

Data Modeling

Designing data structures that are queryable, maintainable, and performant at scale.

Last reviewed 29 Sep 2026 ·

Prerequisites: [SQL](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) · [DE Concepts](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/)

Related: [dbt](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/) · [Snowflake](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/snowflake-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/snowflake-reference/) · [NoSQL and Operational Stores](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/nosql-operational-stores/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/nosql-operational-stores/) · [BI Tools](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/bi-tools/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/bi-tools/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Practice: [Lab 02 — dbt Transformations](https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/02-dbt-transformations) (https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/02-dbt-transformations)

Overview

Challenge: Source systems store data in the shape the *application* needs — many normalized tables, cryptic column names, status codes, and history overwritten on every update. Answering a simple business question requires complex joins, and different analysts arrive at different results.

Solution: Data modeling deliberately shapes data for *analysis*. The most common approach is the **star schema**: a central **fact** table of measured events (orders, clicks, payments) surrounded by **dimension** tables that describe them (customer, product, date).

Application tables (OLTP)
(star schema)

orders, order_items, customers,
addresses, products, categories,

Analytics model

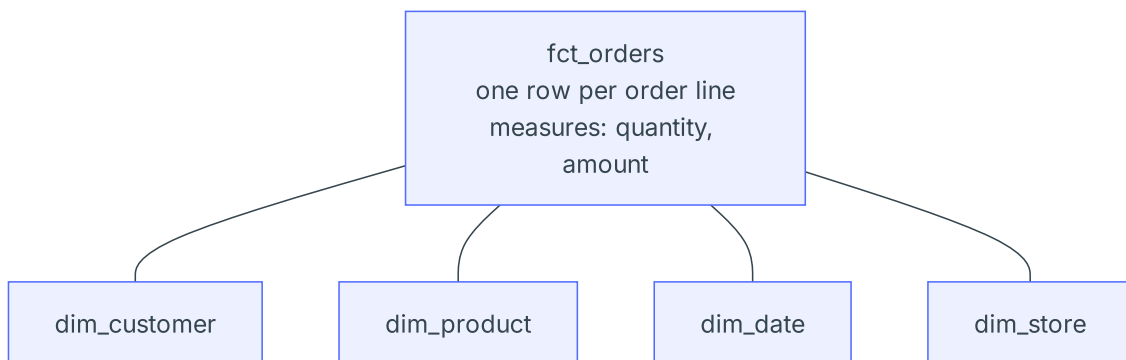
dim_date
|

```

promos, payments, refunds ...    →    dim_customer -
fct_orders - dim_product          |
(built for fast writes)           |    dim_promo
                                   |    (built for
                                   |    simple, fast reads)

```

Key design decisions: the **grain** (what a single row represents) and how **history** is handled when descriptive attributes change (slowly changing dimensions).



On this page

Basic - [Why Data Modeling Matters](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#why-data-modeling-matters) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#why-data-modeling-matters) - [Normalization](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#normalization) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#normalization) - [Entity-Relationship Design](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#entity-relationship-design) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#entity-relationship-design) - [Star Schema](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#star-schema) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#star-schema)

Intermediate - [Snowflake Schema](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#snowflake-schema) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#snowflake-schema) - [Fact Tables](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#fact-tables) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#fact-tables) - [Dimension Tables](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#dimension-tables) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#dimension-tables) - [Slowly Changing Dimensions \(SCDs\)](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#slowly-changing-dimensions-scds) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#slowly-changing-dimensions-scds)

Advanced - [One Big Table \(OBT\)](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#one-big-table-obt) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#one-big-table-obt) - [Data Vault](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#data-vault) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#data-vault) - [Layered Transformation Architecture](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#layered-transformation-architecture) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#layered-transformation-architecture) - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#common-pitfalls)

Reference - Cheat Sheet (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#cheat-sheet>) - **Interview Questions** (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#interview-questions>) - **Further Reading** (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/#further-reading>)

Why Data Modeling Matters

A bad data model doesn't fail loudly — it fails slowly: queries get harder to write, joins multiply, analysts distrust numbers, and fixing it means rewriting everything.

