(36) | UNIQUE | 550e8400-e29b-41d4-a716-446655440000 | Distributed IDs |
| country_code | VARCHAR(2) | ISO 3166-1 alpha-2 | US, UK, DE | Country |
| gender | VARCHAR(1) | CHECK IN ('M','F','O') | M, F, O | Gender |
| status | VARCHAR(20) | CHECK IN (list) | active, inactive, pending | State tracking |
| year | INT | CHECK >= 1900 AND <= current_year+1 | 2023 | Calendar year |
| latitude | DECIMAL(10,8) | -90 to 90 | 37.7749 | Geographic coords |
| currency_code | VARCHAR(3) | ISO 4217 | USD, EUR, GBP | Currency |

## The Enterprise Data Dictionary

A data dictionary is the system of record for all table and column definitions. Every enterprise should have one. It documents the who, what, where, when, and why of your data.

### What a Data Dictionary Contains

- Business Name: 'Customer Email', not just 'cust_email'
- Technical Name: 'customer.email'
- Data Type: 'VARCHAR(255)'
- Length/Precision: For all string and numeric types
- NULL Allowed?: Yes/No
- Primary/Foreign Key?: Yes, and which table referenced
- Default Value: If any
- Business Definition: What this means to the business
- Data Owner: Who maintains this data
- Sensitivity/Classification: Public, internal, confidential, PII
- Last Updated: Date of last change
- Examples: Sample values
- Derived/Calculated?: How computed, if applicable
- Validation Rules: Checks and constraints

### Example Data Dictionary Entries

```
TABLE: customer
DESCRIPTION: Represents individual customers and accounts
DATA OWNER: Sales team

  COLUMN: customer_id
  BUSINESS NAME: Customer ID
  TYPE: BIGINT
  NULL ALLOWED: No
  PK/FK: Primary Key
  DEFINITION: Unique system-generated identifier for each customer
  EXAMPLES: 1001, 1002, 5000

  COLUMN: email
  BUSINESS NAME: Email Address
  TYPE: VARCHAR(255)
  NULL ALLOWED: No
  PK/FK: Unique Index
  VALIDATION RULES: Must match email pattern, unique across table
  DEFINITION: Primary email for customer contact
  SENSITIVITY: Internal (not public, but shared with marketing)
  EXAMPLES: john@example.com, mary.smith@corp.com

  COLUMN: created_at
  BUSINESS NAME: Account Creation Date
  TYPE: TIMESTAMP
  NULL ALLOWED: No
  DEFAULT: CURRENT_TIMESTAMP
  DEFINITION: Date and time customer account was created
  EXAMPLES: 2023-01-15 10:30:45, 2023-06-22 14:22:11
```

## Metadata: Technical, Business, and Operational

Metadata is data about data. There are three types to track.

#### Technical Metadata

Table names, column names, data types, constraints, indexes, storage location, replication settings. This is what the data dictionary captures.

#### Business Metadata

Definitions, ownership, how to use, who has permission, data quality rules. 'Customer' means a person or company that has purchased at least once. Owned by the Sales team.

#### Operational Metadata

When data was loaded, how many rows, refresh frequency, last successful load, any errors. 'Customer table refreshed at 6 AM daily. Last run: 2023-12-15 06:15:22, 1.2M rows loaded, 0 errors.'

> *Modern data catalogs (Alation, Collibra, Dataedo) automatically track technical and operational metadata. You add business metadata manually. Invest in a data catalog if your organization has more than ~100 tables.*

## Abbreviation Standards

If you must abbreviate (in composite key names, for example), have a standard list so everyone uses 'cust_id' not 'c_id' or 'customer_identifier'.

| Abbreviation | Meaning | Example Usage | Notes |
|---|---|---|---|
| id | identifier | customer_id, product_id | Universal, always use 'id' not 'identifier' |
| pk | primary key | pk_customer | In constraint names |
| fk | foreign key | fk_customer_in_order | In constraint names |
| src | source | src_system, source_table | For ETL/integration tables |
| tgt | target | tgt_customer | For ETL destination |
| dim | dimension | dim_customer (in warehouses) | Dimensional modelling term |
| fact | fact table | fact_sales (in warehouses) | Dimensional modelling term |
| tmp | temporary | tmp_staging_orders | Not permanent tables |
| hist | history | customer_hist | Historical/archived data |
| cnt | count | order_cnt, transaction_cnt | For aggregate/summary columns |
| amt | amount | order_amt, discount_amt | Financial amounts |
| pct | percent | discount_pct, tax_pct | Percentage fields |
| dt | date | order_dt, created_dt | Use sparingly; 'date' is clearer |
| ts | timestamp | created_ts, updated_ts | Use sparingly; 'created_at' is clearer |

## Complete Naming Standard Document Template

```
NAMING STANDARDS FOR [COMPANY] DATA MODELS
Version: 1.0
Last Updated: 2023-12-15
Owner: Data Architecture Team

1. TABLE NAMING
   - Use singular nouns (customer, not customers)
   - Use snake_case (customer_order, not CustomerOrder)
   - No prefixes (tbl_, tmp_) unless necessary for temporary/staging
   - Examples: customer, product, order, payment

2. COLUMN NAMING
   - Use snake_case consistently
   - Primary key: '{table_name}_id' (customer_id, product_id)
   - Foreign key: '{referenced_table}_id' (customer_id in order table)
   - Boolean columns: prefix with 'is_' or 'has_' (is_active, has_paid)
   - Date columns: suffix with '_date' (order_date, birth_date)
   - Timestamp columns: suffix with '_at' (created_at, updated_at)
   - Amount columns: suffix with '_amt' or '_amount' (order_amt, discount_amt)
   - Percent columns: suffix with '_pct' or '_percent' (tax_pct)

3. CONSTRAINTS
   - Primary Key: 'pk_{table_name}' (pk_customer)
   - Foreign Key: 'fk_{table}_{referenced_table}' (fk_order_customer)
   - Unique: 'uk_{table}_{columns}' (uk_customer_email)
   - Check: 'ck_{table}_{condition}' (ck_order_status)

4. INDEXES
   - Standard: 'idx_{table}_{columns}' (idx_customer_email)
   - Unique: 'uidx_{table}_{columns}' (uidx_customer_email)

5. ABBREVIATIONS
   - Use full words when possible (email, not eml)
   - Approved abbreviations: id, pk, fk, src, tgt, dim, fact, tmp, amt, pct
   - No single-letter abbreviations (c_id is not acceptable)

6. SPECIAL CASES
   - Staging tables: prefix with 'stg_' (stg_customer_load)
   - Historical/archive: suffix with '_hist' (customer_hist)
   - Temporary: prefix with 'tmp_' (tmp_calculation)
```

```
7. DATA DICTIONARY MAINTENANCE
   - Update entry within 24 hours of schema change
   - Include business definition for all tables
   - Mark sensitive/PII data clearly
   - Document data owner for each table
```

## Summary: Standards = Clarity = Success

Good naming conventions and standards feel like bureaucracy at first. But six months later when someone new joins and can understand your database in minutes instead of days, the value becomes clear. Standards scale; heroics don't.

### Conclusion: From Theory to Practice

You've now learned the foundations of data modelling: conceptual thinking, entity-relationship diagrams, normalization rules, the process for building models, and standards for maintaining them. These aren't just academic exercises. They're the difference between databases that work for five years and systems that collapse under complexity.

The real skill in data modelling is asking the right questions: What is this entity really? Does a customer always need an address? Can we change prices retroactively, or do we need history? Can a person belong to two departments? These questions, asked early, save months of rework later.

In Part II, we'll take these foundations and build real-world models for actual business domains: e-commerce, healthcare, finance, and more. You'll see how these principles apply when stakes are high and requirements are complex. Keep these foundational concepts close as you move forward.

> *Next: Part II - Real-World Data Models. We'll see entity-relationship, normalization, and standards applied to actual business scenarios where decisions matter.*

# CHAPTER 6: The Dimensional Bus Architecture

Welcome to the Dimensional Bus—Ralph Kimball's answer to the question: 'How do I build a data warehouse that doesn't make me want to quit?' Instead of designing one massive 3NF monstrosity, you build process by process, reusing dimensions as you go. It's like IKEA furniture for data: modular, reproducible, and somehow it actually works. Let's figure out why this is such a big deal.

## Why Dimensional Modelling? The 3NF Limitation

Relational databases are optimized for transactional processing, not analytics. Third Normal Form (3NF) — the gold standard for transactional design — breaks data into many small tables to eliminate redundancy. A transaction query joins dozens of tables. An analytics query doing the same becomes glacially slow.

> *In a 3NF sales database: 50 joins to answer 'What was revenue by customer and product last month?' With dimensional modelling: 3 joins (fact to dim_customer, dim_product, dim_date).*

## The Dimensional Bus: Kimball's Vision

Here's the beautiful idea: instead of trying to model your entire company at once (spoiler: you'll fail), you tackle one process at a time. Sales? Build a star schema around it. Returns? Another star. Inventory? Another. The trick is that they DON'T live in isolation. You reuse the same dimensions everywhere. Same dim_customer in sales, returns, AND support. Same dim_date everywhere. This means one query can slice across the entire enterprise without a mess of custom views and duct tape.

## The 4-Step Dimensional Design Process

Every dimensional model follows Kimball's four-step process. Let's walk through it with a retail sales example: a grocery store chain analyzing point-of-sale transactions.

### Step 1: Select the Business Process

A business process is a repeatable event that generates data. Examples: sales, returns, inventory movement, employee hours, website clicks. We choose which process to model first based on business priority and data availability. For our grocery store, we start with the most critical: point-of-sale transactions.

**Example: We analyze our retail operations and identify these business processes:**

- POS Transaction (daily sales transactions)
- Product Returns (when customers return items)
- Inventory Movement (stock checks and transfers)
- Staff Scheduling (labor planning and hours)
- Promotion Execution (promotional pricing and effectiveness)

## Step 2: Declare the Grain

The grain is the atomic level of the fact table — the lowest level of detail. Declaring grain first ensures all dimensions and facts are at the same level. For our POS transactions, the grain is: ONE ROW PER SKU PER TRANSACTION PER REGISTER.

**Why declare grain first? Because everything else depends on it:**

- Which dimensions do we need? (All that describe one row at this grain)
- Which facts are valid? (All that are numeric measurements at this grain)
- How do we handle Type 2 slowly-changing dimensions? (Do we store version keys?)

*If you don't declare grain upfront, dimensions mysteriously explode and facts become non-additive. A common mistake: declaring grain as 'per transaction' but then adding product_weight as a dimension (which is per product, not per transaction).*

## Step 3: Identify the Dimensions

Dimensions are the descriptive attributes that surround a fact. They answer: WHO, WHAT, WHEN, WHERE, WHY, HOW. For each dimension, ask: 'Does every row in my fact have exactly one value for this dimension at the declared grain?'

**For our POS transaction fact (one row per SKU per transaction):**

| Dimension | Sample Attributes | Rows in Dim | Why It's a Dim |
|---|---|---|---|
| dim_date | year, month, day, day_of_week, is_holiday | ~7,300 | Every transaction has exactly one date |
| dim_time | hour, minute, second, time_period (morning/afternoon/night) | ~86,400 | Every transaction has a timestamp |
| dim_product | sku, product_name, category, brand, price, weight | ~50,000 | Every item scanned has one product |
| dim_store | store_id, city, state, store_manager, store_size | ~400 | Every transaction happens at one store |
| dim_register | register_id, register_type, is_self_checkout | ~2,000 | Every transaction occurs at one register |
| dim_customer | customer_id, name, loyalty_tier, zip_code | ~1,000,000 | Transaction is by/for one customer (or anonymous) |

## Step 4: Identify the Facts

Facts are the numeric measurements of the business process. At the declared grain, every measurement is additive across certain dimensions. For a POS transaction line item, the facts are the quantities and amounts.

**Facts for our POS transaction (one row per SKU per transaction):**

| Fact | Data Type | Additivity | Business Meaning |
|---|---|---|---|
| quantity_sold | DECIMAL(10,2) | Fully additive | Units of SKU sold in this transaction |
| unit_price | DECIMAL(10,4) | Non-additive | Price per unit at time of sale |
| extended_amount | DECIMAL(12,2) | Fully additive | quantity × unit_price before tax/discount |
| discount_amount | DECIMAL(12,2) | Fully additive | Promotional or loyalty discount applied |
| tax_amount | DECIMAL(12,2) | Fully additive | Sales tax on this line item |
| net_amount | DECIMAL(12,2) | Fully additive | Final amount customer paid for this line |

# The Enterprise Bus Matrix

So you built a star schema for sales. Great. Now someone says 'we need returns.' You don't start from scratch—you ask: 'Can we reuse dim_customer? dim_product? dim_date?' The Enterprise Bus Matrix is basically a spreadsheet that forces this conversation. It's your roadmap. It prevents the chaos of everyone building their own customer dimension with different definitions.

## What is the Bus Matrix?

A matrix with rows = business processes and columns = conformed dimensions. An X marks when a dimension is shared. This simple tool forces conversations: 'Do we use the same dim_customer in sales, returns, and support, or different versions?' Conforming now prevents data chaos later.

## A Complete Retail Bus Matrix

**Rows are 8 business processes; columns are 10 conformed dimensions:**

| Process | Date | Time | Product | Store | Customer | Register | Promo | Channel | Employee |
|---|---|---|---|---|---|---|---|---|---|
| POS Transaction | X | X | X | X | X | X | X | | X |
| Product Return | X | X | X | X | X | X | | | X |
| Inventory Count | X | | X | X | | | | | |
| Promotion Planning | X | | X | X | | | X | | X |
| Staff Schedule | X | X | | | | | | | X |
| Price Change | X | | X | X | | | | | |
| Online Order | X | X | X | | X | | | X | |
| Customer Loyalty | X | | | | X | | X | | |

## How to Read and Use the Bus Matrix

- VERTICAL read: A dimension's column shows which processes share it. dim_date appears in ALL processes (time is universal).
- HORIZONTAL read: A process's row shows its dimensions. POS Transaction has 7 dimensions; Staff Schedule has only 3.
- PLANNING: Which dimensions should we build as conformed from day 1? (Rows 1-3). Which can wait? (Rows 6-8).
- CONFORMANCE: Before releasing a new process, ensure its dimensions match existing ones. If 'Product' in Promotion Planning differs from 'Product' in POS, you've created silos.

# Conformed Dimensions: The Connective Tissue

Here's the problem: you've got sales with their version of 'customer,' returns with their version, and support with yet another. They all think they're right. You pull them together and realize the customer IDs don't match. A conformed dimension solves this: ONE dim_customer, used everywhere. Same customer key, same attributes, same slowly-changing-dimension logic. Cross-process queries? Now possible.

## Why Conformance Matters

Without conformance, you're stuck. The sales team says 'customer 12345' but the returns team says 'cust_12345'—different system, different format. You can't join them. You pull your hair out writing custom mappings. With conformed dimensions, both teams reference the SAME dim_customer. Same surrogate key. One query answers: 'Compare revenue and returns for customer X.' Done.

> *Conformance doesn't mean identical source systems. You can pull customer data from CRM (POS), ERP (returns), and support ticketing system. But you reconcile them into ONE dim_customer. This is the ETL layer's job.*

## Creating Conformed Dimensions

**Process for building a conformed dim_customer across sales, returns, support:**

- AUDIT: Examine customer data from all three source systems. What fields overlap? What's unique to each?
- DESIGN: Create a master dim_customer with all necessary attributes. Add lineage columns (source_system) if needed.
- RECONCILE: Define business rules for duplicates (same person, different email). This is your MDM (Master Data Management) logic.
- BUILD: ETL loads all sources into one dim. Apply SCD Type 2 to track changes.
- VALIDATE: Before releasing to analysts, verify that customer_key = X joins correctly in all three facts.

## Example: dim_customer Across Three Processes

```
-- Single conformed dim_customer used by sales, returns, and support
CREATE TABLE dim_customer (
    customer_key INT PRIMARY KEY,          -- Surrogate key (same across all facts)
    customer_natural_id VARCHAR(50),       -- Natural key (email or assigned ID)
    customer_name VARCHAR(100),
    city VARCHAR(50),
    state CHAR(2),
    zip_code VARCHAR(10),
    loyalty_tier VARCHAR(20),              -- Gold, Silver, Bronze
    lifetime_revenue DECIMAL(12,2),        -- Updated daily from all three processes
    customer_since_date DATE,
    is_current BOOLEAN,
    effective_date DATE,
    expiry_date DATE,
    source_system VARCHAR(20),             -- CRM, ERP, Support system
    dw_load_date TIMESTAMP
);

-- A single customer_key = 12345 appears in:
-- fact_pos_transaction (when they buy),
-- fact_product_return (when they return),
-- fact_support_ticket (when they contact support).
-- Because they use the same customer_key, one query joins all three.
```

# Conformed Facts: Same Metric, Same Math

You know that moment when finance says revenue is $1M but sales says $1.2M? That's because 'revenue' has six different definitions depending on who you ask. A conformed fact is when you go to war and force everyone to agree: here's how we calculate revenue, and we use that definition EVERYWHERE. Finance calculates it one way, sales calculates it the same way, and your dashboard actually tells the truth.

## Example: Revenue Conformed Across Sales and Finance

**How to ensure 'revenue' means the same thing everywhere:**

- DEFINITION: Revenue = (quantity × unit_price) - discounts + taxes. In USD.
- SOURCE: Both facts pull unit_price from the same dim_product.
- TIMING: Both recognize revenue on the same transaction date.
- CURRENCY: Non-USD sales are converted to USD using the same exchange rate table.
- ADJUSTMENTS: Refunds and chargebacks are recorded as negative amounts in the same fact table (not separate rows).

# Bottom-Up (Kimball) vs Top-Down (Inmon)

There's been a holy war in data warehousing for 30 years. Kimball says: start small, build star schemas by business process, move fast, ship value to analysts. Inmon says: think big, build a 3NF data model first, normalize everything, think about query optimization later. We're team Kimball in this book, but you should understand both so when someone at a conference mentions 'Bill Inmon,' you don't look lost.

