# DATA MODELLING FOR DATA ENGINEERS

**A Complete Interview & Warehousing Playbook**

---

**By Manjinder Brar**

**Premium Edition — 2025**

---

**Version 1.0**
Last Updated: *Dec 2025*

**Contact**
Email
Linkedin

# Introduction: What Data Modelling Actually Means in Interviews

Data modelling in interviews is not about drawing pretty diagrams or memorizing star vs. snowflake definitions. At senior and FAANG-level data engineering interviews, modelling is a **thinking skill**, not a textbook skill. The interviewer wants to see how you understand the business, how you handle ambiguity, how you design for scale, and how you make decisions under constraints.

At its core, data modelling is the ability to convert *messy*, *incomplete* business requirements into a *clean*, *scalable*, *analytics-ready* data representation. It's the bridge between product, engineering, data science, and analytics.

In high-level interviews, you're expected to demonstrate not just *what* model you build, but *why* you chose it, what tradeoffs you made, what assumptions you clarified, and how your model holds up under real-world pressures like late data, schema evolution, GDPR constraints, multi-source truth, and rapid growth.

FAANG companies evaluate your modelling maturity through four dimensions:

- *Business Clarity* - how well you uncover the core business questions your model must answer.
- *Logical Precision* - how you identify entities, relationships, grains, and change-tracking strategies.
- *Architectural Alignment* - whether your model integrates cleanly into a warehouse, lakehouse, streaming, or ML feature ecosystem.
- *Tradeoff Thinking* - how you justify modeling decisions under constraints like cost, latency, historical storage, and data quality.

In real interviews, a strong candidate doesn't jump to fact/dimension tables immediately. They slow down, ask the right scoping questions, define entities with clean boundaries, choose the correct grain, map relationships clearly, and show how the model supports analytics, experimentation, and machine learning, all while keeping it maintainable for years.

This guide is designed to train that exact skill. Not generic diagrams, not surface-level SQL. You'll learn how to think like a senior Data Engineer who understands data deeply, asks the right questions instinctively, and models systems with long-term clarity and scalability. The same thinking behind how top tech companies design the data+sets powering recommendations, fraud detection, sales intelligence, payments, and growth analytics.

By the time you finish, you'll understand not only how to build strong data models but how to *present*, *defend*, and *evolve* them the way FAANG interviewers expect.

## FAANG interviews evaluate reasoning, not diagrams

FAANG companies don't care if your final diagram is pretty, they care about how you *think*. A strong data modelling answer isn't judged by how many entities or arrows you draw, but by the clarity, depth, and logic behind every modelling decision. Here's exactly what they evaluate:

1. *Your ability to define the core business question first :* Interviewers want to see whether you start with *why the model exists*. Do you understand the business metric, the analytical workflow, and the downstream consumer before shaping tables?

2. *How you choose grain and defend it:* Grain is the single most important DE modelling decision. Interviewers look for clarity on *why* you picked that level and how it impacts joins, storage, accuracy, and query performance.

3. *Whether you can translate messy inputs into clean relational structures:* FAANG problems always include ambiguity: optional fields, evolving schemas, or undefined relationships. They evaluate how you impose structure without overfitting.

4. *Your ability to surface and resolve trade-offs* : They expect you to articulate concrete trade-offs:

    - Wide vs. narrow tables
    - SCD Type 1 vs. Type 2
    - Data Vault vs. dimensional
    - Normalization vs. denormalization
    - Your reasoning must be explicit, not implied

5. *How you enforce data quality & reproducibility*: Interviewers check if you bake constraints, contracts, and lineage into your model from day one not as an afterthought.

6. *Your awareness of scale, cost, and evolution:* A good answer shows you understand how your model behaves with:

    - 10x growth
    - schema drift
    - backfills
    - late-arriving data
    - new analytical use cases

7. *The clarity of your communication under pressure:* Ultimately, they're testing whether you can communicate modelling logic the way a senior engineer would in a design review i.e structured, justified, and focused on impact.