Bad model symptoms:

- "I need to join 12 tables to get revenue by region"
- "I don't know which orders table is the correct one"
- "The numbers in the report don't match the numbers in the query"
- "Every new question requires a new ETL job"

Good model symptoms:

- "SELECT SUM(revenue) FROM fct_orders GROUP BY region"
- "One orders fact table, one customer dimension, one date dimension"
- "The numbers always match because there's one source of truth"

The goal: make common analytical questions simple SQL, and rare questions possible SQL.

Normalization

Normalization removes redundancy by splitting data into related tables.

Normal forms

1NF (First Normal Form):

- Each column holds one value (no arrays, no comma-separated lists)
- Each row is unique (has a primary key)

Bad: orders(id, customer_name, customer_email, items="pen,paper,stapler")

Good: orders(id, customer_id), order_items(order_id, product_id)

2NF (Second Normal Form):

- 1NF + every non-key column depends on the WHOLE primary key
- Eliminates partial dependencies (applies to composite keys)

Bad: order_items(order_id, product_id, product_name)
product_name depends on product_id alone, not the composite key

Good: order_items(order_id, product_id, quantity)
products(product_id, product_name)

3NF (Third Normal Form):

- 2NF + no non-key column depends on another non-key column
- Eliminates transitive dependencies

Bad: orders(order_id, customer_id, customer_city, customer_country)
customer_country depends on customer_city, not order_id

Good: orders(order_id, customer_id)
customers(customer_id, city_id)
cities(city_id, city_name, country)

Normalization vs denormalization

	Normalized (3NF)	Denormalized
Storage	Less (no duplication)	More
Update anomalies	None	Risk — must update in multiple places
Query complexity	Higher (many joins)	Lower (few joins)
Read performance	Slower (joins)	Faster
Use case	OLTP (transactional)	<u>OLAP</u> (analytical)

Rule of thumb: Normalize for writes (OLTP), denormalize for reads (analytics).

Entity-Relationship Design

Before writing DDL, sketch what entities exist and how they relate.

```
Entities:      Customer, Order, Product, Category, Promotion
Relationships:
  Customer  —< Order      (one customer, many orders)
  Order     —< OrderItem  (one order, many line items)
  OrderItem >— Product    (many line items reference one product)
  Product   >— Category   (many products belong to one category)
  Order     >— Promotion  (many orders may use one promotion – nullable)
```

Cardinality notation:

- < one-to-many
- >—< many-to-many (needs a junction table)
- zero or one (optional / nullable)

```
-- Entity-relationship translated to SQL
CREATE TABLE customers (
  id          INT PRIMARY KEY,
  name        VARCHAR(200) NOT NULL,
  email       VARCHAR(200) UNIQUE NOT NULL,
  region      VARCHAR(50),
  created_at  TIMESTAMP DEFAULT NOW()
);

CREATE TABLE products (
  id          INT PRIMARY KEY,
  name        VARCHAR(200) NOT NULL,
  category_id INT REFERENCES categories(id),
  price       DECIMAL(10,2) NOT NULL CHECK (price >= 0),
  is_active   BOOLEAN DEFAULT TRUE
);

CREATE TABLE orders (
  id          INT PRIMARY KEY,
  customer_id INT NOT NULL REFERENCES customers(id),
  promotion_id INT REFERENCES promotions(id), --
```

```

nullable (optional)
    status          VARCHAR(20) CHECK (status IN
('placed','shipped','delivered','cancelled')),
    created_at      TIMESTAMP NOT NULL DEFAULT NOW(),
    total_amount    DECIMAL(12,2) NOT NULL
);

CREATE TABLE order_items (
    order_id        INT NOT NULL REFERENCES orders(id),
    product_id      INT NOT NULL REFERENCES products(id),
    quantity        INT NOT NULL CHECK (quantity > 0),
    unit_price      DECIMAL(10,2) NOT NULL,
    PRIMARY KEY (order_id, product_id)
);

```

Star Schema

The most common pattern for analytical data warehouses. One central **fact table** surrounded by **dimension tables**.



```

-- Dimension: customers
CREATE TABLE dim_customer (
    customer_key INT PRIMARY KEY,          -- surrogate
    key (warehouse-generated)
    customer_id  VARCHAR(50) NOT NULL,     -- natural key
    (from source system)
    name         VARCHAR(200),
    email        VARCHAR(200),
    region       VARCHAR(50),
    segment      VARCHAR(50),              -- derived:
    e.g. VIP, Standard, At-risk
    valid_from   DATE NOT NULL,
    valid_to     DATE,                     -- NULL =
    currently active record