## Kimball vs Inmon: Head-to-Head

| Criterion | Kimball (Dimensional) | Inmon (3NF) |
|---|---|---|
| Starting Point | Individual business processes | Enterprise-wide data model |
| Design Method | Bottom-up (process by process) | Top-down (all at once) |
| Deliverable Timeline | First dimensional model in 3-6 months | 3NF design in 1-2 years |
| Data Structure | Denormalized star schema | Fully normalized 3NF |
| Number of Tables | Few (1 fact + 6-10 dims per process) | Many (50-200 tables) |
| Query Joins | 3-5 joins for typical query | 20-50 joins needed |
| Query Speed | Fast (few joins, aggregates pre-computed) | Slow without heavy indexing |
| ETL Complexity | Moderate (conformation logic) | High (massive integration) |
| Storage Size | Larger (denormalization redundancy) | Smaller (normalization) |
| BI Tool Friendliness | Excellent (tools assume star) | Fair (requires views/materialized views) |
| Metadata/Lineage | Conformed dims create natural lineage | Lineage implicit in 3NF design |
| Scalability | Excellent (dimensions are reusable) | Moderate (monolithic 3NF model) |

## When Each Approach Wins

- KIMBALL: Multi-process enterprise, stakeholders demand fast time-to-value, business model changes frequently, BI tools are central, storage is cheap, analysts need self-service querying.
- INMON: Single-process or tightly integrated processes, data integration is critical and complex, strict data governance (e.g., finance, healthcare), storage cost is critical (rarely true now), heavy ETL/batch processing is normal.

### The Modern Hybrid Approach

Here's what actually wins: build a normalized 3NF layer (the ODS) to integrate all your messy source systems, then layer Kimball dimensional models on top for BI. You get the best of both: a clean, auditable source of truth AND fast, dimensional queries for analysts. Cloud warehouses made this cheap—storage and compute are decoupled, so the redundancy doesn't kill your budget anymore.

## SQL: The Bus Matrix as Metadata

In real life, you'll maintain the Bus Matrix in a metadata table. This isn't just documentation—it's queryable. You can ask: 'Which dimensions are actually being reused?' 'Which business process have we modeled yet?' 'Is there some dimension living in only one process that should be conformed?' The Bus Matrix becomes your roadmap, living and breathing in your warehouse.

```
-- Bus Matrix as a metadata table
CREATE TABLE dw_bus_matrix (
    business_process VARCHAR(50),
    conformed_dimension VARCHAR(50),
    is_required BOOLEAN,
    implementation_status VARCHAR(20),  -- Planned, In Progress, Complete
    notes VARCHAR(500)
);

-- Insert all 8 × 10 combinations where applicable:
INSERT INTO dw_bus_matrix VALUES
('POS Transaction', 'dim_date', TRUE, 'Complete', 'Every sale has a date'),
('POS Transaction', 'dim_product', TRUE, 'Complete', 'Every line has a product'),
('Product Return', 'dim_date', TRUE, 'Complete', 'Return has a date'),
...

-- Query: Which dimensions are most reused?
SELECT conformed_dimension, COUNT(*) as process_count
FROM dw_bus_matrix
WHERE is_required = TRUE
GROUP BY conformed_dimension
ORDER BY process_count DESC;
-- Output: dim_date appears in 8 processes, dim_product in 6, dim_customer in 5, etc.
```

# CHAPTER 7: Fact Tables — The Complete Guide

Facts are the numbers: revenue, quantity, counts. They're the heart of the star schema. If you get them right, you'll build amazing dashboards. If you get them wrong—storing them at the wrong grain, mixing additivity types, storing text instead of numbers—you'll spend three months answering 'wait, can we really sum this?' Let's get it right.

## What Makes a Good Fact?

Three things: (1) It's a number—not a flag, not a date, not a name. (2) It can be summed (at least across SOME dimensions). (3) It lives at the declared grain. If your grain is 'per transaction,' don't store 'average customer spend'—that's per customer, not per transaction. Violate these rules and you'll have a fact table that lies.

> *Text, dates, and flags are NOT facts. They are dimension attributes. If you store 'product_name' in a fact, you've denormalized into the fact table — a mistake.*

## Additivity: The Dimensionality of Facts

This is where most people mess up. You can sum revenue across all dimensions and it means something. You can sum account balance across accounts (total balance) but NOT across time (that's nonsense). And you can't sum prices at all. Understanding additivity is the difference between correct analytics and dashboards that lie to the CEO at 3am.

### Fully Additive Facts

The easy ones. Revenue, quantity, transaction count. Sum them across every dimension and the result makes sense: sum(revenue) across all customers, all stores, all days = total company revenue. No tricks. This is what you want in life.

The tricky ones. Account balance, inventory on hand, employee headcount. You CAN sum across customers '(what's the total inventory?') or stores ('what's the total balance?'), but NOT across time. If you sum Dec 1 balance + Dec 2 balance, you get garbage. This trips everyone up on their first data warehouse.

### Non-Additive Facts

Don't store these as facts. Unit price, percentages, ratios, temperatures. You can't add $5 + $10 + $8 and call it 'total price'—that's nonsense. Either (1) store the numerator and denominator separately and let analysts recalculate the ratio themselves, or (2) calculate the ratio at query time. Just don't clutter your fact table with computed garbage.

### Common Measures Classified by Additivity

| Measure | Type | Fully Additive? | How to Query |
|---|---|---|---|
| Revenue | Monetary | Yes | SUM(revenue) |
| Quantity Sold | Count | Yes | SUM(quantity_sold) |
| Unit Price | Price | No | WEIGHTED_AVG(unit_price, quantity_sold) |
| Discount % | Percent | No | SUM(discount_amount) / SUM(extended_amount) |
| Account Balance | Balance | Semi | Can SUM across accounts, NOT across time |

| Measure | Type | Fully Additive? | How to Query |
|---|---|---|---|
| Inventory on Hand | Inventory | Semi | Can SUM across locations, NOT time |
| Employee Count | Headcount | Semi | Can SUM across departments, NOT time |
| Temperature | Measurement | No | AVERAGE(temperature) |
| Customer Satisfaction | Score | No | AVERAGE(satisfaction_score) |
| Transaction Count | Count | Yes | COUNT(*) or SUM(transaction_count) |

## SQL: Semi-Additive Balance Queries

Here's where most SQL goes wrong. If you want month-end balance, you can't SUM. You have to grab the LAST balance of the month. Let me show you both the wrong way (the way most people write it) and the right way:

```
-- WRONG: Sum of all daily balances (nonsense)
SELECT account_id, SUM(balance_amount)
FROM fact_daily_balance
WHERE year = 2024 AND month = 1
GROUP BY account_id;

-- RIGHT: The balance on the last day of the month
SELECT account_id, balance_amount
FROM fact_daily_balance
WHERE year = 2024 AND month = 1 AND day = 31
GROUP BY account_id;

-- Or using window functions to get the last row per account per month:
WITH monthly_balance AS (
    SELECT
        account_id,
        EXTRACT(YEAR_MONTH FROM date_key) as year_month,
        balance_amount,
        ROW_NUMBER() OVER (PARTITION BY account_id, EXTRACT(YEAR_MONTH FROM date_key)
                           ORDER BY date_key DESC) as rn
    FROM fact_daily_balance
)
SELECT account_id, year_month, balance_amount
FROM monthly_balance
WHERE rn = 1;
```

# Four Types of Fact Tables

## Type 1: Transaction Fact Tables

One row per event. A customer buys something, you get one row. They buy again, another row. These tables grow FAST and can get huge, but they're the most detailed and most flexible for ad-hoc queries. Most star schemas you'll build are transaction fact tables.

### Example: Retail POS Transactions

Every time someone scans a SKU at checkout, you get one row. A grocery store? That's 100,000+ rows a day, easily. But it's perfect for BI: you can slice by product, by store, by time, by customer. Everything is traceable back to the actual transaction.

```
CREATE TABLE fact_pos_transaction (
    transaction_key BIGINT PRIMARY KEY,
    -- Foreign Keys to Conformed Dimensions
    date_key INT NOT NULL,                  -- FK to dim_date
    time_key INT NOT NULL,                  -- FK to dim_time (HHMM)
    product_key INT NOT NULL,               -- FK to dim_product
    store_key INT NOT NULL,                 -- FK to dim_store
    register_key INT NOT NULL,              -- FK to dim_register
    customer_key INT NOT NULL,              -- FK to dim_customer (may be -1 for anonymous)
    employee_key INT NOT NULL,              -- FK to dim_employee (cashier)
    promotion_key INT NOT NULL,             -- FK to dim_promotion (may be 0 for no promo)
    -- Degenerate Dimensions (no separate table, data just here)
    transaction_number VARCHAR(20),         -- e.g., "RTL-20240314-00001234"
    transaction_line_number INT,            -- Line item number within transaction
    -- Fully Additive Facts
    quantity_sold DECIMAL(10,3),
    extended_amount DECIMAL(12,2),          -- qty × unit_price
    discount_amount DECIMAL(12,2),
    tax_amount DECIMAL(12,2),
    net_amount DECIMAL(12,2),
    -- Semi-Additive (if we track inventory at point of sale)
    unit_cost DECIMAL(10,4),
    cost_amount DECIMAL(12,2),
    -- Metadata
    dw_load_date TIMESTAMP,
    source_system VARCHAR(20)
);

-- Create indexes for common queries
CREATE INDEX idx_transaction_datekey ON fact_pos_transaction(date_key, store_key);
CREATE INDEX idx_transaction_productkey ON fact_pos_transaction(product_key, date_key);
```

### Five Query Patterns on Transaction Facts

- Total sales by product and day: GROUP BY product_key, date_key
- Customer lifetime value: GROUP BY customer_key (sum all transactions)
- Store performance: GROUP BY store_key, date_key
- Promotion effectiveness: Compare net_amount WITH promotion_key = 0 vs > 0
- Hourly sales trend: GROUP BY date_key, time_key

```
-- Query 1: Daily sales by product
SELECT d.date, p.product_name,
       SUM(f.quantity_sold) as total_qty,
       SUM(f.net_amount) as total_revenue
FROM fact_pos_transaction f
JOIN dim_date d ON f.date_key = d.date_key
JOIN dim_product p ON f.product_key = p.product_key
WHERE d.date >= '2024-01-01'
GROUP BY d.date, p.product_name
ORDER BY d.date, total_revenue DESC;

-- Query 2: Customer lifetime value
SELECT c.customer_name, c.loyalty_tier,
       COUNT(*) as transaction_count,
       SUM(f.quantity_sold) as total_items_purchased,
       SUM(f.net_amount) as lifetime_revenue
FROM fact_pos_transaction f
JOIN dim_customer c ON f.customer_key = c.customer_key
WHERE c.is_current = TRUE
GROUP BY c.customer_name, c.loyalty_tier
HAVING SUM(f.net_amount) > 1000
ORDER BY lifetime_revenue DESC;

-- Query 3: Promotion effectiveness
SELECT d.date,
       p.promotion_name,
       SUM(CASE WHEN f.promotion_key = 0 THEN f.net_amount ELSE 0 END) as regular_sales,
       SUM(CASE WHEN f.promotion_key > 0 THEN f.net_amount ELSE 0 END) as promo_sales,
       (SUM(CASE WHEN f.promotion_key > 0 THEN f.quantity_sold ELSE 0 END) /
        SUM(CASE WHEN f.promotion_key = 0 THEN f.quantity_sold ELSE 0 END) - 1) * 100 as lift_pct
FROM fact_pos_transaction f
JOIN dim_date d ON f.date_key = d.date_key
JOIN dim_promotion p ON f.promotion_key = p.promotion_key
WHERE d.date BETWEEN '2024-01-01' AND '2024-01-31'
  AND f.promotion_key > 0
GROUP BY d.date, p.promotion_name;
```

## Degenerate Dimensions

See transaction_number and transaction_line_number in the fact table? Those are degenerate dimensions. They LOOK like dimension foreign keys, but there's no dim_transaction table. Why? Because a transaction number IS just a number—there's nothing to join to, no attributes to add. Store it for traceability (auditing, debugging), but don't create an entire table for it.

> *Degenerate dimensions are fine. Store them for traceability (auditing, error correction). But don't create a dim_transaction table with just a transaction_number column. That's wasteful.*

## Type 2: Periodic Snapshot Fact Tables

One row per entity per time period. If you have 10,000 customers and you take a daily snapshot, that's 10,000 rows EVERY day. Dense, predictable growth. Perfect for 'show me account balances as of month-end' or 'inventory levels by date.' You get history without needing to track change.

### Example: Monthly Account Balance Snapshot

A bank wants to know: 'What was the balance on Dec 31?' So they take a snapshot of EVERY account on that date. 10,000 accounts = 10,000 rows. Then Jan 31? Another 10,000 rows. Now you can see how balances changed month-over-month without wrestling with transaction details. Clean, simple, and it answers the question perfectly.

```
CREATE TABLE fact_account_balance_snapshot (
    -- Grain: One row per account per month
    account_key INT NOT NULL,              -- FK to dim_account
    date_key INT NOT NULL,                 -- FK to dim_date (month-end date)
    -- Foreign Keys
    customer_key INT NOT NULL,             -- FK to dim_customer
    branch_key INT NOT NULL,               -- FK to dim_branch
    account_type_key INT NOT NULL,         -- FK to dim_account_type (Checking, Savings, etc.)
    -- Facts (semi-additive: do NOT sum across time!)
    opening_balance DECIMAL(14,2),
    closing_balance DECIMAL(14,2),
    total_deposits DECIMAL(14,2),
    total_withdrawals DECIMAL(14,2),
    transaction_count INT,
    -- Calculated measures
    interest_earned DECIMAL(12,2),
    service_fees DECIMAL(10,2),
    -- Metadata
    dw_load_date TIMESTAMP
);

-- Sample data: account 12345 has one row per month
INSERT INTO fact_account_balance_snapshot VALUES
(12345, '2024-01-31', 1001, 5, 1, 50000, 52350.75, 52350.75, 5000, 10000, 42, 45.50, 5.00,
CURRENT_TIMESTAMP),
(12345, '2024-02-29', 1001, 5, 1, 52350.75, 55200.00, 55200.00, 4000, 8500, 38, 48.75, 5.00,
CURRENT_TIMESTAMP),
(12345, '2024-03-31', 1001, 5, 1, 55200.00, 58100.50, 58100.50, 3500, 7200, 41, 50.25, 5.00,
CURRENT_TIMESTAMP);
```

### Loading Pattern: Creating the Snapshot

```
-- ETL: Calculate snapshot for each account at month-end
INSERT INTO fact_account_balance_snapshot
SELECT
    a.account_key,
    d.date_key,
    a.customer_key,
    a.branch_key,
    a.account_type_key,
    -- opening_balance = previous month's closing_balance
    LAG(snapshot.closing_balance)
        OVER (PARTITION BY a.account_key ORDER BY d.date) as opening_balance,
    -- Current month-end balance (from accounts table or transaction sum)
    a.current_balance as closing_balance,
    -- Deposits and withdrawals this month (from transaction fact)
```

```
    COALESCE(SUM(CASE WHEN tr.amount > 0 THEN tr.amount ELSE 0 END), 0) as total_deposits,
    COALESCE(SUM(CASE WHEN tr.amount < 0 THEN ABS(tr.amount) ELSE 0 END), 0) as total_withdrawals,
    COUNT(DISTINCT tr.transaction_id) as transaction_count,
    a.interest_earned_this_month,
    a.service_fees_this_month,
    CURRENT_TIMESTAMP
FROM dim_account a
JOIN dim_date d ON EXTRACT(MONTH FROM d.date) = EXTRACT(MONTH FROM CURRENT_DATE)
                AND EXTRACT(YEAR FROM d.date) = EXTRACT(YEAR FROM CURRENT_DATE)
                AND d.day_of_month = LAST_DAY(d.date)  -- Only month-end dates
LEFT JOIN transactions tr ON tr.account_id = a.account_natural_id
                         AND EXTRACT(MONTH FROM tr.transaction_date) = EXTRACT(MONTH FROM
CURRENT_DATE)
WHERE a.is_active = TRUE
GROUP BY a.account_key, d.date_key, a.customer_key, a.branch_key, a.account_type_key;
```

## Query Patterns: Snapshot Analysis

```
-- Query 1: Month-over-month balance change
SELECT c.customer_name,
       p.close_balance as prev_month_balance,
       curr.closing_balance as current_month_balance,
       (curr.closing_balance - p.closing_balance) as balance_change,
       ((curr.closing_balance - p.closing_balance) / p.closing_balance) * 100 as change_pct
FROM fact_account_balance_snapshot curr
JOIN fact_account_balance_snapshot p
    ON curr.account_key = p.account_key
    AND curr.date_key = p.date_key + 1  -- Next month
JOIN dim_customer c ON curr.customer_key = c.customer_key
WHERE EXTRACT(YEAR FROM curr.date) = 2024
ORDER BY balance_change DESC;

-- Query 2: High-balance accounts trending down (churn risk)
WITH monthly_balance AS (
    SELECT account_key, date_key, closing_balance,
           ROW_NUMBER() OVER (PARTITION BY account_key ORDER BY date_key DESC) as rn
    FROM fact_account_balance_snapshot
)
SELECT a.account_natural_id, c.customer_name,
       prev_3mo.closing_balance as balance_3mo_ago,
       current.closing_balance as current_balance,
       (current.closing_balance - prev_3mo.closing_balance) as trend
FROM monthly_balance current
LEFT JOIN monthly_balance prev_3mo ON current.account_key = prev_3mo.account_key
                                   AND prev_3mo.rn = 3
WHERE current.rn = 1
  AND current.closing_balance > 100000
  AND (current.closing_balance - prev_3mo.closing_balance) < -10000
ORDER BY trend;
```

## Type 3: Accumulating Snapshot Fact Tables

One row per entity lifecycle with multiple milestone dates. As the entity progresses through stages, the row is updated. Used for processes with stages: orders, loans, insurance claims, employee onboarding.