## Modelling = Business Clarity + Data Clarity + Scalable Architecture

Strong data modelling in interviews isn't about drawing pretty schemas. It's about showing that you deeply understand *why* the business needs something, *what* data actually represents in real life, and *how* to structure that data so the company can scale without chaos.

Interviewers want to see that you can connect messy business workflows to clean, reliable data structures. If you can explain the relationships, constraints, and growth patterns behind your design, not just the tables you're already ahead of 90% of candidates.

## Why "drawing a star schema" is not enough anymore

In today's interviews, companies already know that most candidates have memorized a handful of star schemas, snowflakes, slowly changing dimensions, and "best practices." Hiring teams are no longer impressed by someone sketching a generic *sales_fact* with *dim_customer* and *dim_product*. They want to see whether you can *think*, not whether you can *recall*.

Modern data modelling rounds test whether your design is shaped by the actual business problem, the data realities, and the scalability constraints of that specific scenario.

FAANG teams are trained to detect "crammed answers", diagrams that look polished but are disconnected from the use case, ignore edge cases, and fail under real data volume or schema drift.

- What they truly want is a candidate who can reason from first principles:
- What problem are we solving?
- What questions must this data answer?
- What granularity is needed?
- What trade-offs exist?
- When you anchor your model to the business context and justify every modelling choice, you separate yourself instantly from those who rely on memorized shapes.

# Fundamentals You Must Know Cold

Before you even touch a schema, every interviewer expects you to have a rock-solid grip on the fundamentals. Not the textbook definitions but the *why* behind each concept, the trade-offs they introduce, and how they shape real production systems.

At FAANG-level interviews, fundamentals are not warm-up questions. They are the lens through which every case study is evaluated. If your understanding is shallow, your entire design falls apart the moment they apply pressure. But when you know these basics cold:

- cardinality
- grain
- SCD patterns
- data quality constraints
- normalization vs denormalization
- late-arriving logic
- incremental modelling, and more

you can defend every decision with clarity and confidence.

This section makes sure you're not guessing, not memorizing, but *thinking like a senior data engineer,* someone who can build models that scale, evolve, and support analytics, ML, and product use-cases without breaking under real-world complexity.

## OLTP vs OLAP

In interviews, companies expect you to understand that data models depend on the workload. OLTP and OLAP aren't just system types: they force different design choices.

### ***OLTP (Online Transaction Processing)***

These systems run the business: payments, sign-ups, ride requests, orders.

They demand:

- Row-oriented design for fast inserts/updates
- Normalized schemas to avoid anomalies
- Strict ACID transactions
- High concurrency + low latency (milliseconds)

Your model here must protect *integrity* and run at *scale*.

***OLAP (Online Analytical Processing)***

These systems analyze the business: dashboards, trends, forecasting, ML features.

They demand:

- Columnar storage for large scans
- Denormalized models (star/snowflake) for fast aggregation
- Batch or streaming ingestion, not per-row transactions
- Query performance + read scalability

Your model here must deliver *speed* and *clarity* for analytics.

***The interview expectation:***

You must show you know how modeling decisions change when the system shifts from high-write transactional workloads (OLTP) to high-read analytical workloads (OLAP).

If you treat them the same, it signals you don't understand real-world architecture.

## 3NF vs Star vs Snowflake

In interviews, you're not being tested on whether you *know the definitions*, they want to see if you understand *why each modelling style exists*, what problem it solves, and how your choice changes downstream *latency, scalability, data quality,* and *analytics speed.*

### *3NF (Third Normal Form) - OLTP Optimization*

3NF models are built for operational systems where priorities are: *fast writes, minimal redundancy, strict consistency,* and *small transactions.*

- Highly normalized → many tables, many joins.
- Excellent for source databases (e.g., CRM, billing, app DB).
- Prevents anomalies during inserts/updates.
- Not optimized for analytics as too many joins, slow for large scans.
- In DE interviews, you mention 3NF as *the format of operational data before ingestion.*

*When interviewers expect you to pick 3NF:*

When the workload is *transaction-heavy* and the goal is to maintain *data integrity at scale.*

### *Star Schema - Analytics Optimization & Simplicity*

Star schema exists because analysts don't want to join 25 tables and BI tools don't want to scan 100M-row fact tables with unnecessary dimensions.