```

```

        is_current      BOOLEAN DEFAULT TRUE
    );

-- Dimension: dates (pre-populated for the full date range)
CREATE TABLE dim_date (
    date_key            INT PRIMARY KEY,      -- YYYYMMDD integer
    (20240315)
    date                DATE NOT NULL,
    year                INT,
    quarter              INT,
    month                INT,
    month_name           VARCHAR(10),
    week_of_year         INT,
    day_of_week          INT,
    day_name             VARCHAR(10),
    is_weekend           BOOLEAN,
    is_holiday           BOOLEAN,
    fiscal_quarter       INT,
    fiscal_year          INT
);

-- Fact: orders (one row per order)
CREATE TABLE fct_orders (
    order_key           INT PRIMARY KEY,      -- surrogate key
    order_id            VARCHAR(50),         -- natural key
    customer_key        INT REFERENCES
dim_customer(customer_key),
    product_key         INT REFERENCES
dim_product(product_key),
    date_key            INT REFERENCES dim_date(date_key),
    promotion_key       INT REFERENCES
dim_promotion(promotion_key),

    -- Measures (the numbers analysts aggregate)
    order_amount        DECIMAL(12,2),
    quantity            INT,
    discount_amount     DECIMAL(12,2),
    net_amount          DECIMAL(12,2),

    -- Degenerate dimensions (attributes with no dimension
table)
    order_status        VARCHAR(20),

```

```
shipping_method VARCHAR(50)
);
```

Why surrogate keys? - Natural keys from source systems change (customer emails change, order IDs get reused) - Surrogate keys are stable warehouse-internal identifiers - Required to support SCD Type 2 (multiple rows per entity over time)

Star schema queries are simple:

```
-- Revenue by region and quarter
SELECT
    c.region,
    d.fiscal_quarter,
    d.fiscal_year,
    SUM(f.net_amount) AS revenue,
    COUNT(DISTINCT f.order_key) AS order_count
FROM fct_orders f
JOIN dim_customer c ON f.customer_key = c.customer_key
JOIN dim_date d ON f.date_key = d.date_key
WHERE c.is_current = TRUE
GROUP BY 1, 2, 3
ORDER BY 3, 2, 1;
```

Snowflake Schema

A normalized star schema — dimension tables reference other dimension tables.

```
dim_geography (city, country)
|
dim_customer — fct_orders — dim_product —
dim_category
```

```
-- Snowflake: product references category (not embedded)
CREATE TABLE dim_category (
    category_key INT PRIMARY KEY,
    category_id VARCHAR(50),
    name VARCHAR(100),
    department VARCHAR(100)
);
```

```
CREATE TABLE dim_product (
  product_key INT PRIMARY KEY,
  product_id VARCHAR(50),
  name VARCHAR(200),
  category_key INT REFERENCES dim_category(category_key),
  -- normalized
  price DECIMAL(10,2)
);
```

Star vs Snowflake:

	Star	Snowflake
Joins per query	Fewer	More
Storage	More (duplication)	Less
Query performance	Faster	Slower
Maintenance	Simpler	More complex
Use when	Default choice	Very large dimensions with deep hierarchies

Recommendation: Use star schema by default. Only snowflake when dimension tables are large enough that duplication is a real cost.

Fact Tables

Fact tables store events and measurements. Every row is one business event.

Types of fact tables

1. Transaction facts (most common)
 - One row per event: each order, each click, each payment
 - Examples: fct_orders, fct_page_views, fct_payments
2. Periodic snapshot facts
 - One row per entity per period: daily balance, weekly inventory
 - Examples: fct_account_balance_daily, fct_inventory_weekly