### Example: Order Fulfillment with 5 Milestones

An order goes: Order Placed → Payment Confirmed → Picked → Shipped → Delivered. We track all five dates in one row. As the order progresses, we UPDATE the row.

```
CREATE TABLE fact_order_accumulating_snapshot (
    -- Grain: One row per order (updated as order progresses)
    order_key INT PRIMARY KEY,
    -- Foreign Keys to Conformed Dimensions
    customer_key INT NOT NULL,
    product_key INT NOT NULL,
    store_key INT NOT NULL,
    -- Role-Playing Dates (same dim_date used multiple times, different roles)
    order_placed_date_key INT NOT NULL,
    order_placed_time_key INT NOT NULL,
    payment_confirmed_date_key INT,        -- NULL until payment confirmed
    payment_confirmed_time_key INT,
    order_picked_date_key INT,             -- NULL until item picked from warehouse
    order_picked_time_key INT,
    order_shipped_date_key INT,            -- NULL until shipped
    order_shipped_time_key INT,
    order_delivered_date_key INT,          -- NULL until delivered
    order_delivered_time_key INT,
    -- Order Attributes
    order_quantity INT,
    order_amount DECIMAL(12,2),
    -- Milestone Lag Metrics (calculated)
    days_to_payment INT,                   -- payment_date - order_date
    days_to_pick INT,
    days_to_ship INT,
    days_to_delivery INT,
    total_days_to_delivery INT,
    -- Status
    order_status VARCHAR(20),              -- Placed, Paid, Picked, Shipped, Delivered
    is_complete BOOLEAN,
    -- Metadata
    dw_load_date TIMESTAMP,
    dw_update_date TIMESTAMP
);
```

### Loading Pattern: Accumulating Updates

```
-- Sample data: One order, updated as it progresses
INSERT INTO fact_order_accumulating_snapshot
VALUES (1001, 100, 500, 10, 20240101, 0930, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
        2, 150.00, NULL, NULL, NULL, NULL, NULL, 'Placed', FALSE, NOW(), NOW());

-- ETL runs hourly: Payment gets confirmed
UPDATE fact_order_accumulating_snapshot
SET payment_confirmed_date_key = 20240101,
    payment_confirmed_time_key = 1045,
    days_to_payment = 0,
    order_status = 'Paid',
    dw_update_date = NOW()
```

```
WHERE order_key = 1001 AND payment_confirmed_date_key IS NULL;

-- ETL runs next day: Item picked
UPDATE fact_order_accumulating_snapshot
SET order_picked_date_key = 20240102,
    order_picked_time_key = 0800,
    days_to_pick = 1,
    order_status = 'Picked',
    dw_update_date = NOW()
WHERE order_key = 1001 AND order_picked_date_key IS NULL;

-- ETL runs same day: Shipped
UPDATE fact_order_accumulating_snapshot
SET order_shipped_date_key = 20240102,
    order_shipped_time_key = 1400,
    days_to_ship = 1,
    order_status = 'Shipped',
    dw_update_date = NOW()
WHERE order_key = 1001 AND order_shipped_date_key IS NULL;

-- ETL runs 3 days later: Delivered
UPDATE fact_order_accumulating_snapshot
SET order_delivered_date_key = 20240105,
    order_delivered_time_key = 1000,
    days_to_delivery = 3,
    total_days_to_delivery = 4,  -- order_date to delivery_date
    order_status = 'Delivered',
    is_complete = TRUE,
    dw_update_date = NOW()
WHERE order_key = 1001 AND order_delivered_date_key IS NULL;

-- Query: Show progression of this order
SELECT order_key, order_status,
       do.date as order_placed,
       COALESCE(dp.date, 'Not Yet') as payment_confirmed,
       COALESCE(dpick.date, 'Not Yet') as order_picked,
       COALESCE(ds.date, 'Not Yet') as order_shipped,
       COALESCE(dd.date, 'Not Yet') as order_delivered,
       total_days_to_delivery
FROM fact_order_accumulating_snapshot f
LEFT JOIN dim_date do ON f.order_placed_date_key = do.date_key
LEFT JOIN dim_date dp ON f.payment_confirmed_date_key = dp.date_key
LEFT JOIN dim_date dpick ON f.order_picked_date_key = dpick.date_key
LEFT JOIN dim_date ds ON f.order_shipped_date_key = ds.date_key
LEFT JOIN dim_date dd ON f.order_delivered_date_key = dd.date_key
WHERE f.order_key = 1001;
```

## Type 4: Factless Fact Tables

No numeric facts — the row's existence itself IS the fact. Two types: Event tracking (something happened) and Coverage tracking (something is available).

### Type 4a: Event Tracking (Student Enrollment)

A student enrolls in a course. The fact is the enrollment itself — no amounts or quantities. The row's existence means 'this student is taking this course.'

```
CREATE TABLE fact_student_enrollment (
    -- Grain: One row per student per course per semester
    student_key INT NOT NULL,
    course_key INT NOT NULL,
    semester_key INT NOT NULL,
    -- No numeric facts — just keys
    -- But metadata is useful:
    enrollment_date DATE,
    grade_received VARCHAR(2),  -- A, B, C, etc. (Dimension-like, not numeric)
    is_complete BOOLEAN,
    dw_load_date TIMESTAMP,
    PRIMARY KEY (student_key, course_key, semester_key)
);

-- Query: Gap analysis. Which students are NOT enrolled in any course this semester?
SELECT s.student_key, s.student_name
FROM dim_student s
WHERE NOT EXISTS (
    SELECT 1 FROM fact_student_enrollment f
    WHERE f.student_key = s.student_key
      AND f.semester_key = 20241  -- Current semester
);

-- Query: Course popularity
SELECT c.course_name, COUNT(*) as enrollment_count
FROM fact_student_enrollment f
JOIN dim_course c ON f.course_key = c.course_key
WHERE f.semester_key = 20241
GROUP BY c.course_name
ORDER BY enrollment_count DESC;
```

### Type 4b: Coverage Tracking (Promotion-Product)

A promotion covers certain products. The row's existence means 'this product is included in this promotion.' We later analyze: 'promoted products that didn't actually sell' (a gap to investigate).

```
CREATE TABLE fact_promotion_product_coverage (
    -- Grain: One row per promotion per product
    promotion_key INT NOT NULL,
    product_key INT NOT NULL,
    date_key INT NOT NULL,
    -- No numeric facts
    is_on_sale BOOLEAN DEFAULT TRUE,
    dw_load_date TIMESTAMP,
    PRIMARY KEY (promotion_key, product_key, date_key)
);

-- Query: Promoted products that didn't sell (opportunity gap)
SELECT p.product_name, pr.promotion_name, COUNT(DISTINCT f.customer_key) as customers_who_bought
```

```
FROM fact_promotion_product_coverage fpc
LEFT JOIN fact_pos_transaction f ON fpc.product_key = f.product_key
                                AND fpc.date_key = f.date_key
JOIN dim_product p ON fpc.product_key = p.product_key
JOIN dim_promotion pr ON fpc.promotion_key = pr.promotion_key
WHERE fpc.date_key = 20240314
GROUP BY p.product_name, pr.promotion_name
HAVING COUNT(DISTINCT f.customer_key) = 0  -- No sales despite promotion
ORDER BY p.product_name;
```

# Comparison: All Fact Types

A quick reference comparing the four fact types across key dimensions:

| Aspect | Transaction | Periodic Snapshot | Accumulating | Factless |
|---|---|---|---|---|
| Grain | Per event/transaction | Per entity per period | Per entity lifecycle | Per dimension combo |
| Rows Over Time | Unbounded growth | Bounded (1 row per entity per period) | One row, many updates | Varies |
| Sparsity | Sparse (not all combinations) | Dense (full coverage) | Very dense | Varies |
| Update Pattern | Insert only | Nightly batch inserts | Frequent updates | Insert or update |
| Use Case | Transaction detail analysis | Balance/status tracking | Process milestones | Existence/coverage |
| Example | POS transactions | Account balance snapshot | Order fulfillment | Course enrollment |
| Additive Facts | Yes (mostly) | Semi-additive | Lag metrics | None (factless) |

# Advanced Fact Table Patterns

## Multi-Currency Facts

When a business operates globally, amounts may be in different currencies. Store BOTH local_amount and reporting_amount. Use point-in-time exchange rates.

```
CREATE TABLE fact_international_sales (
    sales_key BIGINT PRIMARY KEY,
    date_key INT,
    customer_key INT,
    product_key INT,
    currency_key INT,           -- FK to dim_currency (USD, EUR, GBP, etc.)
    quantity_sold DECIMAL(10,2),
    local_amount DECIMAL(12,2),         -- In customer's local currency
    local_currency VARCHAR(3),          -- USD, EUR, etc.
    reporting_amount DECIMAL(12,2),     -- Always in USD (reporting currency)
    exchange_rate DECIMAL(8,4),         -- Rate used for conversion (point-in-time)
    dw_load_date TIMESTAMP
);

-- ETL: Apply exchange rate from dim_currency at time of sale
INSERT INTO fact_international_sales
SELECT ...,
    local_amount,
    c.currency_code,
    local_amount * dc.exchange_rate_to_usd as reporting_amount,
    dc.exchange_rate_to_usd,
    ...
FROM sales_source s
JOIN dim_currency dc ON s.currency_code = dc.currency_code
  AND s.sale_date BETWEEN dc.effective_date AND dc.expiry_date;

-- Query: Global revenue (all in USD, comparable)
SELECT SUM(reporting_amount) as total_usd_revenue
FROM fact_international_sales
WHERE date_key >= 20240101;
```

## Late-Arriving Facts

A fact arrives after its period has closed. Example: A transaction on Jan 30 doesn't post to the bank until Feb 5. Depending on source system reliability, you may: (1) Insert with the transaction date (date_key = Jan 30), or (2) Track it as a late arrival and reconcile.

> *For financial systems, late-arriving facts are critical. You'll add a 'is_late_arrival' flag and potentially insert a correction transaction in the subsequent period.*

## Error Correction: Reversals and Adjustments

When data is incorrect, you don't DELETE from the fact table. Instead, insert a reversal (negative amounts) or an adjustment. This maintains audit trail.

```
-- Original transaction (incorrect)
INSERT INTO fact_pos_transaction (..., net_amount) VALUES (..., 150.00);

-- Correction: Insert a reversal with negative amount
INSERT INTO fact_pos_transaction (..., net_amount) VALUES (..., -150.00);
-- Optionally, insert the correct amount:
INSERT INTO fact_pos_transaction (..., net_amount) VALUES (..., 145.00);

-- Query result: 150.00 - 150.00 + 145.00 = 145.00 (correct total)
-- Audit trail preserved: analyst can see reversal and correction.
```

# Common Fact Table Mistakes

**8 mistakes that plague fact table designs and how to avoid them:**

| Mistake | Why It's Bad | Fix |
|---|---|---|
| Storing dimension attributes (product_name) | Redundant storage, SCD complexity | Only store foreign keys; join dims for attributes |
| Non-numeric facts (status text) | Can't aggregate; breaks dimensionality | Store in dimension; use flags (0/1) if needed |
| Incorrect grain (fact has multiple customer keys) | Additive facts become non-sensical | Declare grain upfront; add degenerate dims if needed |
| Mixing additivity types without flagging | Analysts SUM non-additive facts | Document in metadata; use naming conventions |
| No surrogate keys on dimensions | SCD Type 2 becomes impossible | Every dimension needs surrogate key |
| Fact table at different grains | Aggregations are wrong | If grain changes, create separate fact |
| Ignoring NULL handling | NULLs in aggregations cause confusion | Use -1 surrogate key for unknown/not applicable |
| No dw_load_date or source_system | Can't audit; can't troubleshoot stale data | Always include metadata columns |

# CHAPTER 8: Dimension Tables — The Complete Guide

Dimensions answer the 'who, what, when, where, why, how' of a business event. While fact tables are sparse and numeric, dimensions are denormalized and descriptive. A fact table might have millions of rows; a dimension thousands or fewer — but with hundreds of columns of context.

## Dimension Characteristics

- DESCRIPTIVE: Text, codes, hierarchies. Answer 'what is this?'
- DENORMALIZED: Attributes stored flat (not in sub-tables), unlike 3NF.
- SMALL: Thousands to millions of rows (not billions).
- WIDE: Many columns (10-200) per row.
- SLOW-CHANGING: Attributes change infrequently (Type 2 SCD handles it).
- SURROGATE-KEYED: Artificial key (1, 2, 3...) separate from natural key.

## Conformed Dimensions

A conformed dimension is shared across multiple fact tables. dim_customer appears in fact_sales, fact_returns, and fact_support. Same surrogate key, same natural key, same attributes. This enables cross-process analysis.

> *Building conformed dimensions requires governance. A 'Master Data Management' team owns each conformed dimension and ensures all sources align.*

### Creating and Populating a Conformed Dimension

```
CREATE TABLE dim_customer (
    customer_key INT PRIMARY KEY,          -- Surrogate key
    customer_natural_id VARCHAR(50),       -- Natural key (business ID)
    customer_name VARCHAR(100),
    email VARCHAR(100),
    phone VARCHAR(20),
    country VARCHAR(50),
    state VARCHAR(50),
    city VARCHAR(50),
    zip_code VARCHAR(10),
    customer_segment VARCHAR(30),         -- Gold, Silver, Bronze, Prospect
    lifetime_revenue DECIMAL(14,2),
    -- SCD Type 2 columns
    is_current BOOLEAN DEFAULT TRUE,
    effective_date DATE,
    expiry_date DATE DEFAULT '9999-12-31',
    -- Metadata
    source_system VARCHAR(20),
    dw_load_date TIMESTAMP,
    dw_update_date TIMESTAMP
);

-- Populate from multiple source systems
```

```
INSERT INTO dim_customer
SELECT
    ROW_NUMBER() OVER (ORDER BY crm.customer_id) as customer_key,
    crm.customer_id,
    crm.customer_name,
    crm.email,
    crm.phone,
    crm.country,
    crm.state,
    crm.city,
    crm.zip_code,
    seg.segment_name,
    0 as lifetime_revenue,  -- Will be updated by daily jobs
    TRUE,
    CURRENT_DATE,
    '9999-12-31',
    'CRM',
    CURRENT_TIMESTAMP,
    CURRENT_TIMESTAMP
FROM crm.customer crm
LEFT JOIN crm.customer_segment seg ON crm.segment_id = seg.segment_id
WHERE crm.customer_status = 'Active';
```

## Example: dim_customer Across Three Processes

One dim_customer, used in three fact tables. Same surrogate key everywhere:

```
-- Three facts, one dimension
SELECT c.customer_name,
       COUNT(DISTINCT CASE WHEN f.fact_type = 'sales' THEN f.fact_id END) as sales_count,
       COUNT(DISTINCT CASE WHEN f.fact_type = 'returns' THEN f.fact_id END) as returns_count,
       COUNT(DISTINCT CASE WHEN f.fact_type = 'support' THEN f.fact_id END) as support_tickets
FROM dim_customer c
LEFT JOIN fact_sales fs ON c.customer_key = fs.customer_key
LEFT JOIN fact_returns fr ON c.customer_key = fr.customer_key
LEFT JOIN fact_support_ticket fst ON c.customer_key = fst.customer_key
WHERE c.is_current = TRUE
GROUP BY c.customer_name;
```

# Role-Playing Dimensions

A single dimension appears multiple times in one fact table, playing different roles. Example: dim_date plays 3 roles in an order fact: order_date, ship_date, delivery_date. Instead of three separate dim_date tables, we use views or aliases.

## Example: Date Dimension Plays 3 Roles

```
CREATE TABLE dim_date (
    date_key INT PRIMARY KEY,              -- YYYYMMDD format
    full_date DATE UNIQUE,
    year INT,
    month INT,
    day INT,
    day_of_week INT,
    day_of_year INT,
    quarter INT,
    is_holiday BOOLEAN,
    holiday_name VARCHAR(50)
);

-- Create views for each role
CREATE VIEW dim_date_order_placed AS
SELECT date_key as order_placed_date_key, year as order_year, month as order_month, ...
FROM dim_date;

CREATE VIEW dim_date_shipped AS
SELECT date_key as shipped_date_key, year as ship_year, month as ship_month, ...
FROM dim_date;

CREATE VIEW dim_date_delivered AS
SELECT date_key as delivered_date_key, year as delivery_year, month as delivery_month, ...
FROM dim_date;

-- The fact table references all three via foreign keys
CREATE TABLE fact_order (
    order_key INT,
    order_placed_date_key INT REFERENCES dim_date(date_key),
    order_shipped_date_key INT REFERENCES dim_date(date_key),
    order_delivered_date_key INT REFERENCES dim_date(date_key),
    ...
);

-- Query: Orders by order date with ship date and delivery delay
SELECT
    op.order_year,
    op.order_month,
    COUNT(*) as order_count,
    AVG(DATEDIFF(dd.delivered_date_key, os.shipped_date_key)) as avg_delivery_days
FROM fact_order f
JOIN dim_date_order_placed op ON f.order_placed_date_key = op.order_placed_date_key
JOIN dim_date_shipped os ON f.order_shipped_date_key = os.shipped_date_key
JOIN dim_date_delivered dd ON f.order_delivered_date_key = dd.delivered_date_key
GROUP BY op.order_year, op.order_month;
```

# Junk Dimensions

Low-cardinality flags and indicators (is_rush_order, is_gift_wrap, is_taxable) accumulate in fact tables. Rather than have 10 boolean columns, combine them into a single junk dimension. This reduces fact table width and improves query performance.

## Building a Junk Dimension

A junk dimension is typically built by taking the Cartesian product of all low-cardinality attributes. If you have 3 flag combinations (rush: Y/N, gift_wrap: Y/N, taxable: Y/N), you get 2×2×2 = 8 rows.

```
-- Example: Build junk dimension from 3 flags
CREATE TABLE dim_order_junk (
    order_junk_key INT PRIMARY KEY,
    is_rush_order BOOLEAN,
    is_gift_wrap BOOLEAN,
    is_taxable BOOLEAN
);

-- Populate with all combinations (2^3 = 8 rows)
INSERT INTO dim_order_junk
SELECT
    ROW_NUMBER() OVER (ORDER BY rush, gift, tax) as order_junk_key,
    rush as is_rush_order,
    gift as is_gift_wrap,
    tax as is_taxable
FROM (
    SELECT 'Y' as rush UNION ALL SELECT 'N'
) r CROSS JOIN (
    SELECT 'Y' as gift UNION ALL SELECT 'N'
) g CROSS JOIN (
    SELECT 'Y' as tax UNION ALL SELECT 'N'
) t;

-- Fact table now has one junk key instead of 3 boolean columns
CREATE TABLE fact_order_simplified (
    order_key INT,
    ...
    order_junk_key INT REFERENCES dim_order_junk(order_junk_key),
    ...
);

-- Query: Compare revenue by order type
SELECT
    j.is_rush_order,
    j.is_gift_wrap,
    j.is_taxable,
    COUNT(*) as order_count,
    SUM(f.order_amount) as total_revenue
FROM fact_order_simplified f
JOIN dim_order_junk j ON f.order_junk_key = j.order_junk_key
GROUP BY j.is_rush_order, j.is_gift_wrap, j.is_taxable
ORDER BY total_revenue DESC;
```

## When NOT to Use a Junk Dimension

- If a flag is changed frequently (SCD Type 2 needed), keep it in a separate dimension.

- If a flag has business meaning and needs metrics attached, make it a proper dimension.
- If the Cartesian product becomes very large (>10,000 rows), the junk dimension loses value.

# Degenerate Dimensions

A degenerate dimension is a key stored in the fact table that has NO corresponding dimension table. Example: transaction_number, invoice_number, order_number. These have no descriptive attributes — they're just identifiers.

```
CREATE TABLE fact_pos_transaction (
    ...
    -- Degenerate dimensions (keys with no dim table)
    transaction_number VARCHAR(20),
    transaction_line_number INT,
    ...
);

-- Why not create a dim_transaction table?
-- Because there's nothing to describe. A transaction_number IS just a number.
-- There are no attributes like transaction_name or transaction_category.
-- So we store it directly in the fact.

-- Query: Trace a specific transaction
SELECT * FROM fact_pos_transaction
WHERE transaction_number = 'RTL-20240314-00001234'
ORDER BY transaction_line_number;
```

# Mini-Dimensions (Type 4 SCD)

When a dimension has a few attributes that change frequently (every month), while most attributes change rarely, split into a base dimension (Type 2) and a mini-dimension (Type 4). The fact table gets both keys.