- Facts = numeric, grain-defined tables.
- Dimensions = denormalized entities with descriptive attributes.
- "Flattened enough" for speed, but maintains clarity.

Star schemas are ideal for:

- BI dashboards
- business metrics
- OLAP workloads
- fact-level aggregations
- query performance with columnar storage

*Why interviewers love Star schemas:*

They reveal whether you understand the *grain*, business logic, *SCD patterns*, and *query patterns,* not just shapes on a whiteboard.

## ***Snowflake Schema - Controlled Normalization for Scalability***

Snowflake = Star schema **+** *normalized dimensions.*

You normalize dimensions when:

- dimensions become *too large*
- slow-changing reference data needs history
- storage and update efficiency matter
- minimizing duplication reduces governance overhead

Examples:

Splitting `location_dim` into `country_dim`, `city_dim`, `postal_dim`
Splitting product hierarchy into `category_dim`, `subcategory_dim`, etc.

*When FAANG interviewers expect Snowflake:*

When the data model must support **massive dimensions**, strict **data governance**, or multi-layer hierarchical attributes.

## ***The Interview Mindset***

The interviewer does *not* care which one you pick.

They care whether you can answer:

- *"Why did you pick this model for this specific workload?"*
- *"How does this choice reduce cost?"*
- *"How does this improve query performance?"*
- *"How does this scale to billions of rows?"*
- *"What trade-offs are you making?"*

They're not checking for definitions, they're checking for *architecture thinking*.

## Fact tables & Dimension Tables

In real interviews, companies don't care whether you can *recite* the definition of a fact or dimension table. They want to see if you understand ***why*** these structures exist and how they support analytical decision-making at scale.

### ***Fact Table***

A Fact Table captures the *events* happening in the business i.e ***measurable actions*** like orders, payments, clicks, rides, impressions. It answers:

> ***"What happened?"***

Facts are typically grain-specific, additive or semi-additive, and are designed at the lowest granularity possible so downstream teams can slice them however they need.

Interviewers often test whether you can correctly define the grain before modelling anything else.

### ***Dimensions Table***

A Dimension Table captures the *context* around those events i.e customers, products, time, locations, campaigns. It answers:

> ***"Who/what/where/when did the event relate to?"***

Dimensions enable slicing, filtering, and grouping. They should be built with descriptive attributes, slow-changing strategies, and clear surrogate key logic.

What matters in interviews is not the dictionary definition, it's whether you can show the interplay between facts and dimensions. Companies want engineers who understand how these structures support billions of events, evolving schemas, SCD management, late-arriving data, GDPR deletes, and downstream ML/BI workloads.

If you can demonstrate that ***facts track behavior*** while ***dimensions track meaning,*** and that your modelling choices preserve analytical correctness under scale and business ambiguity, you're already above 90% of candidates.

## Surrogate keys vs Natural keys

In Data Modelling Interviews, the difference between surrogate and natural keys isn't a memorization test, it's a reasoning test. Interviewers want to see whether you can think in terms of *data stability*, *governance*, *scalability*, and *long-term maintainability* rather than "textbook definitions."

### ***Natural Keys (Business Keys)***

Natural keys come directly from the source system eg: user_id, product_id, email, product_code, booking_id, device_id, etc. They have business meaning and often exist before your data warehouse.

*When they work well:*

- The value is *stable for life* (like a vendor-provided transaction ID).
- The domain guarantees *global uniqueness.*
- Downstream consumers expect to use the same identifiers used in upstream systems.