3. Accumulating snapshot facts

One row per business process lifecycle: tracks milestones

Example: fct_order_fulfillment (placed_at, shipped_at, delivered_at all on one row)

```
-- Transaction fact: one row per order line item
CREATE TABLE fct_order_items (
  surrogate_key    BIGINT PRIMARY KEY,
  order_id         VARCHAR(50),
  order_item_id    VARCHAR(50),
  customer_key     INT,
  product_key      INT,
  date_key         INT,

  -- Additive measures (can SUM across any dimension)
  quantity         INT,
  unit_price       DECIMAL(10,2),
  gross_amount     DECIMAL(12,2),
  discount_amount  DECIMAL(12,2),
  net_amount       DECIMAL(12,2),

  -- Semi-additive (SUM only along some dimensions)
  -- e.g., inventory levels can be summed across products
  -- but not time

  -- Non-additive (never SUM – use AVG, COUNT, ratios)
  unit_margin_pct  DECIMAL(5,2)
);

-- Accumulating snapshot: one row per order, updated as it
-- progresses
CREATE TABLE fct_order_fulfillment (
  order_key        INT PRIMARY KEY,
  order_id         VARCHAR(50),
  customer_key     INT,

  -- Milestone date keys (NULL until milestone reached)
  placed_date_key  INT,
  confirmed_date_key INT,
  shipped_date_key  INT,
```

```

delivered_date_key INT,
returned_date_key  INT,

-- Lag measures (computed from milestones)
days_to_ship      INT,      -- shipped - placed
days_to_deliver   INT,      -- delivered - shipped
is_on_time         BOOLEAN
);

```

Dimension Tables

Dimensions provide context for facts — who, what, where, when.

```

-- Wide dimension: many attributes, denormalized
CREATE TABLE dim_customer (
  customer_key    INT PRIMARY KEY,
  customer_id     VARCHAR(50) NOT NULL,

  -- Identity
  full_name       VARCHAR(200),
  email           VARCHAR(200),

  -- Geography (denormalized from dim_geography)
  city            VARCHAR(100),
  state           VARCHAR(100),
  country         VARCHAR(100),
  region          VARCHAR(50),

  -- Segmentation
  segment         VARCHAR(50),      -- VIP / Standard / At-
risk
  tier             VARCHAR(20),      -- Gold / Silver /
Bronze
  acquisition_channel VARCHAR(100),

  -- SCD Type 2 fields
  valid_from      DATE NOT NULL,
  valid_to        DATE,
  is_current      BOOLEAN DEFAULT TRUE,
  source_system   VARCHAR(50),

```

```
    dbt_updated_at    TIMESTAMP
);
```

Junk dimensions — combine low-cardinality flags/codes into one table to avoid fact table bloat:

```
-- Instead of 5 flag columns on the fact table:
CREATE TABLE dim_order_flags (
    flag_key          INT PRIMARY KEY,
    is_first_order    BOOLEAN,
    is_gift            BOOLEAN,
    has_promotion      BOOLEAN,
    is_subscription    BOOLEAN,
    is_international   BOOLEAN
);
-- fct_orders.flag_key → dim_order_flags
```

Slowly Changing Dimensions (SCDs)

What happens when a customer moves cities or changes their name? SCD types define the strategy.

Type 0 — Ignore changes

```
-- Never update. Original value is kept forever.
-- Use when: the attribute should never change (birthdate,
signup date)
```

Type 1 — Overwrite

```
-- Overwrite the old value. No history.
-- Use when: corrections (typo fix), or history doesn't
matter
UPDATE dim_customer
SET email = 'new@email.com'
WHERE customer_id = 'C001';
```

Type 2 — Add a new row (most common)

```
-- Close the old row, insert a new row. Full history preserved.
-- Requires: surrogate key, valid_from, valid_to, is_current

-- Close old row
UPDATE dim_customer
SET valid_to = CURRENT_DATE - 1,
    is_current = FALSE
WHERE customer_id = 'C001' AND is_current = TRUE;

-- Insert new row
INSERT INTO dim_customer (customer_key, customer_id,
    region, valid_from, valid_to, is_current)
VALUES (nextval('customer_seq'), 'C001', 'EMEA',
    CURRENT_DATE, NULL, TRUE);
```

```
-- Query: current state
SELECT * FROM dim_customer WHERE is_current = TRUE;

-- Query: what was the customer's region when they placed
order #1234?
SELECT c.region
FROM fct_orders f
JOIN dim_customer c ON f.customer_key = c.customer_key
WHERE f.order_id = '1234';
-- Works because fct_orders stores the customer_key at the
time of the order
```

Type 3 — Add a column

```
-- Store previous value in a separate column. Only one
level of history.
-- Use when: you need "previous" but not full history

ALTER TABLE dim_customer
ADD COLUMN previous_region VARCHAR(50),
ADD COLUMN region_changed_at DATE;
```

```

UPDATE dim_customer
SET previous_region = region,
    region          = 'EMEA',
    region_changed_at = CURRENT_DATE
WHERE customer_id = 'C001';