## Example: Customer with Changing Demographics

Customer's name, email, phone change rarely. But age_band, income_band, credit_score_band change monthly (after review). Create mini-dim for these.

```
-- Base dimension (Type 2: full history)
CREATE TABLE dim_customer (
    customer_key INT PRIMARY KEY,
    customer_natural_id VARCHAR(50),
    customer_name VARCHAR(100),        -- Changes rarely
    email VARCHAR(100),                 -- Changes rarely
    phone VARCHAR(20),                  -- Changes rarely
    is_current BOOLEAN,
    effective_date DATE,
    expiry_date DATE,
    dw_load_date TIMESTAMP
);

-- Mini-dimension (Type 4: high-velocity attributes)
CREATE TABLE dim_customer_demographics_mini (
    customer_demographics_key INT PRIMARY KEY,
    customer_key INT,                  -- FK to base dim
    age_band VARCHAR(20),              -- 18-25, 26-35, etc.
    income_band VARCHAR(20),           -- Low, Medium, High
    credit_score_band VARCHAR(20),     -- Excellent, Good, Fair, Poor
    effective_date DATE,
    expiry_date DATE,
    dw_load_date TIMESTAMP
```

```
);

-- Fact uses both keys
CREATE TABLE fact_transaction (
    ...
    customer_key INT,                  -- FK to dim_customer (base)
    customer_demographics_key INT,     -- FK to dim_customer_demographics_mini
    ...
);

-- ETL: Update mini-dim monthly
-- 1. Check if demographic attributes have changed
-- 2. If yes, insert new row in mini-dim (close old row's expiry_date)
-- 3. Future transactions use new demographics key

-- Query: Product affinity by age band
SELECT d.age_band, p.product_category, COUNT(*) as purchase_count
FROM fact_transaction f
JOIN dim_customer_demographics_mini d ON f.customer_demographics_key = d.customer_demographics_key
JOIN dim_product p ON f.product_key = p.product_key
WHERE d.is_current = TRUE
GROUP BY d.age_band, p.product_category;
```

# Outrigger Dimensions

A dimension that points to another dimension (not directly to the fact). Example: dim_product → dim_brand. This is mild snowflaking. Kimball generally discourages it ('denormalize dimensions'), but it can be justified for very large dimensions with stable hierarchies.

```
-- Denormalized (Kimball way): Put brand info in dim_product
CREATE TABLE dim_product_denormalized (
    product_key INT PRIMARY KEY,
    product_name VARCHAR(100),
    brand_name VARCHAR(50),
    brand_country VARCHAR(50),
    category_name VARCHAR(50),
    ...
);

-- Outrigger (slight snowflaking): Separate dim_brand
CREATE TABLE dim_brand (
    brand_key INT PRIMARY KEY,
    brand_name VARCHAR(50),
    brand_country VARCHAR(50),
    ...
);

CREATE TABLE dim_product_with_outrigger (
    product_key INT PRIMARY KEY,
    product_name VARCHAR(100),
    brand_key INT REFERENCES dim_brand(brand_key),  -- Points to dim, not fact
    category_name VARCHAR(50),
    ...
);

-- Query with outrigger (one extra join)
SELECT b.brand_name, COUNT(*) as unit_sales
FROM fact_sales f
JOIN dim_product_with_outrigger p ON f.product_key = p.product_key
JOIN dim_brand b ON p.brand_key = b.brand_key
WHERE f.date_key >= 20240101
GROUP BY b.brand_name;
```

# Slowly Changing Dimension Introduction

Dimensions change. A customer's address, a product's category, an employee's salary. The SCD framework (Types 0-7) determines how to handle these changes. We'll detail this extensively in Chapter 9.

# Surrogate Keys: Why, What, How

Every dimension needs a surrogate key — an artificial integer (1, 2, 3...) separate from the business/natural key. Reasons: (1) SCD Type 2 requires multiple rows per entity; surrogate keys distinguish them. (2) Smaller foreign key columns (INT vs VARCHAR). (3) Hides natural key changes from the fact table.

## The Unknown and Not Applicable Members

Every dimension has two special rows: -1 for 'Unknown' (data missing at load time) and -2 for 'Not Applicable' (dimension not relevant for this fact).

```
-- Pre-load special rows
INSERT INTO dim_customer VALUES
    (-1, 'UNKNOWN', 'Unknown Customer', NULL, NULL, NULL, NULL, NULL, NULL, NULL, 0,
     TRUE, '1900-01-01', '9999-12-31', 'SYSTEM', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP),
    (-2, 'NOT_APPLICABLE', 'Not Applicable', NULL, NULL, NULL, NULL, NULL, NULL, NULL, 0,
     TRUE, '1900-01-01', '9999-12-31', 'SYSTEM', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP);

-- Use -1 when customer_id is missing in source
UPDATE fact_transaction
SET customer_key = -1
WHERE customer_key IS NULL;

-- Use -2 when transaction type doesn't involve a customer
UPDATE fact_inventory_adjustment
SET customer_key = -2
WHERE transaction_type = 'STOCK_COUNT';
```

# Dimension Hierarchies

Hierarchies allow drill-down analysis: Company → Region → Store → Department. Kimball recommends denormalizing hierarchies into the dimension (not separate tables).

## Fixed-Depth Hierarchy (Geographic)

```
CREATE TABLE dim_store (
    store_key INT PRIMARY KEY,
    store_id VARCHAR(20),
    store_name VARCHAR(100),
    -- Denormalized hierarchy levels
    country VARCHAR(50),
    state VARCHAR(50),
    city VARCHAR(50),
    district VARCHAR(50),
    store_manager VARCHAR(100),
    ...
);

-- Query: Roll-up by geography
SELECT c.country, s.state, COUNT(*) as store_count, SUM(f.sales_amount) as revenue
FROM fact_sales f
JOIN dim_store s ON f.store_key = s.store_key
GROUP BY c.country, s.state
ORDER BY c.country, revenue DESC;
```

## Variable-Depth Hierarchy (Organization Chart)

Some hierarchies are ragged (not all paths are the same depth). Example: Organization chart where some managers have 2 levels of reports, others have 5. Denormalization becomes complex; consider a bridge table.

```
CREATE TABLE dim_employee (
    employee_key INT PRIMARY KEY,
```

```
    employee_id VARCHAR(20),
    employee_name VARCHAR(100),
    -- Can't denormalize ragged hierarchy here
    ...
);

-- Bridge table for org chart
CREATE TABLE dim_employee_bridge (
    employee_key INT,
    manager_key INT,
    hierarchy_level INT,      -- 1 = direct report, 2 = skip-level, etc.
    PRIMARY KEY (employee_key, manager_key)
);

-- Query: Organization rollup
SELECT m.employee_name, COUNT(DISTINCT e.employee_key) as employee_count
FROM dim_employee_bridge b
JOIN dim_employee e ON b.employee_key = e.employee_key
JOIN dim_employee m ON b.manager_key = m.employee_key
WHERE b.hierarchy_level = 1  -- Direct reports only
GROUP BY m.employee_name;
```

# Comparison Table: All Dimension Types

| Dimension Type | Use Case | Example | SCD Type | Key Points |
|---|---|---|---|---|
| Conformed | Shared across facts | dim_customer in sales, returns, support | Type 2 | Same key everywhere; requires governance |
| Role-Playing | Multiple uses in one fact | dim_date as order_date, ship_date | Same dim | Use views; multiple FK in fact |
| Junk | Low-card flags combined | is_rush, is_gift, is_taxable | Type 1 | Cartesian product of combos |
| Degenerate | Key with no attributes | transaction_number | N/A | Lives in fact, no dim table |
| Mini | High-velocity attributes | age_band, income_band | Type 4 | Separate from base dim |
| Outrigger | Dim-to-dim relationship | dim_product → dim_brand | Type 2 | Mild snowflaking |
| Shrunken | Higher-grain aggregate | dim_month (subset of dim_date) | Same dim | Used in aggregate facts |

# Common Dimension Mistakes

| Mistake | Why It's Bad | Fix |
|---|---|---|
| Storing dimension attributes in fact | Redundancy; SCD nightmare | Always join to dimension |
| Multiple hierarchies in one dimension | Reporting complexity; no roll-up | Create separate dimension or use bridge |
| No surrogate keys | SCD Type 2 impossible; big natural keys | Generate surrogate key for every dim |
| Forgetting unknown/not applicable members (-1, -2) | NULLs break SQL logic and aggregates | Always pre-load -1 and -2 |
| Inconsistent naming across conformed dimensions | Analysts confused which dims to join | Central governance; naming standards |
| Snowflaking (too many dim-to-dim joins) | Complex queries; slow | Denormalize 95% of cases |
| Not tracking SCD type in metadata | Dimension changes misunderstood | Document each attribute's SCD type |

# CHAPTER 9: Slowly Changing Dimensions — Complete Reference

Dimensions change. A customer moves to a new address. A product changes category. An employee gets a promotion. The Slowly Changing Dimension (SCD) framework, developed by Kimball, defines how to track these changes. Types 0-7 offer different trade-offs between history, complexity, and query speed.

## Why SCD Matters: The Price-at-Time-of-Sale Problem

A customer buys a product on January 1 at price $100. On January 31, the price changes to $90. In February, an analyst runs a report: 'revenue by product.' Without SCD, there's ambiguity: Was the Jan 1 sale at $100 or $90? SCD ensures the Jan 1 transaction is forever linked to the Jan 1 price. The Feb 1 transaction links to the Feb 1 price.

## Type 0: Fixed (Retain Original)

The dimension value never changes (or you pretend it doesn't). The original value is retained forever. Use this for audit/regulatory attributes: 'what was the customer's credit score at account opening?' That value freezes.

```
CREATE TABLE dim_account (
    account_key INT,
    account_id VARCHAR(20),
    account_type VARCHAR(20),
    customer_name VARCHAR(100),
    original_credit_score INT,         -- Type 0: Never changes
    current_credit_score INT,          -- Separate column, Type 1
    account_opened_date DATE,          -- Type 0: Never changes
    is_current BOOLEAN,
    effective_date DATE,
    expiry_date DATE,
    ...
);

-- The original_credit_score at account opening is immutable.
-- If the customer's credit improves, current_credit_score changes (Type 1 overwrite),
-- but original_credit_score stays the same.
```

## Type 1: Overwrite

Simply UPDATE the attribute. Lose all history. Use for: corrections, attributes where history has no business value, or non-critical fields.

```
-- Type 1: Just overwrite
UPDATE dim_product
SET product_category = 'Electronics'
WHERE product_id = 'SKU-12345';

-- Before: Category was 'Office Supplies'
-- After: Category is 'Electronics'
```

```
-- History: Lost (no way to know it was 'Office Supplies')

-- Good use cases:
-- - Fixing typos (customer_name spelled wrong)
-- - Updating administrative fields (warehouse_location)
-- Bad use cases:
-- - Changing a customer's address (you lose where they lived before)
-- - Changing a product's price (you lose historical pricing)
```

# Type 2: Add New Row (Full History)

THE most important SCD type. When an attribute changes, CLOSE the old row (set expiry_date to yesterday, is_current = FALSE) and INSERT a new row (set effective_date to today, is_current = TRUE). Fact table surrogate keys point to specific versions, so historical facts remain linked to the correct dimension state.

## Type 2 Schema

```
CREATE TABLE dim_customer_type2 (
    customer_key INT PRIMARY KEY,       -- Surrogate key (each version has different key)
    customer_natural_id VARCHAR(50),    -- Same for all versions
    customer_name VARCHAR(100),
    address VARCHAR(200),
    city VARCHAR(50),
    state CHAR(2),
    zip_code VARCHAR(10),
    -- SCD Type 2 tracking columns
    is_current BOOLEAN DEFAULT TRUE,
    effective_date DATE,
    expiry_date DATE DEFAULT '9999-12-31',
    dw_load_date TIMESTAMP
);

-- Sample data: Customer moved in Feb, changed name in Apr
INSERT INTO dim_customer_type2 VALUES
(100, 'CUST-001', 'John Smith', '123 Oak St', 'Boston', 'MA', '02101',
FALSE, '2024-01-01', '2024-02-29', '2024-01-10');
(101, 'CUST-001', 'John Smith', '456 Elm Ave', 'Cambridge', 'MA', '02138',
FALSE, '2024-03-01', '2024-04-29', '2024-03-01');
(102, 'CUST-001', 'John Q. Smith', '456 Elm Ave', 'Cambridge', 'MA', '02138',
TRUE, '2024-05-01', '9999-12-31', '2024-05-02');
```

## Type 2 Loading: Detecting Changes and SCD Logic

```
-- Nightly ETL: Detect changes, close old rows, insert new rows
WITH changes AS (
    -- Find attributes that differ between source and current dimension
    SELECT s.customer_natural_id, s.customer_name, s.address, s.city, s.state, s.zip_code,
           d.customer_key, d.customer_name as dim_name, d.address as dim_address,
           d.city as dim_city, d.state as dim_state, d.zip_code as dim_zip
    FROM source_customer s
    LEFT JOIN dim_customer_type2 d ON s.customer_natural_id = d.customer_natural_id
                                   AND d.is_current = TRUE
    WHERE s.customer_natural_id NOT IN (
```

```
        SELECT customer_natural_id FROM dim_customer_type2 WHERE is_current = TRUE
    )
       OR s.customer_name != d.customer_name
       OR s.address != d.address
       OR s.city != d.city
       OR s.state != d.state
       OR s.zip_code != d.zip_code
)
-- Step 1: Close old rows
UPDATE dim_customer_type2
SET is_current = FALSE,
    expiry_date = CURRENT_DATE - 1
FROM changes c
WHERE dim_customer_type2.customer_key = c.customer_key
  AND dim_customer_type2.is_current = TRUE;

-- Step 2: Insert new rows
INSERT INTO dim_customer_type2 (customer_key, customer_natural_id, customer_name, address,
                                city, state, zip_code, is_current, effective_date, expiry_date,
dw_load_date)
SELECT NEXTVAL('dim_customer_key_seq'), s.customer_natural_id, s.customer_name, s.address,
       s.city, s.state, s.zip_code, TRUE, CURRENT_DATE, '9999-12-31', CURRENT_TIMESTAMP
FROM source_customer s
WHERE s.customer_natural_id IN (SELECT customer_natural_id FROM changes);
```

## Type 2 Queries: Point-in-Time, Current, and History

```
-- Query 1: Current state (who is the customer now?)
SELECT customer_natural_id, customer_name, address, city, state, zip_code
FROM dim_customer_type2
WHERE is_current = TRUE;

-- Query 2: What was customer's address on specific date?
SELECT customer_natural_id, customer_name, address
FROM dim_customer_type2
WHERE customer_natural_id = 'CUST-001'
  AND effective_date <= '2024-04-15'
  AND expiry_date >= '2024-04-15';

-- Query 3: Revenue by customer with their current address
SELECT c.customer_name, c.address,
       SUM(f.net_amount) as total_revenue
FROM fact_sales f
JOIN dim_customer_type2 c ON f.customer_key = c.customer_key
WHERE c.is_current = TRUE
  AND f.date_key >= 20240101
GROUP BY c.customer_name, c.address;

-- Query 4: Change history (show all versions of a customer)
SELECT effective_date, expiry_date, customer_name, address, city, state
FROM dim_customer_type2
WHERE customer_natural_id = 'CUST-001'
ORDER BY effective_date;

-- Query 5: Fact linked to specific dimension version
-- The fact_sales table has customer_key (surrogate key).
-- This surrogate key points to a SPECIFIC version of the customer.
SELECT s.sale_date, c.customer_name, c.address, f.sale_amount
FROM fact_sales f
```

```
JOIN dim_date s ON f.date_key = s.date_key
JOIN dim_customer_type2 c ON f.customer_key = c.customer_key
WHERE s.sale_date = '2024-03-15';
-- If customer moved, this returns their address at time of sale, not today's address.
```

## Type 2 Performance and Indexing

Type 2 dimensions grow over time (one row per change). For a slowly-changing attribute, growth is manageable. But in high-transaction scenarios, indexes are critical.

```
-- Essential indexes for Type 2
CREATE INDEX idx_dim_customer_natural_id ON dim_customer_type2(customer_natural_id, is_current);
CREATE INDEX idx_dim_customer_current ON dim_customer_type2(is_current, effective_date);
CREATE INDEX idx_dim_customer_dates ON dim_customer_type2(effective_date, expiry_date);

-- These help queries like:
-- - Find current row for a natural key
-- - Find all current rows
-- - Find row valid on a specific date
```

## Type 3: Previous Value Column

Add a 'previous' column alongside the 'current' column. When an attribute changes, save the old value in the previous column and update the current. Minimal history (one change level); minimal complexity.

```
CREATE TABLE dim_customer_type3 (
    customer_key INT PRIMARY KEY,
    customer_natural_id VARCHAR(50),
    customer_name VARCHAR(100),
    -- Current vs Previous
    current_address VARCHAR(200),
    previous_address VARCHAR(200),
    current_city VARCHAR(50),
    previous_city VARCHAR(50),
    -- Metadata
    address_change_date DATE,
    dw_load_date TIMESTAMP,
    dw_update_date TIMESTAMP
);

-- Type 3 Update: Shift current → previous, then update current
UPDATE dim_customer_type3
SET previous_address = current_address,
    previous_city = current_city,
    current_address = '456 Elm Ave',
    current_city = 'Cambridge',
    address_change_date = CURRENT_DATE,
    dw_update_date = CURRENT_TIMESTAMP
WHERE customer_natural_id = 'CUST-001';

-- Query: Compare current vs previous
SELECT customer_name, current_address, previous_address
FROM dim_customer_type3
WHERE previous_address IS NOT NULL
ORDER BY address_change_date DESC;

-- Limitation: Only tracks one change (previous). If address changes 3 times,
-- you lose the oldest two.
```

## Type 4: Mini-Dimension (Covered in Chapter 8)

High-velocity attributes get their own separate dimension. Base dimension Type 1/2, mini-dimension Type 2. Fact table has keys to both.