*Typical issues:*

- They change. Emails get updated, phone numbers change, SKU formats evolve.
- They are inconsistent across systems. What CRM calls a "customer_id" may not match marketing's definition.
- They leak PII. Emails / phone numbers create governance and compliance headaches.
- They create performance hotspots (long strings, wide keys).

Companies have learned the hard way that relying on natural keys leads to unpredictable breakages, reprocessing, and schema drift. That's why natural keys alone almost never power large-scale analytics anymore.

### ***Surrogate Keys***

A surrogate key is an *artificial, meaningless, technical identifier* you generate: integer sequences, UUIDs, hash keys, etc.

They don't exist in the source system; they exist because you want control.

*Why they matter in real-world systems:*

- They remain stable even when business keys change.
- They avoid leaking PII.
- They keep join operations fast, lightweight, and scalable.

- They provide unified identity across messy source systems.
- They enable SCD Type 2 history tracking cleanly.

Most modern warehouses (Snowflake, BigQuery, Databricks) heavily encourage surrogate keys for performance and governance reasons.

## ***The Interview-Level Thinking (This is what interviewer evaluates)***

You must show that you understand when to use each, and why:

*Use Natural Keys When:*

- The source system defines a globally stable ID (e.g., payment_provider_id).
- The business wants traceability back to the original system.
- No PII concerns exist.

*Use Surrogate Keys When:*

- You need consistent identity across multiple systems.
- The natural key can change.
- You're designing dimensions (Dimension tables almost always need surrogate keys).
- You need SCD2 history, high-performance joins, and long-term stability.

*How to Express This in an Interview*

Instead of saying:

> "A surrogate key is an artificial key we generate."

You say:

> "I use surrogate keys when I need long-term stability, cross-system identity resolution, and fast analytical joins especially for SCD2 dimensions.
>
> Natural keys remain important for traceability, but they're rarely reliable or stable enough on their own in high-scale analytical systems."

This shows maturity, not memorization.

## Slowly Changing Dimensions (SCD Types 0–6)

In interviews, SCDs are not tested as “definitions.” Interviewers want to see whether you understand *when*, *why*, and *how* to use each type depending on business rules, audit needs, compliance, and downstream consumption.

Think of SCDs as *data contract choices:*

“How should historical changes in important business attributes be captured so analytics stays correct, trustworthy, and explainable?”

Below is the interview-grade breakdown.

### *SCD Type 0 - Fixed Attribute (Never Changes)*

These are attributes the business says should *never* be overwritten.

*Example:*

A product’s original launch date, a customer’s signup channel, or an employee’s first manager.

In Interviews:
You show maturity by saying:

> “Even if upstream sends an incorrect update, Type 0 attributes must not be overwritten because they represent historical truth. I’d enforce this via validation rules in my ingestion logic.”

### *SCD Type 1 - Overwrite with Latest*

No history. Latest values overwrite old values.
Used when the past doesn’t matter for analytics.

Example:

Customer’s email, phone number, or preferred language.

In Interviews:
You show maturity by saying:

> “If the business only cares about the current value, Type 1 is perfect and keeps the table light. But I’d confirm with stakeholders because choosing Type 1 incorrectly can lead to irrecoverable loss of history.”

## ***SCD Type 2 - New Row for Every Change (Full History)***

Creates a new record for every change, preserving historical versions.

Example:

Customer address changes, product price updates, employment role transitions.

In Interviews:
You show maturity by saying:

> "Type 2 is correct when historical accuracy affects reporting - - revenue attribution, churn analysis, fraud signals.
>
> ⭐But I'll add effective_start_date, effective_end_date, and is_current flags for reliable joins."

## ***SCD Type 3 - Store Limited History***

Stores current + previous value in the same row.
Keeps *some* history, not full.

Example:

A hotel brand moving customers from bronze → silver tier and wanting both values for comparison.