```

SCD comparison

Type	History	Storage	Complexity	Use when
0	None	Lowest	Trivial	Immutable attributes
1	None	Low	Low	Corrections, history irrelevant
2	Full	High	Medium	Regulatory, auditing, time-travel analysis
3	Previous only	Medium	Low	"Before/after" comparison
6 (hybrid)	Full + current column	Highest	High	Need both full history and easy current-state access

Tool example: dbt snapshots (SCD Type 2)

```

-- snapshots/snap_customers.sql
{% snapshot snap_customers %}

{{
    config(
        target_schema = 'snapshots',
        unique_key     = 'customer_id',
        strategy       = 'timestamp',      -- or 'check'
        updated_at     = 'updated_at',
    )
}}

SELECT * FROM {{ source('raw', 'customers') }}

```

```
{% endsnapshot %}
```

One Big Table (OBT)

Denormalize everything into a single wide table. Controversial but sometimes right.

```
-- OBT: one row per order with all attributes pre-joined
CREATE TABLE orders_obt AS
SELECT
    o.order_id,
    o.created_at,
    o.amount,
    o.status,
    c.name           AS customer_name,
    c.region         AS customer_region,
    c.segment        AS customer_segment,
    p.name           AS product_name,
    p.category       AS product_category,
    pr.code          AS promo_code,
    pr.discount_pct  AS promo_discount
FROM orders o
JOIN customers c ON o.customer_id = c.id
JOIN products  p ON o.product_id  = p.id
LEFT JOIN promos pr ON o.promo_id   = pr.id;
```

OBT pros and cons:

Pros	Cons
Zero joins — fastest queries	Massive duplication
Simple for BI tools	Hard to update (no normalization)
Works well with columnar storage	Column count explosion (100s of cols)
Great for ML feature tables	Historical changes are hard (no <u>SCD</u>)

Use OBT when: final gold layer tables for BI/dashboards, feature stores for ML, or when the audience is analysts who don't write SQL.

Data Vault

A modeling approach for enterprise data warehouses emphasizing auditability, parallel loading, and schema flexibility. Overkill for most teams — documented here for awareness.

Three entity types:

Hub: Stores business keys (one row per unique entity)
Link: Stores relationships between hubs (like a fact table without measures)
Satellite: Stores descriptive attributes and history (SCD Type 2 equivalent)

Example:

```
HUB_CUSTOMER(customer_hk, customer_id, load_ts,
record_source)
HUB_ORDER (order_hk, order_id, load_ts, record_source)
LNK_CUSTOMER_ORDER(link_hk, customer_hk, order_hk,
load_ts, record_source)
SAT_CUSTOMER(customer_hk, load_ts, load_end_ts, name,
email, region, record_source)
```

When to consider Data Vault: - Multiple source systems feeding the same entities - Strict audit requirements (financial services, healthcare) - Schema changes happen frequently - Team size > 10 engineers on the warehouse

Layered Transformation Architecture

Most teams organize warehouse transformations in layers, regardless of the tool that runs them (plain SQL scripts, stored procedures, dbt, SQLMesh, Dataform, or Spark SQL):

Sources	Raw tables loaded from source systems
↓	
Staging	stg_<source>__<entity> One-to-one with a source table, light cleaning only:
↓	rename columns, cast types, add metadata
Intermediate	int_<entity>__<transformation> Business logic, joins, derivations Not exposed to end users
↓	

Marts

fct_<entity> or dim_<entity>

Final, modeled tables for BI and analysis

```
-- staging.stg_billing__orders - rename + cast only
CREATE OR REPLACE VIEW staging.stg_billing__orders AS
SELECT
    id                AS order_id,
    customer          AS customer_id,
    amount / 100.0    AS amount,      -- source
stores minor units (cents)
    status,
    CAST(created AS TIMESTAMP) AS created_at
FROM raw.billing_orders;

-- intermediate.int_orders__enriched - join + derive
CREATE OR REPLACE VIEW intermediate.int_orders__enriched AS
SELECT
    o.order_id,
    o.customer_id,
    o.amount,
    o.status,
    o.created_at,
    c.region,
    c.segment,
    ROW_NUMBER() OVER (PARTITION BY o.customer_id ORDER BY
o.created_at) AS customer_order_num
FROM staging.stg_billing__orders o
JOIN staging.stg_crm__customers c ON o.customer_id =
c.customer_id;

-- marts.fct_orders - final, documented table
CREATE OR REPLACE TABLE marts.fct_orders AS
SELECT
    order_id,
    customer_id,
    amount,
    status,
    created_at,
    region,
    segment,
    customer_order_num,
```

```
customer_order_num = 1 AS is_first_order
FROM intermediate.int_orders__enriched;
```