## Type 6: Type 1 + Type 2 + Type 3 Combined

The most flexible but most complex. New row on change (Type 2) + previous column (Type 3) + overwrite current row for reporting (Type 1). Enables both historical and point-in-time analysis and easy current/previous comparison.

```
CREATE TABLE dim_customer_type6 (
    customer_key INT PRIMARY KEY,
    customer_natural_id VARCHAR(50),
    -- Current values (always updated)
    current_address VARCHAR(200),
    current_city VARCHAR(50),
```

```
    -- Previous values (for easy comparison)
    previous_address VARCHAR(200),
    previous_city VARCHAR(50),
    -- History tracking
    effective_date DATE,
    expiry_date DATE,
    is_current BOOLEAN,
    dw_load_date TIMESTAMP
);

-- Type 6 Load: Detect change, close old row, insert new row, update addresses
-- Step 1: INSERT new row with new address
INSERT INTO dim_customer_type6
VALUES (NEXTVAL('dim_customer_key_seq'), 'CUST-001', '456 Elm Ave', 'Cambridge',
        '123 Oak St', 'Boston', CURRENT_DATE, '9999-12-31', TRUE, CURRENT_TIMESTAMP);

-- Step 2: UPDATE previous current row (set expiry, is_current = FALSE)
UPDATE dim_customer_type6
SET expiry_date = CURRENT_DATE - 1,
    is_current = FALSE
WHERE customer_natural_id = 'CUST-001'
  AND is_current = TRUE
  AND customer_key != LASTVAL();  -- Don't update the row we just inserted

-- Step 3: ALSO update the new row's previous_address from old row's current_address
-- (This is often a separate ETL step)

-- Query Type 6A: Current state (easy, no joins)
SELECT customer_natural_id, current_address, current_city
FROM dim_customer_type6
WHERE is_current = TRUE;

-- Query Type 6B: Historical state (SCD Type 2 style)
SELECT customer_natural_id, current_address
FROM dim_customer_type6
WHERE customer_natural_id = 'CUST-001'
  AND effective_date <= '2024-03-15'
  AND expiry_date >= '2024-03-15';

-- Query Type 6C: Compare current vs previous (SCD Type 3 style)
SELECT customer_natural_id, current_address, previous_address
FROM dim_customer_type6
WHERE is_current = TRUE
  AND current_address != previous_address;  -- Only show those who moved
```

# SCD Type Comparison Table

| Type | Method | History | Complexity | Query Pattern | Storage | Use Case |
|---|---|---|---|---|---|---|
| 0 | Fixed | None (immutable) | Low | Always original | Minimal | Audit values |
| 1 | Overwrite | None | Very Low | Always current | Minimal | Corrections |
| 2 | New Row | Full | High | Point-in-time (with dates) | High | All history |
| 3 | Prev Column | One level | Low | Current vs Previous | Low | Easy comparison |
| 4 | Mini-Dim | Full (separate dim) | Moderate | Both keys in fact | Moderate | High-velocity |
| 6 | 1+2+3 | Full + Easy | Very High | Both modes | High | Max flexibility |

# Hybrid SCD: Different Columns, Different Types

In a single dimension, different columns can have different SCD types. Example: dim_employee with Type 0 (SSN — immutable), Type 1 (city — corrections only), Type 2 (salary_band — full history), Type 3 (job_title — current and previous).

```
CREATE TABLE dim_employee_hybrid (
    employee_key INT PRIMARY KEY,
    employee_id VARCHAR(20),
    -- Type 0: Never changes
    ssn VARCHAR(11),
    hire_date DATE,
    -- Type 1: Overwrite only
    current_address VARCHAR(200),
    -- Type 2: Full history via SCD tracking
    salary_band VARCHAR(20),
    effective_date DATE,
    expiry_date DATE,
    is_current BOOLEAN,
    -- Type 3: Current + Previous
    current_job_title VARCHAR(100),
    previous_job_title VARCHAR(100),
    dw_load_date TIMESTAMP
);

-- Each attribute update follows its SCD type rule:
-- - Update ssn: ERROR (Type 0, immutable)
-- - Update current_address: UPDATE (Type 1, overwrite)
-- - Change salary_band: INSERT new row, close old (Type 2)
-- - Promote employee: Set previous_job_title = current_job_title, then UPDATE current (Type 3)
```

# Late-Arriving Dimensions

A fact row arrives before its dimension. Example: A transaction references customer_id 99999, but customer 99999 doesn't exist in the dimension yet. Create an 'inferred' Type 2 record with just the natural key, then backfill attributes when the real dimension data arrives.

```
-- Transaction arrives for unknown customer
INSERT INTO fact_sales (..., customer_key, ...)
VALUES (..., -1, ...);  -- Use unknown member

-- Later, create inferred member
```

```
INSERT INTO dim_customer
SELECT NEXTVAL('dim_customer_key_seq'), 'CUST-99999', 'Unknown', NULL, NULL, ...
FROM (SELECT DISTINCT customer_id FROM source_transactions WHERE customer_id NOT IN
      (SELECT customer_natural_id FROM dim_customer));

-- Later still, backfill attributes when customer data arrives
UPDATE dim_customer
SET customer_name = 'John Doe',
    address = '123 Main',
    ...
WHERE customer_natural_id = 'CUST-99999'
  AND is_current = TRUE;

-- If this is Type 2, the inferred row becomes the old version:
-- Insert new row with full attributes, close old inferred row
```

# CHAPTER 10: Star Schema vs Snowflake Schema

Two physical design patterns for dimensional models: star (denormalized, Kimball's preference) and snowflake (normalized, resembles a normalized 3NF design). The choice affects query complexity, storage, and tool friendliness.

## Star Schema: Denormalized Dimensions

Dimensions are fully denormalized. All attributes stored flat. Fact table sits in the center, surrounded by dimension tables (resembles a star visually). Simple queries; typically the default Kimball design.

### Star Schema DDL

```
-- Star schema: fact in center, denormalized dimensions around it
CREATE TABLE fact_sales (
    sales_key BIGINT PRIMARY KEY,
    date_key INT,
    product_key INT,
    store_key INT,
    customer_key INT,
    quantity INT,
    amount DECIMAL(12,2),
    FOREIGN KEY (date_key) REFERENCES dim_date(date_key),
    FOREIGN KEY (product_key) REFERENCES dim_product(product_key),
    FOREIGN KEY (store_key) REFERENCES dim_store(store_key),
    FOREIGN KEY (customer_key) REFERENCES dim_customer(customer_key)
);

-- Denormalized dim_product (all attributes here, no sub-tables)
CREATE TABLE dim_product (
    product_key INT PRIMARY KEY,
    product_id VARCHAR(20),
    product_name VARCHAR(100),
    brand_name VARCHAR(50),           -- Denormalized from dim_brand
    brand_country VARCHAR(50),        -- Denormalized from dim_brand
    category_name VARCHAR(50),        -- Denormalized from dim_category
    category_parent VARCHAR(50),      -- Denormalized from dim_category
    supplier_name VARCHAR(100),       -- Denormalized from dim_supplier
    supplier_country VARCHAR(50),     -- Denormalized from dim_supplier
    list_price DECIMAL(10,2),
    is_current BOOLEAN,
    effective_date DATE,
    expiry_date DATE
);

-- Similarly denormalized dim_store, dim_customer, dim_date
```

### Star Schema Query

```
-- Star schema query: 4 simple joins
SELECT d.year, d.month,
```

```
       p.product_name, p.brand_name,
       s.store_name, s.city,
       c.customer_name,
       SUM(f.quantity) as total_qty,
       SUM(f.amount) as total_revenue
FROM fact_sales f
JOIN dim_date d ON f.date_key = d.date_key
JOIN dim_product p ON f.product_key = p.product_key
JOIN dim_store s ON f.store_key = s.store_key
JOIN dim_customer c ON f.customer_key = c.customer_key
WHERE d.year = 2024
  AND p.brand_name = 'Acme'
GROUP BY d.year, d.month, p.product_name, p.brand_name, s.store_name, s.city, c.customer_name;
```

### Advantages of Star

- SIMPLICITY: Analysts understand one fact surrounded by 5-10 dimensions.
- QUERY SPEED: Few joins; query optimizer handles 4-5 joins efficiently.
- BI TOOL FRIENDLY: Looker, Power BI, Tableau assume star schema internally.
- STRAIGHTFORWARD ETL: Load fact, load dimensions, join keys match.

## Snowflake Schema: Normalized Dimensions

Dimensions are normalized into sub-tables. dim_product points to dim_brand, which points to dim_brand_country. Resembles a hierarchical ER diagram more than a star.

### Snowflake Schema DDL

```
-- Snowflake: dimensions are normalized into sub-tables
CREATE TABLE fact_sales (
    sales_key BIGINT,
    date_key INT,
    product_key INT,              -- Points to dim_product (which points to brand/category)
    store_key INT,
    customer_key INT,
    quantity INT,
    amount DECIMAL(12,2)
);

CREATE TABLE dim_product (
    product_key INT PRIMARY KEY,
    product_id VARCHAR(20),
    product_name VARCHAR(100),
    brand_key INT,                -- FK to dim_brand (not brand_name text)
    category_key INT,             -- FK to dim_category
    supplier_key INT,             -- FK to dim_supplier
    list_price DECIMAL(10,2)
);

CREATE TABLE dim_brand (
    brand_key INT PRIMARY KEY,
    brand_id VARCHAR(20),
    brand_name VARCHAR(50),
    country_key INT               -- FK to dim_brand_country (further normalized)
```

```
);

CREATE TABLE dim_brand_country (
    country_key INT PRIMARY KEY,
    country_name VARCHAR(50),
    region VARCHAR(50)
);

CREATE TABLE dim_category (
    category_key INT PRIMARY KEY,
    category_id VARCHAR(20),
    category_name VARCHAR(50),
    parent_category_key INT        -- FK to parent category
);

-- Similar for dim_supplier, dim_store (pointing to dim_location), etc.
```

### Snowflake Schema Query

```
-- Snowflake query: Many more joins (7+)
SELECT d.year, d.month,
       p.product_name, b.brand_name, bc.country_name,
       c.category_name,
       s.store_name, l.city,
       cust.customer_name,
       SUM(f.quantity) as total_qty,
       SUM(f.amount) as total_revenue
FROM fact_sales f
JOIN dim_date d ON f.date_key = d.date_key
JOIN dim_product p ON f.product_key = p.product_key
JOIN dim_brand b ON p.brand_key = b.brand_key
JOIN dim_brand_country bc ON b.country_key = bc.country_key
JOIN dim_category c ON p.category_key = c.category_key
JOIN dim_store s ON f.store_key = s.store_key
JOIN dim_location l ON s.location_key = l.location_key
JOIN dim_customer cust ON f.customer_key = cust.customer_key
WHERE d.year = 2024
  AND b.brand_name = 'Acme'
GROUP BY ...;
-- Note: 8 joins vs star's 4. More complex, slower on older query engines.
```

### Advantages of Snowflake

- LESS STORAGE: No redundancy. Brand info stored once, not repeated per product.
- DATA INTEGRITY: If brand changes, update one row, not thousands (3NF discipline).
- EASIER UPDATES: Change brand country once; reflected across all products.

## Head-to-Head Comparison: 12 Criteria

| Criterion | Star Schema | Snowflake Schema |
|---|---|---|
| Query Complexity | Simple (4-5 joins) | Complex (8-15 joins) |
| Query Performance | Fast (modern optimizers handle it well) | Slower (more joins, larger plans) |
| Storage Size | Larger (redundancy from denormalization) | Smaller (normalized, no repeats) |

| Criterion | Star Schema | Snowflake Schema |
|---|---|---|
| Dimension Updates | Update all product rows if brand changes | Update one brand row; cascades |
| Data Integrity | Possible inconsistencies (denormalized) | Enforced (3NF structure) |
| BI Tool Support | Excellent (tools built for star) | Fair (requires materialized views/ETL) |
| ETL Complexity | Moderate (denormalize during load) | High (integrate from many tables) |
| Null Handling | Simple (nulls in fact or product row) | Complex (nulls cascade through hierarchy) |
| Reporting Speed | Fast (fewer joins; pre-aggregates work) | Slower (more joins to traverse) |
| Analyst Learning Curve | Low (star is intuitive) | High (multiple dim layers) |
| Historical Tracking (SCD Type 2) | More complex (denorm changes) | More rows (each level changes) |

## When to Snowflake (The Exceptions)

- MASSIVE dimension (millions of rows): If dim_product has 10M rows and brand has 1K, snowflake saves space (brand info not repeated 10M times).
- COMPLEX hierarchies: Product → Category → SubCategory → SubSubCategory. Snowflake structure reflects this naturally.
- STRICT data governance: Finance/healthcare require 3NF to ensure consistency. Snowflake forces good practices.
- EXTERNAL reporting systems: Some tools (older ERP systems) work better with normalized structures.

## Modern Perspective: Cloud Data Warehouses

Cloud warehouses (Snowflake, BigQuery, Redshift) have changed the calculus. Storage is so cheap that denormalization is almost always preferred. Query engines are powerful enough to optimize 20 joins. Kimball's 'always star' advice is even stronger now.

> *If building a new warehouse today: Use star schema. If you need normalized tables for governance, build those separately (operational data store) and feed a dimensional layer on top.*

# CHAPTER 11: Advanced Dimensional Modelling Patterns

Real-world business complexity often requires patterns beyond basic star schema. This chapter covers advanced techniques for handling heterogeneous products, multi-currency, late arrivals, and more.

## Heterogeneous Products: The Attribute Problem

A clothing retailer sells TVs, shoes, and furniture. TVs have screen_size, shoes have shoe_size, furniture has material. A single product dimension can't efficiently store attributes that don't apply to all products.

### Solution 1: Generic Dimension (Not Recommended)

```
-- Problem: NULL-heavy
CREATE TABLE dim_product_generic (
    product_key INT,
    product_id VARCHAR(20),
    product_name VARCHAR(100),
    product_type VARCHAR(20),      -- TV, Shoe, Furniture
    generic_attribute1 VARCHAR(50),  -- screen_size for TVs, shoe_size for shoes
    generic_attribute2 VARCHAR(50),  -- material for furniture
    ...
);

-- Query becomes: Where do I find screen_size? Is it attribute1 or attribute2? Depends on
product_type.
-- Complex, error-prone, leads to many NULLs.
```

### Solution 2: Entity-Attribute-Value (EAV) — Kimball Warns Against

```
-- EAV: Every attribute is a row
CREATE TABLE dim_product_eav (
    product_key INT,
    product_id VARCHAR(20),
    product_name VARCHAR(100),
    product_type VARCHAR(20)
);

CREATE TABLE dim_product_attribute (
    product_key INT,
    attribute_name VARCHAR(50),     -- screen_size, shoe_size, material, weight
    attribute_value VARCHAR(100),
    PRIMARY KEY (product_key, attribute_name)
);

-- Query: Get TVs with screen size >= 50
SELECT p.product_name
FROM dim_product_eav p
JOIN dim_product_attribute a ON p.product_key = a.product_key
WHERE p.product_type = 'TV'
  AND a.attribute_name = 'screen_size'
```

```
  AND CAST(a.attribute_value AS INT) >= 50;

-- Problem: Every query becomes complex. Comparison queries (find matching products) are slow.
-- Kimball generally advises against EAV in data warehouses.
```

### Solution 3: Modern Approach — JSON/VARIANT Columns

Cloud warehouses support semi-structured data. Store product-specific attributes as JSON in a single column. Query using native JSON functions.

```
-- Modern approach: VARIANT/JSON column (Snowflake, BigQuery)
CREATE TABLE dim_product_modern (
    product_key INT,
    product_id VARCHAR(20),
    product_name VARCHAR(100),
    product_type VARCHAR(20),
    attributes VARIANT                -- JSON column for flexible attributes
);

-- Insert examples
INSERT INTO dim_product_modern VALUES
(1, 'TV-001', '55-inch OLED', 'TV', '{"screen_size": 55, "resolution": "4K", "hdr": true}'),
(2, 'SHOE-001', 'Running Shoe Size 10', 'Shoe', '{"shoe_size": 10, "gender": "M", "color": "blue"}'),
(3, 'SOFA-001', 'Leather Sofa', 'Furniture', '{"material": "leather", "seating_capacity": 3, "color":
"black"}');

-- Query: Get TVs with screen size >= 50
SELECT product_name
FROM dim_product_modern
WHERE product_type = 'TV'
  AND attributes:screen_size >= 50;  -- Snowflake syntax
```

### Solution Comparison

| Approach | Complexity | Query Speed | Storage | Flexibility | Use Case |
|---|---|---|---|---|---|
| Generic Dimension | High (nulls) | Medium | Medium | Low | Don't use |
| EAV | Very High | Slow | Large | Very High | Rare (reporting on attributes) |
| JSON/VARIANT | Low | Fast | Efficient | High | Modern (best choice) |

## Multi-Currency Facts

Store both local_amount and reporting_amount. Apply point-in-time exchange rates.

```
CREATE TABLE dim_exchange_rate (
    exchange_rate_key INT PRIMARY KEY,
    from_currency CHAR(3),
    to_currency CHAR(3),           -- Usually USD
    exchange_rate DECIMAL(8,4),
    effective_date DATE,
    expiry_date DATE DEFAULT '9999-12-31'
);

CREATE TABLE fact_international_sales (
```

```
    sales_key BIGINT PRIMARY KEY,
    date_key INT,
    product_key INT,
    customer_key INT,
    exchange_rate_key INT,
    quantity_sold INT,
    local_amount DECIMAL(12,2),
    local_currency CHAR(3),
    reporting_amount DECIMAL(12,2),  -- Converted to USD
    FOREIGN KEY (exchange_rate_key) REFERENCES dim_exchange_rate(exchange_rate_key)
);

-- ETL: Apply rate at time of sale
INSERT INTO fact_international_sales
SELECT ..., r.exchange_rate_key, ...,
       s.local_amount,
       s.currency,
       s.local_amount * r.exchange_rate,
       ...
FROM source_sales s
JOIN dim_exchange_rate r ON s.currency = r.from_currency
  AND s.sale_date BETWEEN r.effective_date AND r.expiry_date;

-- Query: Global revenue comparison (all USD)
SELECT p.product_name, SUM(f.reporting_amount) as total_usd_revenue
FROM fact_international_sales f
JOIN dim_product p ON f.product_key = p.product_key
GROUP BY p.product_name;
```

# Late-Arriving Dimensions (Inferred Members)

A fact references a dimension that doesn't exist yet. Create an 'inferred' placeholder row, then backfill when the real dimension data arrives.

```
-- Inferred member pattern
INSERT INTO dim_customer
VALUES (NEXTVAL('dim_customer_key_seq'), 'CUST-99999', 'Inferred Customer',
        NULL, NULL, NULL, NULL, NULL, 'Inferred', TRUE, CURRENT_DATE, '9999-12-31',
CURRENT_TIMESTAMP);

-- Later, backfill
UPDATE dim_customer
SET customer_name = 'John Doe',
    customer_address = '123 Main St',
    ...
WHERE customer_natural_id = 'CUST-99999'
  AND customer_name = 'Inferred Customer';

-- For Type 2, close inferred row and insert final row instead
UPDATE dim_customer SET is_current = FALSE, expiry_date = CURRENT_DATE - 1
WHERE customer_natural_id = 'CUST-99999' AND is_current = TRUE;

INSERT INTO dim_customer
VALUES (NEXTVAL('dim_customer_key_seq'), 'CUST-99999', 'John Doe',
        '123 Main St', 'Boston', 'MA', '02101', NULL, 'Active', TRUE, CURRENT_DATE, '9999-12-31',
CURRENT_TIMESTAMP);
```

