Step 1: Read-Heavy vs. Write-Heavy Systems

System Type	Primary Operation	Schema Style	Justification
Transactional (OLTP)	**Write-Heavy**	**Normalized (3NF)**	High-frequency inserts/updates. Normalization prevents anomalies and ensures that a single record (like a bank balance) is only stored in one place.
Reporting (BI/Dashboards)	**Read-Heavy**	**Denormalized (Star Schema)**	Users need quick answers to complex questions. Reducing joins by "flattening" data into wide tables speeds up dashboard load times.
Data Warehouse (OLAP)	**Hybrid/Read-Centric**	**Vault or Star Schema**	Stores massive historical data. It uses a mix: normalized "Hubs" for integrity and denormalized "Dimensions" for easy end-user querying.

Step 2: Performance vs. Duplication

Why Joins Hurt Performance

When you join 5 tables, the database engine must find matching keys across 5 different storage locations.

- **The Cost:** Memory usage spikes, and "CPU wait time" increases as the engine sorts and merges datasets.
- **Example:** In a 3NF schema, getting a simple "Order Total per City" might require joining `Orders` → `OrderItems` → `Products` → `Customers` → `Cities`.

Why Duplication Risks Inconsistency

If you denormalize and store `customer_city` directly in the `Orders` table to avoid a join:

- **The Risk:** If a customer moves from New York to Dubai, you must update every historical order row to keep the city accurate, or accept that old orders will show "New York" while new ones show "Dubai."
- **Example:** A "Update Anomaly" occurs when one row is updated but its duplicates are missed, leading to "Split Brain" data.

Step 3: Case Study Evaluation

Case 1: Core Banking Transactions

- **Recommendation: Strictly Normalized (3NF)**
- **Justification:** Data integrity is the #1 priority. We cannot risk a "double-spend" or a balance error due to duplicate data. Transactional speed is managed by powerful hardware and indexing, not by flattening tables.

Case 2: Product Catalog API

- **Recommendation: Denormalized (Document/NoSQL style)**
- **Justification:** Read-heavy. When a customer clicks a product, they need the name, price, specs, and reviews instantly. Fetching a single pre-built "Product JSON" is faster than joining 10 relational tables for every page load.

Case 3: Daily Sales Reporting

- **Recommendation: Denormalized (Star Schema/Flat Table)**
- **Justification:** Performance-centric. Managers running "Year-over-Year" reports shouldn't wait for trillions of joins. We trade off storage space (duplication) for the ability to scan a single "Wide Table" quickly.

Step 4: Design Reflection

Where Normalization is Mandatory

- **Source Systems:** Any system where data is entered by a human or a machine (ERPs, CRMs).
- **Consequence of failure:** If the source is messy, the entire downstream data pipeline is "garbage in, garbage out."

Where Denormalization is Acceptable

- **The "Gold" Layer:** In dbt, we often keep our `staging` models normalized but denormalize our final `marts` for the BI tools.
- **Safeguards:** Use **dbt tests** to ensure that even if data is duplicated, it remains consistent (e.g., testing that `unit_price` in an analytics table matches the source).

Decision Principles

1. **Identify the "Pain":** Is the query slow (need denormalization) or is the data wrong (need normalization)?
2. **Follow the "Rule of One":** Store truth in one place (Normalized Staging), but serve it in many shapes (Denormalized Marts).