Rules that keep layers maintainable: staging models never join; business logic lives in intermediate models; marts are the only layer BI tools read; each model has one grain and a tested primary key.

Tooling: transformation frameworks automate the dependency order between these layers. In dbt, for example, `FROM staging.stg_billing__orders` becomes `FROM {{ ref('stg_billing__orders') }}`, and raw tables are referenced with `{{ source('billing', 'orders') }}` — see the [dbt guide](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/) (<https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/>).

Common Pitfalls

1. Using natural keys as fact table join keys 
Problem: source system natural keys change; history breaks
Fix: Always use surrogate keys in dimension tables
2. Putting measures in dimension tables
Problem: "total_orders" on dim_customer goes stale immediately
Fix: Measures belong in fact tables; derive them at query time
3. One massive fact table with 200 columns
Problem: hard to maintain, columns have inconsistent grain
Fix: Separate facts by grain (order-level vs item-level)
4. Not defining grain before building
Problem: analysts aggregate incorrectly, double-counting
Fix: Every fact table's documentation must state its grain
e.g., "fct_orders: one row per order"
e.g., "fct_order_items: one row per order line item"
5. Ignoring NULL foreign keys

Problem: INNER JOINS silently drop those fact rows (and LEFT JOINS show

NULL attributes) – metrics disagree depending on how you join

Fix: Use a "Unknown" or "Not Applicable" dimension row (key = -1)

so NULLs never appear in fact table FK columns

6. SCD Type 1 when Type 2 was needed

Problem: customer moved regions; historical orders now show wrong region

Fix: Use SCD Type 2 for any dimension attribute that affects

historical analysis

7. Wide dim tables with 100+ columns from many sources

Problem: slow to maintain, unclear ownership

Fix: Split into role-playing dimensions or separate dims per source

8. Using date strings instead of integer date keys

Problem: slow joins, no pre-computed date attributes

Fix: Use INTEGER date keys (YYYYMMDD) and a pre-populated dim_date

Cheat Sheet

Decision	Rule of thumb
First step for any fact table	Write down the grain: "one row per ____"
Measures (amounts, counts, durations)	Fact tables
Descriptive attributes (names, categories, regions)	Dimension tables
Join keys in facts	Surrogate keys, not source-system natural keys
Missing dimension value	Point to an "Unknown" row (key -1), never leave the FK NULL

Decision	Rule of thumb
Attribute change, history irrelevant or a typo fix	SCD Type 1 (overwrite)
Attribute change that affects historical reporting	SCD Type 2 (new row + <code>valid_from</code> / <code>valid_to</code> / <code>is_current</code>)
Only need "previous value"	SCD Type 3 (extra column)
Attribute that must never change	SCD Type 0
Dashboards / BI audience	Star schema, or One Big Table in Gold
Many sources, heavy audit requirements	Data Vault in the raw/integration layer, star schema on top
Transformation layering	<code>stg_</code> (1:1 with source) → <code>int_</code> (logic) → <code>fct_</code> / <code>dim_</code> (marts)

Fact table types: transaction (one row per event) · periodic snapshot (one row per entity per period) · accumulating snapshot (one row per process, updated at milestones) · factless (events with no measures, e.g. attendance)

Dimension patterns: conformed (shared across facts) · role-playing (`dim_date` as order date *and* ship date) · junk (bundle of low-cardinality flags) · degenerate (an ID kept in the fact with no dimension, e.g. `order_number`) · outrigger (dimension that references another dimension)

Point-in-time join for SCD Type 2:

```
JOIN dim_customer c
  ON f.customer_id = c.customer_id
 AND f.order_ts >= c.valid_from
 AND f.order_ts < COALESCE(c.valid_to, '9999-12-31')
```

Interview Questions

Q: What is the grain of a fact table and why is it the first thing to decide? A: The grain is what one row represents — "one row per order line item" or "one row per customer per day". Every other design decision follows from it: which dimensions apply, which measures

are additive, and how the table can be aggregated safely. Mixing grains in one table (order-level shipping cost next to item-level price) leads to double-counting when someone sums a column.