# Aggregate Fact Tables for Performance

Aggregate facts are pre-calculated summaries at a coarser grain. If your transaction fact has billions of rows, a monthly aggregate by product and store might have millions. Query the aggregate for speed.

```
-- Atomic (detail) fact
CREATE TABLE fact_transaction (
    transaction_key BIGINT PRIMARY KEY,
    date_key INT,
    product_key INT,
    store_key INT,
    quantity INT,
    amount DECIMAL(12,2)
);  -- Billions of rows

-- Aggregate fact: One row per product per store per month
CREATE TABLE fact_transaction_monthly_agg (
    date_key INT,                          -- Month-end date
    product_key INT,
    store_key INT,
    -- Shrunken dimension keys (only those in this aggregate)
    -- No customer_key, transaction_key, etc.
    quantity_sold INT,
    amount DECIMAL(12,2),
    transaction_count INT,
    PRIMARY KEY (date_key, product_key, store_key)
);  -- Millions of rows

-- Load aggregate from atomic fact
INSERT INTO fact_transaction_monthly_agg
SELECT
    LAST_DAY(d.date) as date_key,  -- Month-end
    f.product_key,
    f.store_key,
    SUM(f.quantity) as quantity_sold,
    SUM(f.amount) as amount,
    COUNT(*) as transaction_count
FROM fact_transaction f
JOIN dim_date d ON f.date_key = d.date_key
WHERE d.year = 2024 AND d.month = 1
GROUP BY d.year, d.month, f.product_key, f.store_key;

-- Query aggregate (much faster)
SELECT p.product_name, s.store_name,
       SUM(a.quantity_sold) as total_qty,
       SUM(a.amount) as total_revenue
FROM fact_transaction_monthly_agg a
JOIN dim_product p ON a.product_key = p.product_key
JOIN dim_store s ON a.store_key = s.store_key
WHERE a.date_key >= '2024-01-31'
GROUP BY p.product_name, s.store_name;

-- BI tools use aggregate navigation to route queries to the aggregate automatically
```

# Audit Dimensions: Tracking Data Lineage

Add an audit dimension to track which ETL batch loaded the fact, when, and from what source system. Useful for troubleshooting stale data and auditing.

```
CREATE TABLE dim_audit (
    audit_key INT PRIMARY KEY,
    etl_batch_id VARCHAR(50),
    source_system VARCHAR(50),
    load_timestamp TIMESTAMP,
    load_date DATE,
    load_hour INT,
    row_count INT,
    error_count INT,
    etl_process_name VARCHAR(100),
    etl_server VARCHAR(50)
);

-- Fact table includes audit_key
CREATE TABLE fact_sales (
    sales_key BIGINT,
    date_key INT,
    product_key INT,
    ...
    audit_key INT REFERENCES dim_audit(audit_key)
);

-- Query: Which batch loaded this data? Is it stale?
SELECT a.etl_batch_id, a.source_system, a.load_timestamp, a.row_count
FROM fact_sales f
JOIN dim_audit a ON f.audit_key = a.audit_key
GROUP BY a.etl_batch_id, a.source_system, a.load_timestamp, a.row_count
ORDER BY a.load_timestamp DESC
LIMIT 1;
```

## Correction Transactions: Reversals and Adjustments

When a fact is incorrect, don't delete. Insert a reversal (negative amount) and optionally the correction.

```
-- Original transaction (wrong amount)
INSERT INTO fact_sales VALUES (1001, 20240101, ..., 500.00);

-- Reversal
INSERT INTO fact_sales VALUES (1002, 20240101, ..., -500.00);

-- Correction
INSERT INTO fact_sales VALUES (1003, 20240101, ..., 450.00);

-- Query result: 500.00 - 500.00 + 450.00 = 450.00
-- Audit trail preserved; corrections visible to analysts
```

## Advanced Summary

Advanced patterns address real-world complexity: heterogeneous products (JSON columns), multi-currency (dual amounts + exchange rates), late arrivals (inferred members), performance (aggregates), and auditability (audit dimensions). None are required for basic models, but all are essential for enterprise warehouses.

# dbt — Complete Fundamentals Deep Dive

I write SQL, argue about naming conventions, and explain to people why you can't just JOIN everything together. This is dbt: the tool that made me bearable to work with. dbt turns 'I ran some SQL scripts' into 'I built a production data pipeline with lineage, testing, and documentation.' Whether you're in Snowflake, BigQuery, Redshift, or Postgres, dbt's the same. It's the part that makes SQL feel like actual software engineering instead of a folder of scripts nobody understands.

## dbt-1: What Is dbt and Why It Exists

### The Problem dbt Solves

For decades, ETL meant: extract from sources, transform in some external tool (Informatica? Talend? A Python script that nobody understands?), load into the warehouse. The transformation code lived outside the warehouse, wasn't version controlled, and when something broke, you'd spend 3 hours reading spaghetti code trying to find which script is supposed to run after which other script.

Then cloud warehouses got stupid-powerful. Snowflake, BigQuery, Redshift: they can handle massive scale. So the obvious question became: why are we transforming outside the warehouse? Why not load raw, then transform in SQL (where the data lives)? ELT: Extract, Load raw, Transform inside. Sounds simple. But actually building an ELT pipeline that doesn't suck? That's where dbt enters.

Before dbt, ELT meant: 'I have 47 SQL files in a folder. Run them in this order. Actually, I don't know what order. Let me check the comments... the comments are from 2018 and wrong. Let me run it and see what breaks.' Version control doesn't help; SQL files don't track dependencies. dbt fixes this: models reference each other explicitly, dbt builds a DAG, everything runs in the right order, and you can trace exactly which data depends on which.

### The ELT Revolution

ELT (Extract-Load-Transform) moves the transformation workload to the warehouse itself:

```
Traditional ETL:
Source Systems → [Transform in Staging Tool] → Data Warehouse → BI Tool

Modern ELT (dbt approach):
Source Systems → Data Warehouse (raw layer) → [Transform inside warehouse with dbt] → BI Tool
```

Benefits of ELT over ETL:

- Transformation runs in the warehouse where data lives (no data movement)
- SQL is the lingua franca of data (everyone knows it, version control works perfectly)
- Warehouse compute is cheaper than external transformation tools
- Debugging is straightforward: query the warehouse directly
- Data lineage is automatically tracked via dbt DAG

### What dbt Actually Does

dbt's trick: take your SQL + YAML + variables, compile Jinja templates, execute against your warehouse:

- SQL files (your data models)
- YAML configuration (sources, tests, descriptions)
- Jinja templates (variables, macros, logic)

- Output: compiled raw SQL executed against your warehouse

```
Input (you write):
{{ source('raw', 'customers') }}
SELECT * FROM {{ source('raw', 'customers') }}

Output (dbt compiles and executes):
SELECT * FROM raw_db.public.customers
```

dbt also does four critical things:

- Dependency management: executes models in the correct order (the DAG)
- Data quality testing: runs tests and fails builds if data is bad
- Documentation: generates an interactive site showing tables, columns, and lineage
- Environment abstraction: dev/test schemas, different databases, different warehouses

## dbt Core vs dbt Cloud

dbt comes in two flavors. dbt Core is open-source. dbt Cloud is the SaaS product built on top of dbt Core.

| Feature | dbt Core (Open Source) | dbt Cloud (SaaS) |
|---|---|---|
| Cost | Free | Paid ($25-2000+/month) |
| Installation | pip install dbt-snowflake | Web UI, managed infrastructure |
| Scheduling | Manual or cron/Airflow | Built-in job scheduler |
| IDE | Your text editor + terminal | Web-based dbt IDE |
| CI/CD | GitHub Actions, custom | Built-in dbt Cloud CI/CD |
| Collaboration | Git-based reviews | Team workspace + Git |
| Monitoring | Manual logging | Built-in run history and alerts |
| Documentation | Manual dbt docs serve | Hosted docs.getdbt.com |
| Lineage | Local dbt docs | Interactive web lineage |
| Multi-warehouse | Any (you manage) | Any (cloud manages) |
| Artifact storage | Local filesystem | dbt Cloud bucket |
| Debugging | CLI output | Web UI + Cloud IDE |
| Best for | Learning, solo dev | Production teams |
| Setup time | 15-30 minutes | 5 minutes (no setup) |

*Start with dbt Core locally to learn. When your team grows, migrate to dbt Cloud for collaboration, scheduling, and monitoring. Both share the same core concepts and commands.*

## How dbt Works: Step-by-Step

Understanding dbt's process is essential. Here's what happens when you run dbt run:

- Step 1: Parse — Read all .sql and .yml files, understand project structure
- Step 2: Build DAG — Figure out dependencies using ref() and source() macros
- Step 3: Compile — Replace Jinja variables/macros with actual values, produce SQL
- Step 4: Execute — Run compiled SQL against warehouse in dependency order
- Step 5: Return artifacts — Create manifest.json, run_results.json, updated docs

```
# Step 1: You write this in models/staging/stg_customers.sql
SELECT
  customer_id,
  lower(customer_name) as customer_name,
  created_at
FROM {{ source('raw', 'customers') }}
```

```
WHERE deleted_at IS NULL

# Step 3: dbt compiles to this SQL (shown with dbt compile)
SELECT
  customer_id,
  lower(customer_name) as customer_name,
  created_at
FROM raw_db.public.customers
WHERE deleted_at IS NULL

# Step 4: dbt executes the compiled SQL against Snowflake/BigQuery/etc
```

## The DAG: Dependency Acyclic Graph

The DAG is dbt's mental model of your project. It's a graph where each node is a model, test, or source, and edges represent dependencies (ref() and source() calls). dbt topologically sorts this graph and executes in dependency order. This is why you never need to worry about 'run this model first'—dbt figures it out.

```
sources (raw data, never created by dbt)
    ↓
    ■■ stg_customers (SELECT FROM source)
    ■■ stg_orders (SELECT FROM source)
    ↓
intermediate models (joins, aggregations)
    ■■ int_customer_metrics (FROM stg_customers)
    ■■ int_order_details (FROM stg_orders)
    ↓
mart models (final business tables)
    ■■ fct_orders (FROM int_order_details)
    ■■ dim_customers (FROM int_customer_metrics)
```

The cooking analogy: chop vegetables → cook them → plate the dish → serve. You must chop before cooking, cook before plating. The DAG ensures this order automatically.

## Installation: Setting Up dbt Core

Install dbt Core for your specific warehouse:

```
# For Snowflake (most popular in enterprise)
pip install dbt-snowflake

# For BigQuery (popular in Google Cloud)
pip install dbt-bigquery

# For Postgres (great for learning locally)
pip install dbt-postgres

# For Redshift
pip install dbt-redshift

# For DuckDB (local, no warehouse needed)
pip install dbt-duckdb
```

Verify installation:

```
$ dbt --version
Core version 1.7.0
```

## dbt init: Scaffolding Your First Project

Initialize a new dbt project:

```
dbt init my_analytics_project
```

dbt will ask you questions (warehouse type, authentication details). Once complete, you'll have:

```
my_analytics_project/
■■■ dbt_project.yml             # Project configuration (name, version, paths)
■■■ profiles.yml (in ~/.dbt/)   # Warehouse credentials (do NOT commit to git)
■■■ models/                     # Your SQL transformation files
■■■ tests/                      # Data quality tests
■■■ macros/                     # Reusable SQL functions (Jinja macros)
■■■ seeds/                      # Static CSV data files
■■■ analyses/                   # Ad-hoc queries (not models)
■■■ snapshots/                  # SCD Type 2 configurations
■■■ packages.yml                # External package dependencies
■■■ README.md                   # Project documentation
```

## profiles.yml: Warehouse Credentials

Located at ~/.dbt/profiles.yml (not in your project, never committed to git). This file stores connections to your warehouse(s). Each profile defines outputs (dev, prod, etc). dbt uses the target to determine which connection to use.

Example: Snowflake

```
my_analytics_project:
  outputs:
    dev:
      type: snowflake
      account: xy12345.us-east-1  # Your account ID
      user: {{ env_var('DBT_USER') }}  # Get from environment
      password: {{ env_var('DBT_PASSWORD') }}
      database: analytics_dev
      schema: dbt_dev        # Your working schema
      warehouse: compute_wh
      threads: 4             # Parallel query execution
      client_session_keep_alive: false

    prod:
      type: snowflake
      account: xy12345.us-east-1
      user: dbt_service_account
      password: {{ env_var('PROD_PASSWORD') }}
      database: analytics
      schema: analytics        # Production schema
      warehouse: prod_wh
      threads: 8

  target: dev  # Use 'dev' by default (override with dbt run --target prod)
```

Example: BigQuery

```
my_analytics_project:
  outputs:
    dev:
      type: bigquery
      project: my-gcp-project
      dataset: analytics_dev
      location: us-east1
      threads: 4
      timeout_seconds: 300
      keyfile: ~/.dbt/keyfile.json  # Service account key
      method: service-account
```

```
  target: dev
```

*Never hardcode passwords in profiles.yml. Use environment variables with {{ env_var('VAR_NAME') }} and set them in your shell or CI/CD platform.*

## dbt_project.yml: Project Configuration

This file defines project-wide settings and model configurations. Located in your project root.

```
name: 'my_analytics_project'
version: '1.0.0'
config-version: 2
profile: 'my_analytics_project'

# Where are the code files?
model-paths: ['models']
analysis-paths: ['analyses']
test-paths: ['tests']
macro-paths: ['macros']
seed-paths: ['seeds']
snapshot-paths: ['snapshots']
resource-paths: ['resources']

# Where to put generated files
target-path: 'target'      # dbt outputs here
cleaning-paths: ['target', 'dbt_packages']

# How to build models by default (can override per-model)
models:
  my_analytics_project:
    # Staging: lightweight views (1:1 with sources)
    staging:
      materialized: view
      schema: staging

    # Intermediate: business logic (usually ephemeral)
    intermediate:
      materialized: ephemeral

    # Marts: final tables
    marts:
      materialized: table
      schema: analytics
      post-hook:
        - "GRANT SELECT ON {{ this }} TO ROLE analyst;"

# Variables (default values, override with --vars)
vars:
  min_date: '2020-01-01'
  dbt_lookback_days: 90
```

*The dbt_project.yml hierarchy is powerful. Set defaults at the root level, override at the folder level, and override again at the model level with config() blocks.*

## Your First Model: Complete Example

Create models/staging/stg_customers.sql:

```
-- models/staging/stg_customers.sql
-- Purpose: One row per customer, cleaned attributes from raw.customers
```

```
-- Grain: customer_id

{{ config(
  materialized='view',
  schema='staging',
  tags=['daily']
) }}

SELECT
  customer_id,
  lower(trim(customer_name)) as customer_name,
  lower(trim(email)) as email,
  created_at,
  current_timestamp as dbt_loaded_at
FROM {{ source('raw', 'customers') }}
WHERE deleted_at IS NULL
```

Key parts:

- {{ config(...) }} — Config block sets materialization, schema, tags
- {{ source('raw', 'customers') }} — References external raw table (declared in sources.yml)
- trim(), lower() — Data cleaning
- current_timestamp — Audit column showing when data was loaded

## Running Your First Model

Execute dbt:

```
$ cd my_analytics_project
$ dbt run

14:32:45  Running with dbt=1.7.0
14:32:47  Found 1 model, 0 tests
14:32:48  Executing stg_customers... [1/1]
14:32:50  ✓ SUCCESS: created view stg_customers
14:32:50  Finished successfully
```

What dbt did:

- Compiled the Jinja template ({{ source(...) }}) to raw SQL
- Executed CREATE VIEW stg_customers AS SELECT ...
- Created the view in your dev schema (from profiles.yml)
- Stored metadata in target/manifest.json (the DAG)

## Viewing the Compiled SQL

See exactly what SQL dbt ran:

```
$ dbt compile

# Then check target/compiled/my_analytics_project/models/staging/stg_customers.sql

SELECT
  customer_id,
  lower(trim(customer_name)) as customer_name,
  lower(trim(email)) as email,
  created_at,
  current_timestamp as dbt_loaded_at
FROM analytics.raw.customers  # dbt replaced {{ source(...) }} with actual location
```

```
WHERE deleted_at IS NULL
```

## Core dbt Commands in Depth

| Command | What It Does | When to Use |
|---|---|---|
| dbt run | Execute all models in DAG order | Build transformations |
| dbt test | Run all tests on models/sources | Validate data quality |
| dbt build | dbt run + dbt test (modern best practice) | CI/CD pipelines, production |
| dbt compile | Generate SQL without executing | Review compiled SQL, CI checks |
| dbt debug | Check warehouse connection and paths | Troubleshoot setup issues |
| dbt deps | Install packages from packages.yml | Once per new package |
| dbt seed | Load CSV seeds into warehouse | Load lookup tables |
| dbt snapshot | Execute snapshots for SCD Type 2 | Track historical changes |
| dbt docs generate | Create documentation site | After model changes |
| dbt docs serve | Serve docs at localhost:8000 | Browse documentation locally |
| dbt source freshness | Check if sources are up-to-date | Monitor data ingestion |
| dbt run --select stg_* | Run only staging models (selection) | Partial builds |
| dbt run --models fct_orders | Run model and downstream | Rebuild fact table |
| dbt run --exclude dim_* | Run everything except dimensions | Skip expensive models |
| dbt run --full-refresh | Rebuild incremental tables from scratch | Fix data issues |
| dbt test --fail-fast | Stop on first test failure | Quick validation |
| dbt parse | Parse project without connecting warehouse | Check syntax |

## The ref() Macro: Most Important Concept

ref() is the foundation of dbt. It references another dbt model and creates a dependency in the DAG. ref() is warehouse-agnostic and environment-aware.

```
-- In any model, use ref() to reference another model
SELECT *
FROM {{ ref('stg_customers') }}
JOIN {{ ref('stg_orders') }}
  ON stg_customers.customer_id = stg_orders.customer_id
```

How ref() compiles differently in dev vs prod:

```
# Dev (dbt run --target dev):
# profiles.yml says schema: dbt_dev
SELECT * FROM analytics_dev.dbt_dev.stg_customers

# Prod (dbt run --target prod):
# profiles.yml says schema: analytics
SELECT * FROM analytics.analytics.stg_customers
```

Why ref() is revolutionary:

- Dependency tracking — dbt knows which models depend on which
- Environment awareness — same code works in dev/test/prod
- Automatic ordering — no need to manually order model execution
- Refactoring safety — rename a model, update all references automatically (with dbt rename)

## source() Macro: Declaring External Data

Use source() to reference external tables that dbt doesn't create. This enables freshness checks and lineage.

```
-- In a model
SELECT * FROM {{ source('raw', 'customers') }}

-- Compiles to:
SELECT * FROM raw_db.public.customers
```

Difference between ref() and source():

| Macro | What It Tracks | Use For |
|---|---|---|
| ref() | dbt models (tables/views you create) | Joining transformed data |
| source() | External tables (not created by dbt) | Raw data from sources |

# dbt-2: Sources, Seeds, and Documentation

## Sources: Declaring External Data

Sources are external tables in your warehouse that dbt reads from but doesn't create. Declaring them in YAML gives dbt awareness of your data lineage, enables freshness monitoring, and provides a place to document raw data.

```
# models/staging/_sources.yml
version: 2

sources:
  - name: raw
    description: 'Raw data from Shopify and internal systems'
    database: raw_db
    schema: public
    tables:
      - name: customers
        description: 'Shopify customer records'
        columns:
          - name: customer_id
            description: 'Unique customer ID'
          - name: customer_name
            description: 'Full name'
      - name: orders
        description: 'Shopify orders'
        columns:
          - name: order_id
            description: 'Unique order ID'

  - name: stripe
    description: 'Stripe payment events'
    database: raw_db
    schema: stripe_events
    tables:
      - name: payments
        description: 'Payment transactions'
```

Then reference sources in models with the source() macro:

```
-- models/staging/stg_customers.sql
SELECT
  customer_id,
  customer_name,
  created_at
FROM {{ source('raw', 'customers') }}
WHERE deleted_at IS NULL
```

## Source Freshness Checks

Monitor if your raw data is being refreshed on schedule. Define freshness thresholds in sources.yml:

```
sources:
  - name: raw
    freshness:
      warn_after: {count: 12, period: hour}
      error_after: {count: 24, period: hour}
    tables:
      - name: customers
        loaded_at_field: created_at  # Which column tracks when data arrived
      - name: orders
        loaded_at_field: updated_at
```

Check freshness with: dbt source freshness

```
$ dbt source freshness

14:32:45  Running with dbt=1.7.0
14:32:47  WARNING: source raw.customers freshness check failed
14:32:47  Last update: 2024-01-15 08:00:00 (5 hours ago)
14:32:47  ERROR: source raw.orders freshness check failed
14:32:47  Last update: 2024-01-14 10:00:00 (26 hours ago)
```

## Seeds: CSV Lookup Tables

Seeds are CSV files in your dbt project that get seeded (loaded) into the warehouse. Use seeds for small, static reference data that changes rarely. Never use for large datasets.

```
# seeds/country_codes.csv
country_code,country_name,region
US,United States,North America
CA,Canada,North America
MX,Mexico,North America
GB,United Kingdom,Europe
DE,Germany,Europe
FR,France,Europe
JP,Japan,Asia
CN,China,Asia
IN,India,Asia

# seeds/product_categories.csv
product_id,category,subcategory
1,Electronics,Phones
2,Electronics,Laptops
3,Clothing,Shirts
4,Clothing,Pants
```

Load seeds once (or after updates):

```
$ dbt seed

14:32:45  Running with dbt=1.7.0
14:32:47  Loading seed country_codes.csv
14:32:48  ✓ Created table country_codes (9 rows)
14:32:48  Loading seed product_categories.csv
14:32:49  ✓ Created table product_categories (4 rows)
```

*Never use seeds for:*

- Large datasets (>100k rows) — will bloat your dbt directory
- Frequently changing data — defeats version control purpose
- Data you generate (use dbt models instead)
- Secrets or credentials — never commit these

## Documentation: Living with Code

dbt's documentation paradigm is revolutionary: descriptions live in .yml files next to code, not in external wikis that get stale. This keeps documentation in sync with actual data.

```
# models/staging/_stg_sources.yml
version: 2

models:
  - name: stg_customers
```

```
    description: 'One row per customer with cleaned attributes from raw.customers'
    columns:
      - name: customer_id
        description: 'Unique customer identifier, primary key'
        tests:
          - unique
          - not_null
      - name: customer_name
        description: 'Customer full name, trimmed and lowercased'
      - name: email
        description: 'Customer email, lowercased for consistency'
```

## Doc Blocks: Long Descriptions

For longer descriptions, use doc blocks ({% docs ... %}). Create a separate file or add to schema.yml:

```
# models/staging/_stg_sources.yml
{% docs customer_id %}
The unique customer identifier assigned by our Shopify instance.
Generated sequentially starting from ID 1 in January 2020.
Used as the primary key in all customer-related tables.
Never NULL, always positive integer.
{% enddocs %}

columns:
  - name: customer_id
    description: '{{ doc("customer_id") }}'
```

## Generating and Serving Documentation

Generate documentation:

```
$ dbt docs generate

14:32:45  Running with dbt=1.7.0
14:32:48  Generating documentation
14:32:50  ✓ Docs generated successfully
```

Serve documentation locally:

```
$ dbt docs serve

14:32:50  Serving docs at http://127.0.0.1:8000

# Open browser to http://localhost:8000
```

You'll see:

- Project overview with READMEs
- Model browser (searchable)
- Column documentation with data types and tests
- Interactive DAG showing dependencies
- Test history and execution times

## YAML File Organization Strategies

Two approaches to organizing .yml files:

Option 1: One file per folder (recommended for large projects)

```
models/
■■■ staging/
■   ■■■ stg_customers.sql
■   ■■■ stg_orders.sql
■   ■■■ _stg_sources.yml          # All staging docs here
■■■ intermediate/
■   ■■■ int_customer_metrics.sql
■   ■■■ _int_models.yml           # All intermediate docs
■■■ marts/
    ■■■ fct_orders.sql
    ■■■ dim_customers.sql
    ■■■ _mart_models.yml          # All mart docs
```

Option 2: One file per model (simpler for small projects)

```
models/
■■■ stg_customers.sql
■■■ stg_customers.yml            # Docs for this model
■■■ stg_orders.sql
■■■ stg_orders.yml               # Docs for this model
■■■ fct_orders.sql
■■■ fct_orders.yml
```

*Pick one convention and stick with it. Consistency beats perfection.*

# dbt-3: Materializations — The Complete Guide

## What Are Materializations?

Materializations define how dbt persists your model results in the warehouse. They're one of dbt's most powerful features. Different materializations suit different use cases: staging needs lightweight views, marts need performant tables, slowly-changing data needs snapshots. Configure materializations in dbt_project.yml or override per-model with config() blocks.

## View: CREATE VIEW (Lightweight, Always Fresh)

A view is a stored query that executes every time you reference it. No data is stored. Lightweight and always reflects the latest source data, but can be slow downstream.

```
-- models/staging/stg_customers.sql
{{ config(materialized='view') }}

SELECT
  customer_id,
  trim(customer_name) as customer_name,
  lower(email) as email,
  created_at
FROM {{ source('raw', 'customers') }}
WHERE deleted_at IS NULL

-- dbt creates:
CREATE VIEW analytics.staging.stg_customers AS
SELECT ...

-- When you query the view:
SELECT * FROM stg_customers
-- dbt executes the underlying SELECT each time
```

When to use views:

- Staging models (1:1 with sources, minimal processing)
- Rarely accessed models
- Development environments (fast iteration)
- Small datasets
- Models that should always reflect current source data

Avoid views when:

- Many downstream models reference the view (slow queries)
- Expensive transformations (heavy joins, aggregations)
- The underlying query is run frequently

## Table: CREATE TABLE AS SELECT (Materialized Data)

A table is materialized data stored in the warehouse. Data is static until the next rebuild. Very fast to query downstream, but uses storage and becomes stale.

```
-- models/marts/fct_orders.sql
{{ config(materialized='table') }}

SELECT
  order_id,
  customer_id,
```

```
  sum(order_total) as total_spent,
  count(*) as order_count,
  max(order_date) as last_order_date
FROM {{ ref('stg_orders') }}
GROUP BY 1, 2

-- dbt creates:
CREATE TABLE analytics.marts.fct_orders AS
SELECT ...

-- Subsequent queries are fast (just table scan):
SELECT * FROM fct_orders WHERE customer_id = 123
```

When to use tables:

- Mart models (final consumer-facing data)
- Frequently accessed by many downstream queries
- Expensive calculations (joins, aggregations)
- Models where slightly stale data is acceptable

When NOT to use:

- Very large datasets with frequent runs (cost, time)
- Data that must be always fresh
- Intermediate models that are rarely accessed

## Incremental: Only Process New Data

The most powerful materialization for large datasets. Instead of rebuilding the entire table, incremental models only process new data since the last run. This is the key to scaling dbt.

```
-- models/marts/fct_events.sql
{{ config(
  materialized='incremental',
  unique_key='event_id',
  on_schema_change='fail'
) }}

SELECT
  event_id,
  user_id,
  event_timestamp,
  event_type,
  event_data
FROM {{ source('raw', 'events') }}

{% if is_incremental() %}
  -- Only process events since the last run
  WHERE event_timestamp > (SELECT max(event_timestamp) FROM {{ this }})
{% endif %}
```

How is_incremental() works:

- Returns TRUE when the table exists AND this is not a --full-refresh
- Returns FALSE on the first run (table doesn't exist yet)
- Use {% if is_incremental() %} to conditionally add WHERE clauses

## Incremental Strategy: Append (Simple, Fastest)

Append just inserts new rows. Best for immutable events (events can't be deleted/updated).

```
{{ config(materialized='incremental', unique_key='event_id', on_schema_change='ignore') }}

SELECT
  event_id,
  user_id,
  event_timestamp,
  event_type,
  event_data
FROM {{ source('raw', 'events') }}

{% if is_incremental() %}
  WHERE event_timestamp > (SELECT max(event_timestamp) FROM {{ this }})
{% endif %}
```

Use when: Event logs, clickstream data, immutable transaction records

## Incremental Strategy: Merge (Deduplicating, Flexible)

Merge updates existing rows (by unique_key) and inserts new rows. Deduplicates automatically.

```
{{ config(
  materialized='incremental',
  unique_key='customer_id',
  incremental_strategy='merge',
  on_schema_change='sync_all_columns'
) }}

SELECT
  customer_id,
  customer_name,
  email,
  updated_at
FROM {{ source('raw', 'customers') }}

{% if is_incremental() %}
  WHERE updated_at >= (SELECT max(updated_at) FROM {{ this }})
{% endif %}
```

Use when: Customer data, slowly changing dimensions, fact tables with updates

```
-- Compiled MERGE for Snowflake:
MERGE INTO analytics.marts.dim_customers t
USING (
  SELECT * FROM new_data
) s
ON t.customer_id = s.customer_id
WHEN MATCHED THEN UPDATE SET t.* = s.*
WHEN NOT MATCHED THEN INSERT *
```

## Incremental Strategy: Delete+Insert (Partition Refresh)

Delete matching rows, then insert new data. Useful for recalculating recent partitions.

```
{{ config(
  materialized='incremental',
  unique_key=['order_id', 'order_date'],
  incremental_strategy='delete+insert'
) }}

SELECT
```

```
  order_id,
  customer_id,
  order_date,
  order_total
FROM {{ source('raw', 'orders') }}

{% if is_incremental() %}
  -- Refresh last 3 days (in case late-arriving data)
  WHERE order_date >= current_date - interval '3 days'
{% endif %}
```

Use when: Daily fact tables where late-arriving data is common

## Incremental: on_schema_change Options

What happens when you add/remove columns in an incremental model:

| Option | Behavior | Use When |
|---|---|---|
| ignore | Do nothing, columns out of sync | Rarely (dangerous) |
| append_new_columns | Add new columns with NULL | Non-critical columns |
| sync_all_columns | Sync all columns every time | Safe, but slower |
| fail | Error and stop build | Strict quality (recommended) |

## Incremental: Full Refresh Escape Hatch

Sometimes you need to rebuild an incremental table from scratch:

```
# Rebuild fct_orders from scratch:
$ dbt run --select fct_orders --full-refresh

# Rebuild ALL incremental models:
$ dbt run --full-refresh
```

When to use --full-refresh:

- Fix data quality issues
- Change unique_key or aggregation logic
- Add new source tables to the model
- After warehouse maintenance

## Ephemeral: CTE Injection (No Storage)

Ephemeral models compile as Common Table Expressions (CTEs) in downstream models. They exist only during compilation, using no storage. Perfect for reusable intermediate logic.

```
-- models/intermediate/int_customer_metrics.sql
{{ config(materialized='ephemeral') }}

SELECT
  customer_id,
  count(*) as total_orders,
  sum(order_total) as lifetime_value,
  avg(order_total) as avg_order_value,
  max(order_date) as last_order_date
FROM {{ ref('stg_orders') }}
GROUP BY 1
```

When you reference this ephemeral model:

```
-- models/marts/dim_customers.sql
SELECT
  c.customer_id,
  c.customer_name,
  m.total_orders,
  m.lifetime_value
FROM {{ ref('stg_customers') }} c
LEFT JOIN {{ ref('int_customer_metrics') }} m
  ON c.customer_id = m.customer_id

-- dbt compiles the ephemeral model as a CTE:
WITH int_customer_metrics AS (
  SELECT customer_id, count(*) as total_orders, ...
  FROM stg_orders
  GROUP BY 1
)
SELECT
  c.customer_id,
  c.customer_name,
  m.total_orders,
  m.lifetime_value
FROM stg_customers c
LEFT JOIN int_customer_metrics m
  ON c.customer_id = m.customer_id
```

When to use ephemeral:

- Reusable calculations referenced by multiple models
- Intermediate logic that shouldn't be tested separately
- Complex joins/aggregations that reduce clutter when inlined as CTEs

## Snapshot: Track Changes Over Time (SCD Type 2)

Snapshots track how data changes over time. They implement Slowly Changing Dimension Type 2 (SCD Type 2): instead of updating rows, they insert new rows with valid_from/valid_to timestamps.

```
-- snapshots/snap_customers.sql
{% snapshot snap_customers %}
  {{config(
    target_schema='snapshots',
    unique_key='customer_id',
    strategy='timestamp',
    updated_at='updated_at'
  )}}

  SELECT
    customer_id,
    customer_name,
    email,
    account_status,
    updated_at
  FROM {{ source('raw', 'customers') }}

{% endsnapshot %}
```

Run snapshots:

```
$ dbt snapshot

14:32:45  Running with dbt=1.7.0
14:32:48  Executing snap_customers
14:32:50  ✓ Created snapshot table snap_customers (1000 rows)
```

dbt generates these columns automatically:

```
customer_id | customer_name | email           | account_status | dbt_valid_from | dbt_valid_to
1           | John Doe      | john@example    | active         | 2024-01-01     | 2024-01-10
1           | John Doe      | john@example    | inactive       | 2024-01-10     | null
```

Query the snapshot to see current values:

```
SELECT * FROM snapshots.snap_customers
WHERE dbt_valid_to IS NULL
```

## Materializations Comparison Table

| Type | Storage | Query Speed | Freshness | Use For |
|---|---|---|---|---|
| View | None | Slow (re-query) | Always fresh | Staging |
| Table | High | Very fast | Stale until rebuild | Marts, BI |
| Incremental | Medium | Very fast | Updated per run | Large facts |
| Ephemeral | None (CTE) | N/A (inlined) | Fresh | Intermediate |
| Snapshot | High | Fast | Historical tracking | SCD Type 2 |

## Materialization Decision Guide

Use this flowchart to decide which materialization to use:

```
Is this a staging model (1:1 with source)?
  YES → View
  NO → Is it referenced by multiple models?
    YES → Is it expensive to compute?
      YES → Table (create once, reuse)
      NO → Ephemeral (inline as CTE)
    NO → Is this very large (>100M rows)?
      YES → Incremental (append/merge)
      NO → Table (simpler)

Do you need to track changes over time?
  YES → Snapshot
  NO → (use above flowchart)
```

# dbt-4: Jinja & Macros — dbt's Power Engine

## What is Jinja?

Jinja is a templating language that lets you use variables, loops, and conditionals in SQL. dbt compiles your Jinja+SQL into raw SQL, then executes it. This transforms static SQL into dynamic, reusable transformations. Jinja is the 'glue' that makes dbt powerful.

```
# Example of Jinja in SQL:
SELECT
  customer_id,
  customer_name,
  {% if target.name == 'prod' %}
    email,
    phone
  {% endif %}
FROM {{ source('raw', 'customers') }}
```

## Jinja Syntax Basics

Three types of Jinja delimiters:

```
{{ expression }}     # Output: evaluates and renders
{# comment #}       # Comment: ignored
{% statement %}    # Statement: control flow (if/for/etc)
```

Whitespace control (prevents extra newlines):

```
{%- if condition %} ... {% endif -%}  # Removes whitespace around the block
```

## Variables: var() Function

Access project variables with var('name'). Define defaults in dbt_project.yml or override at runtime.

```
# In dbt_project.yml:
vars:
  min_date: '2020-01-01'
  env_name: 'dev'

# In a model:
SELECT *
FROM {{ source('raw', 'customers') }}
WHERE created_at >= '{{ var("min_date") }}'

# Override at runtime:
$ dbt run --vars '{min_date: "2024-01-01"}'

# Using dbt CLI:
$ dbt run --vars 'env_name: prod'
```

## Built-in Variables

| Variable | Description | Example |
|---|---|---|
| {{ target.name }} | Profile target (dev/prod/test) | if target.name == 'prod' |
| {{ target.schema }} | Target schema from profiles.yml | from_schema = target.schema |
| {{ target.database }} | Target database | from_db = target.database |
| {{ target.threads }} | Number of parallel threads | {{ target.threads }} |
| {{ this }} | Current model (table/view) | max(col) FROM {{ this }} |

| Variable | Description | Example |
|---|---|---|
| {{ execute }} | Is Jinja executing? (True/False) | {% if execute %} ... {% endif %} |
| {{ env_var('VAR') }} | Environment variable | user = {{ env_var('DBT_USER') }} |
| {{ run_started_at }} | When dbt run started | started = {{ run_started_at }} |
| {{ source(...) }} | Reference external table | FROM {{ source('raw', 'tbl') }} |
| {{ ref(...) }} | Reference dbt model | FROM {{ ref('stg_customers') }} |

## Control Flow: if/else

Conditionally generate SQL:

```
SELECT
  customer_id,
  customer_name
{% if target.name == 'prod' %}
  , email
  , phone
  , ip_address
{% else %}
  , 'MASKED' as email
{% endif %}
FROM {{ source('raw', 'customers') }}
```