Q: What is the difference between a star schema and a snowflake schema? A: Both have fact tables at the center. In a star schema, dimensions are denormalized — `dim_product` includes category and department names directly. In a snowflake schema, dimensions are normalized into sub-dimensions (`dim_product` → `dim_category` → `dim_department`). Star schemas mean fewer joins and simpler SQL, which is why they're preferred in modern columnar warehouses where storage is cheap; snowflake schemas save a little storage and reduce update anomalies.

Q: Explain SCD Type 1, 2, and 3. When would you use Type 2? A: Type 1 overwrites the value and keeps no history. Type 2 closes the current row (sets `valid_to`, `is_current` = `false`) and inserts a new row with a new surrogate key, preserving full history. Type 3 keeps the previous value in an extra column, giving one level of history. Use Type 2 whenever historical reports must reflect the attribute *as it was* — a customer's region or a salesperson's territory at the time of a sale — so last year's numbers don't change when someone moves.

Q: Why use surrogate keys instead of natural keys? A: Natural keys come from source systems and can change, be reused, collide across sources, or be missing. Surrogate keys (integer sequences or hashes generated in the warehouse) are stable and unique, make SCD Type 2 possible (one natural key, many versions, each with its own surrogate key), and let you handle unknown members with a reserved key such as `-1` .

Q: What are additive, semi-additive, and non-additive measures? A: Additive measures can be summed across every dimension (revenue, quantity). Semi-additive measures can be summed across some dimensions but not time — an account balance can be summed across accounts on one day, but not across days (take the last value or an average instead). Non-additive measures can't be summed at all — ratios and percentages; store the numerator and denominator and compute the ratio after aggregating.

Q: Star schema, One Big Table, or Data Vault — how do you choose? A: A star schema is the default for analytics: flexible, understandable, and efficient. One Big Table (everything pre-joined) is great for a specific dashboard, ML features, or non-SQL users, but it duplicates data and makes history and reuse harder — it's usually built *from* a star schema in the Gold layer. Data Vault suits large enterprises with many source systems and strict audit needs, as an integration layer that absorbs change, with star schemas built on top for consumption.

Q: How would you model orders and their line items? A: Two fact tables at different grains: `fct_orders` (one row per order — order total, shipping, discount) and `fct_order_items` (one row per line — product, quantity, price). Both share conformed dimensions (customer, date) and the item fact also joins to `dim_product` . Order-level amounts are not repeated on item rows; if they must be, allocate them proportionally so they sum correctly.

Q: How do you handle a fact that arrives before its dimension record (a late-arriving dimension)? A: Don't drop the fact. Either point it at an "Unknown" member (key -1) and re-key it once the dimension arrives, or insert an "inferred" placeholder dimension row with just the natural key and update its attributes when the real record lands. Which to choose depends on how often it happens and whether reprocessing facts is cheap.

Further Reading

- *The Data Warehouse Toolkit, 3rd Edition* — Ralph Kimball & Margy Ross (Wiley). The reference for dimensional modeling.
 - [Kimball Group dimensional modeling techniques](https://www.kimballgroup.com/data-warehouse-business-intelligence-resources/kimball-techniques/dimensional-modeling-techniques/) (https://www.kimballgroup.com/data-warehouse-business-intelligence-resources/kimball-techniques/dimensional-modeling-techniques/) — free one-page summaries of every pattern above
 - [dbt: How we structure our dbt projects](https://docs.getdbt.com/best-practices/how-we-structure/1-guide-overview) (https://docs.getdbt.com/best-practices/how-we-structure/1-guide-overview)
 - *Building a Scalable Data Warehouse with Data Vault 2.0* — Dan Linstedt & Michael Olschimke (Morgan Kaufmann)
 - *Agile Data Warehouse Design* — Lawrence Corr. Practical techniques for gathering modeling requirements with stakeholders.
-

Previous: [Python for DE](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundation-s/python-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundation-s/python-reference/) • **Next:** [Linux & Bash](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/linux-bash/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/linux-bash/) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 29, 2026