Complex conditionals:

```
{% if var('run_type') == 'daily' %}
  WHERE date = current_date
{% elif var('run_type') == 'weekly' %}
  WHERE date >= current_date - 7
{% else %}
  WHERE 1=1
{% endif %}
```

## Control Flow: for Loops

Loop over lists to generate SQL dynamically:

```
-- Dynamically union multiple tables:
{% set tables = ['customers', 'employees', 'partners'] %}

{% for table in tables %}
SELECT
  '{{ table }}' as source_table,
  *
FROM {{ source('raw', table) }}
{% if not loop.last %}UNION ALL{% endif %}
{% endfor %}
```

Loop variables available in Jinja:

```
loop.index      # Current iteration (1-indexed)
loop.index0     # Current iteration (0-indexed)
loop.first      # True on first iteration
loop.last       # True on last iteration
loop.length     # Total number of iterations
```

## Macros: Reusable SQL Functions

Macros let you write reusable SQL logic. Define in macros/ folder, use anywhere.

```
-- macros/cents_to_dollars.sql
{% macro cents_to_dollars(column) %}
  round({{ column }} / 100.0, 2)
{% endmacro %}

-- Use in a model:
SELECT
  order_id,
  {{ cents_to_dollars('order_total_cents') }} as order_total_usd
FROM {{ source('raw', 'orders') }}
```

## Macro Example 1: Generate Surrogate Key

Create a unique hash from multiple columns (common pattern):

```
-- macros/generate_surrogate_key.sql
{% macro generate_surrogate_key(field_list) %}
md5(
  concat_ws('||',
  {% for field in field_list %}
    cast({{ field }} as string)
    {%- if not loop.last %},{% endif %}
  {% endfor %}
  )
)
{% endmacro %}

-- Use it:
SELECT
  {{ generate_surrogate_key(['customer_id', 'order_date', 'product_id']) }} as order_key,
  *
FROM {{ ref('stg_orders') }}
```

## Macro Example 2: Dynamic Audit Columns

Add standard audit columns consistently:

```
-- macros/audit_columns.sql
{% macro audit_columns() %}
  current_timestamp as dbt_created_at,
  current_timestamp as dbt_updated_at,
  '{{ run_started_at }}' as dbt_run_timestamp,
  '{{ target.name }}' as dbt_environment
{% endmacro %}

-- Use it:
SELECT
  customer_id,
  customer_name,
  {{ audit_columns() }}
FROM {{ source('raw', 'customers') }}
```

## Macro Example 3: Conditional Limit in Dev

Limit data in dev environment for faster iteration:

```
-- macros/limit_in_dev.sql
{% macro limit_in_dev(limit_count=100) %}
{% if target.name != 'prod' %}
LIMIT {{ limit_count }}
{% endif %}
{% endmacro %}
```

```
-- Use it:
SELECT *
FROM {{ source('raw', 'events') }}
{{ limit_in_dev(1000) }}
```

## Macro Example 4: Dynamic Column Selection

Select specific columns based on environment:

```
-- macros/select_columns.sql
{% macro select_columns(table, include_pii=false) %}
{% set cols = adapter.get_columns_in_relation(table) %}
{% for col in cols %}
  {% if include_pii or col.name not in ['ssn', 'credit_card', 'ip_address'] %}
    {{ col.name }}
    {%- if not loop.last %},{% endif %}
  {% endif %}
{% endfor %}
{% endmacro %}
```

## Macro Example 5: Union Multiple Tables

Dynamically union staging tables:

```
-- macros/union_tables.sql
{% macro union_tables(table_list) %}
{% for table in table_list %}
SELECT
  '{{ table }}' as source_table,
  *
FROM {{ ref(table) }}
{% if not loop.last %}UNION ALL{% endif %}
{% endfor %}
{% endmacro %}
```

## Common Jinja Patterns

| Pattern | Code Example |
|---|---|
| Null coalescing | coalesce(column, 0) |
| Safe boolean | cast({{ column }} as boolean) |
| Date filtering | WHERE date_col >= '{{ var("min_date") }}' |
| Multiple conditions | {% if a and b %} ... {% endif %} |
| List membership | {% if column in ['a', 'b', 'c'] %} |
| Environment check | {% if target.name == 'prod' %} |
| Iterate with index | {% for item in list %}{{ loop.index }} |
| Schema reference | {{ source('db', table_name) }} |
| CTE generation | WITH cte AS (SELECT ...) SELECT * FROM cte |

# dbt-5: Testing — Complete Guide

## Why Testing Matters in dbt

Data quality issues cascade downstream. A single NULL in a key field breaks dashboards. Bad joins create silent bugs (no error, just wrong numbers). dbt makes it trivial to run automated tests every build. The best data teams test relentlessly. dbt tests are: fast (SQL-based), version-controlled (in Git), and integrated (fail the build if data is bad).

## Generic Test: unique

Ensures a column has no duplicate values. Add to .yml:

```
models:
  - name: stg_customers
    columns:
      - name: customer_id
        tests:
          - unique
          - not_null
```

What it does:

```
SELECT COUNT(*) - COUNT(DISTINCT customer_id) as duplicates
FROM stg_customers
HAVING duplicates > 0
```

## Generic Test: not_null

Ensures no NULL values:

```
columns:
  - name: customer_id
    tests:
      - not_null
  - name: order_id
    tests:
      - not_null

-- Generates:
SELECT * FROM stg_customers WHERE customer_id IS NULL
```

## Generic Test: accepted_values

Ensures column only contains specific values:

```
columns:
  - name: order_status
    tests:
      - accepted_values:
          values: ['pending', 'shipped', 'delivered', 'cancelled', 'refunded']

-- Generates:
SELECT *
FROM fct_orders
WHERE order_status NOT IN ('pending', 'shipped', 'delivered', 'cancelled', 'refunded')
```

## Generic Test: relationships (Foreign Key)

Ensures referential integrity: every customer_id in orders exists in customers:

```
models:
  - name: fct_orders
    columns:
      - name: customer_id
        tests:
          - relationships:
              to: ref('dim_customers')
              field: customer_id

-- Generates:
SELECT *
FROM fct_orders o
LEFT JOIN dim_customers c
  ON o.customer_id = c.customer_id
WHERE c.customer_id IS NULL
```

## Singular Tests: Custom SQL

Write custom tests in tests/ folder. Convention: a query that returns rows = test failed.

```
-- tests/assert_order_totals_positive.sql
-- Test fails if any order has negative total
SELECT *
FROM {{ ref('fct_orders') }}
WHERE order_total < 0

-- tests/assert_no_future_dates.sql
-- Test fails if any created_at is in the future
SELECT *
FROM {{ ref('fct_orders') }}
WHERE created_at > current_timestamp

# Run tests:
$ dbt test

14:32:45  Running with dbt=1.7.0
14:32:48  Running test assert_order_totals_positive
14:32:49  ✓ PASS (0 failures)
14:32:50  Running test assert_no_future_dates
14:32:51  ✓ PASS (0 failures)
```

## Complete Test Configuration

```
version: 2
models:
  - name: fct_orders
    description: 'Fact table of orders'
    columns:
      - name: order_id
        description: 'Unique order ID'
        tests:
          - unique
          - not_null
      - name: customer_id
        description: 'Customer ID (FK to dim_customers)'
        tests:
          - not_null
          - relationships:
              to: ref('dim_customers')
              field: customer_id
      - name: order_total
        description: 'Order total in USD'
```

```
      tests:
        - not_null
    - name: order_status
      description: 'Order status'
      tests:
        - not_null
        - accepted_values:
            values: ['pending', 'shipped', 'delivered', 'cancelled']
    - name: created_at
      description: 'When order was created'
      tests:
        - not_null
```

## Running Tests

```
# Run all tests
$ dbt test

# Test one model
$ dbt test --select fct_orders

# Test with tags
$ dbt test --select tag:critical

# Run models and tests together (modern practice)
$ dbt build

# Build with specific selection
$ dbt build --select path:models/marts
```

## Test Configuration Options

| Option | Description | Example |
|---|---|---|
| severity | warn (don't fail) or error (fail) | severity: warn |
| where | Only test rows matching condition | where: "status != 'archived'" |
| limit | Only test first N rows | limit: 1000 |
| store_failures | Save failing rows to table | store_failures: true |
| tags | Tag test for selection | tags: ['daily', 'critical'] |

## Advanced: dbt-expectations Package

The dbt-expectations package provides 20+ data quality tests beyond the basics:

```
# packages.yml
packages:
  - package: calogica/dbt_expectations
    version: 0.8.0

$ dbt deps

# Use in .yml:
columns:
  - name: user_age
    tests:
      - dbt_expectations.expect_column_values_to_be_between:
          min_value: 0
          max_value: 120
      - dbt_expectations.expect_column_values_to_be_in_type_list:
          column_type: ['number', 'integer']
```

## Testing Strategy by Layer

| Layer | Models to Test | Key Tests |
|---|---|---|
| Sources | External tables | freshness check, row count |
| Staging | 1:1 with source | not_null on keys, accepted_values, unique |
| Intermediate | Business logic | relationships, aggregation counts |
| Marts | Final tables | unique, not_null, relationships, completeness |

# dbt-6: Project Structure & Naming Conventions

## The Standard dbt Project Layout

Successful dbt projects follow a consistent structure. This makes code maintainable, enables collaboration, and follows industry best practices (from dbt Labs and the dbt community).

```
my_analytics/
├── models/                      # All data transformations
│   ├── staging/                 # stg_* models (1:1 with sources)
│   │   ├── stg_customers.sql
│   │   ├── stg_orders.sql
│   │   ├── stg_products.sql
│   │   └── _stg_sources.yml     # Source definitions + docs
│   ├── intermediate/            # int_* models (business logic)
│   │   ├── int_customer_metrics.sql
│   │   ├── int_product_metrics.sql
│   │   └── _int_models.yml
│   └── marts/                   # fct_*, dim_* models (consumer-ready)
│       ├── fct_orders.sql
│       ├── dim_customers.sql
│       ├── dim_products.sql
│       └── _mart_models.yml
├── tests/                       # Singular (custom SQL) tests
│   ├── assert_order_totals_positive.sql
│   ├── assert_no_future_dates.sql
│   └── unit/                    # Unit tests (new in v1.8)
├── macros/                      # Reusable SQL functions (Jinja macros)
│   ├── generate_surrogate_key.sql
│   ├── audit_columns.sql
│   ├── limit_in_dev.sql
│   └── helpers.sql
├── seeds/                       # Static CSV lookup tables
│   ├── country_codes.csv
│   └── product_categories.csv
├── snapshots/                   # SCD Type 2 configurations
│   └── snap_customers.sql
├── analyses/                    # Ad-hoc queries (not models)
│   └── customer_analysis.sql
├── dbt_project.yml              # Project configuration
├── profiles.yml                 # Warehouse connection (in ~/.dbt/)
├── packages.yml                 # External package dependencies
├── README.md                    # Project documentation
└── .gitignore                   # Exclude secrets, target/
```

## Naming Conventions

Consistent naming makes projects self-documenting. Everyone knows a stg_ model is staging, a dim_ model is a dimension, etc.

| Prefix | Materialization | Purpose | Example |
|---|---|---|---|
| stg_ | view | Staging (1:1 with source, minimal processing) | stg_customers.sql |
| int_ | ephemeral/table | Intermediate (business logic, joins) | int_customer_metrics.sql |
| fct_ | table/incremental | Fact table (measurements, events) | fct_orders.sql |
| dim_ | table | Dimension table (attributes, lookups) | dim_customers.sql |
| snap_ | snapshot | Snapshot (SCD Type 2 tracking) | snap_customers.sql |
| test_ | various | Unit test | test_stg_customers.sql |
| tmp_ | ephemeral | Temporary intermediate | tmp_customer_dedup.sql |

## One Model Per File Rule

One .sql file = one model. This keeps projects maintainable, makes the DAG clear, and enables independent testing and documentation.

Do NOT do this:

```
-- models/staging/stg_all.sql (WRONG!)
CREATE OR REPLACE VIEW stg_customers AS ...;
CREATE OR REPLACE VIEW stg_orders AS ...;
CREATE OR REPLACE VIEW stg_products AS ...;
```

Do this instead:

```
-- models/staging/stg_customers.sql
SELECT ...

-- models/staging/stg_orders.sql
SELECT ...

-- models/staging/stg_products.sql
SELECT ...
```

## YAML File Organization

Two recommended approaches:

Approach 1: One .yml file per folder (recommended for large projects)

```
models/staging/
■■■ stg_customers.sql
■■■ stg_orders.sql
■■■ stg_products.sql
■■■ _stg_sources.yml   # All staging models + sources here

models/marts/
■■■ fct_orders.sql
■■■ dim_customers.sql
■■■ _mart_models.yml   # All mart models here
```

Approach 2: One .yml file per model (simpler for small projects)

```
models/
■■■ stg_customers.sql
■■■ stg_customers.yml
■■■ stg_orders.sql
■■■ stg_orders.yml
■■■ fct_orders.sql
■■■ fct_orders.yml
```

*Pick one approach and stick with it. Consistency beats perfection.*

## Folder Organization

- macros/ — Organize by purpose (e.g., macros/data_cleaning/, macros/testing/)
- tests/ — Organize by model or layer (e.g., tests/staging/, tests/marts/)
- seeds/ — Keep flat, group related CSVs (e.g., seed_countries.csv, seed_status_types.csv)
- snapshots/ — Keep flat, name clearly (snap_customers.sql, snap_products.sql)
- analyses/ — Ad-hoc queries that aren't part of the DAG
- docs/ — Custom markdown documentation files (if needed)

## Packages: Standing on Giants' Shoulders

dbt packages extend functionality. Define them in packages.yml and install with dbt deps.

```
# packages.yml
packages:
  - package: dbt-labs/dbt_utils
    version: 1.1.1
  - package: calogica/dbt_expectations
    version: 0.8.0
  - package: dbt-labs/dbt_date
    version: 1.0.4
  - package: dbt-labs/elementary
    version: 0.13.0

$ dbt deps

14:32:45  Installing dbt-labs/dbt_utils
14:32:48  Installing calogica/dbt_expectations
14:32:51  ✓ All packages installed
```

Popular packages:

| Package | Purpose | Key Macros/Features |
|---|---|---|
| dbt_utils | 50+ utility macros | generate_surrogate_key, slugify, hash_records, union_relations |
| dbt_expectations | 20+ quality tests | expect_column_values_to_be_between, expect_table_row_count_to_be_between |
| dbt_date | Date/time utilities | date_spine, iso_week_start, get_date_dimension |
| elementary | Data quality monitoring | dbt_anomaly detection, lineage visualization |
| codegen | Code generation | Auto-generate YAML from database schemas |
| dbt_artifacts | Build artifacts analysis | Track dbt performance, lineage, test results |

## Complete Project Tree with Annotations

```
my_analytics_project/
■■■ dbt_project.yml                  # Project config (name, version, defaults)
■■■ profiles.yml (~/.dbt/)           # Warehouse credentials (DO NOT COMMIT)
■■■ packages.yml                     # External dependencies
■■■ README.md                        # Project documentation
■■■ .gitignore                       # Exclude secrets and artifacts
■
■■■ models/                          # All transformation code
■   ■■■ staging/                     # 1:1 with sources, light cleaning
■   ■   ■■■ stg_customers.sql        # Customer staging
■   ■   ■■■ stg_orders.sql           # Order staging
■   ■   ■■■ stg_products.sql         # Product staging
■   ■   ■■■ stg_payment_events.sql   # Payment events staging
■   ■   ■■■ _stg_sources.yml         # Source definitions + column docs
■   ■■■ intermediate/                # Business logic, joins, aggregations
■   ■   ■■■ int_customer_metrics.sql # Customer agg (ephemeral)
■   ■   ■■■ int_product_sales.sql    # Product sales (ephemeral)
■   ■   ■■■ int_payment_summary.sql  # Payment summary (ephemeral)
■   ■   ■■■ _int_models.yml          # Intermediate model docs
■   ■■■ marts/                       # Final consumer-facing tables
■       ■■■ fct_orders.sql           # Order facts (incremental table)
■       ■■■ fct_events.sql           # Event facts (incremental)
■       ■■■ dim_customers.sql        # Customer dimension (table)
```

```
■       ■■■ dim_products.sql         # Product dimension (table)
■       ■■■ dim_dates.sql            # Date dimension (table)
■       ■■■ _mart_models.yml         # Mart model docs + tests
■
■■■ tests/                           # Custom SQL tests
■   ■■■ assert_order_totals_positive.sql
■   ■■■ assert_no_future_dates.sql
■   ■■■ assert_customer_id_completeness.sql
■   ■■■ unit/                        # Unit tests for logic (new in v1.8)
■       ■■■ test_stg_customers.sql
■
■■■ macros/                          # Reusable SQL functions
■   ■■■ generate_surrogate_key.sql   # Create hash from columns
■   ■■■ audit_columns.sql            # Add load timestamp, env
■   ■■■ limit_in_dev.sql             # Limit rows in dev only
■   ■■■ select_columns.sql           # Dynamic column selection
■   ■■■ union_tables.sql             # Union multiple staging tables
■   ■■■ helpers.sql                  # Other utility macros
■
■■■ seeds/                           # Static CSV lookup tables
■   ■■■ country_codes.csv            # Country lookup
■   ■■■ product_categories.csv       # Product categories
■   ■■■ status_mapping.csv           # Status lookups
■
■■■ snapshots/                       # SCD Type 2 tracking
■   ■■■ snap_customers.sql           # Track customer changes over time
■
■■■ analyses/                        # Ad-hoc analysis queries
■   ■■■ customer_segmentation.sql    # Not a dbt model
■   ■■■ monthly_revenue_trend.sql    # Not a dbt model
■
target/                             # Generated files (NOT committed)
■■■ compiled/                         # Compiled SQL
■■■ manifest.json                     # DAG (dependency graph)
■■■ run_results.json                  # Test/run results
dbt_packages/                       # Installed packages (NOT committed)
```

## Git Best Practices

Never commit to git:

```
# .gitignore
profiles.yml            # Warehouse credentials
target/                 # Compiled SQL and artifacts
dbt_packages/           # Installed packages
.env                    # Environment variables
*.pyc                   # Python cache
```

## Getting Started: First Project Setup

Complete workflow to create your first dbt project:

```
# 1. Install dbt
$ pip install dbt-snowflake  # or dbt-bigquery, dbt-postgres

# 2. Initialize project
$ dbt init my_analytics
$ cd my_analytics

# 3. Create first model (models/staging/stg_customers.sql)
SELECT customer_id, name, email FROM {{ source('raw', 'customers') }}
```