Interview reasoning:

> "Type 3 works only when asking limited 'before vs after' questions. For anything deeper, Type 2 is safer."

## ***SCD Type 4 - Historical Table + Current Table***

The main table stores current value; a second table stores full historical changes.

Use case**:**

Regulatory or compliance-driven systems that need an audit table separate from the main dimension.

Interview reasoning:

> "Type 4 is used when history is required but we don't want the dimension table to explode in size, or when auditors require isolated historical logs."

## ***SCD Type 5 - Type 1 + Type 2 Hybrid***

The dimension table uses Type 1 for fast queries, but a Type 2 history table exists as well.

Use case:

High-traffic BI systems where the dimension must stay compact, but full history is still needed behind the scenes.

Interview reasoning:

> "I'd choose Type 5 when performance in fact joins matters but we can't lose history. It's common in enterprise warehouses where the BI layer must stay clean."

## ***SCD Type 6 - Type 1 + Type 2 + Type 3 Hybrid***

A combination of overwriting, history rows, and previous values.

Use case: Complex analytical systems where:

- you need history for audit,
- current values for BI,
- previous values for trend features (ML).

Interview reasoning:

> "I'd use Type 6 when multiple types of consumers require different forms of the attribute - ML wants signals, BI wants current state, Finance wants history. Type 6 acts as a swiss-army knife."

## ***How to Impress FAANG Interviewers With SCDs***

Instead of listing definitions, do this:

> *1. Tie SCD choice to business rules*
>
> "If marketing needs to understand historical customer behavior, I'd choose Type 2. If they only care about current email, Type 1 is sufficient."

*2. Mention downstream consumers*

“If the ML team relies on historical behavior patterns, Type 2 or 6 is mandatory.”

*3. Mention storage vs performance trade-offs*

“Type 2 increases storage and join cost. If that becomes expensive, Type 4/5 can reduce table bloat.”

*4. Show awareness of GDPR*

“For PII fields, SCDs need extra care - a Type 2 record must be fully deleted if a user requests ‘forget me’.”

***Quick Interview Cheat Sheet:***

| Scenario | Use |
|---|---|
| Attribute never changes | Type 0 |
| Only latest value matters | Type 1 |
| Need full point-in-time history | Type 2 |
| Need current + 1 previous | Type 3 |
| Large dimension - keep current separate from history | Type 4 |
| Need full history + fast single-row access | Type 6 |

## Granularity selection

Granularity is one of the most high-leverage decisions in data modeling because it shapes cost, query performance, scalability, and how much business truth your model can capture. In interviews, they aren't looking for a textbook definition, instead they want to see whether you understand the *trade-offs* behind "how detailed should each record be?"

At its core, granularity means choosing the ***lowest level of detail*** your fact table will store. And this decision depends entirely on *how the business measures reality*.

If you choose a granularity that's ***too fine***, your tables explode in size, costs jump, and queries slow down.
If you choose a granularity that's ***too coarse***, you lose business accuracy, can't run key metrics, and break downstream use cases.

So interviewers look for reasoning like this:

- "If analysts ask questions at the event level, I store facts at the event level."
  (e.g., each click, each impression, each transaction)

- "If the system needs daily summaries only, I'll model at day-level to reduce storage and cost."

- "If we have both use cases, I'll maintain two granularities: atomic facts + aggregated marts."

In real FAANG systems, the guiding principle is:

> *""Model data at the level where business logic fundamentally happens. Aggregate later.""*

This ensures flexibility for future use cases, allows data scientists to recompute features, and supports backfills or late-arriving data without corrupting metrics.

When you explain granularity decisions this way i.e tied to business logic, cost trade-offs, and flexibility, interviewers immediately know you're thinking like an experienced data engineer, not someone repeating definitions.

## Data Contracts (Modern Modelling Requirement)

Data contracts have become a *non-negotiable* part of modern data modelling because companies no longer tolerate pipelines that break every time a source team changes a column.

Interviews evaluate whether you understand that modelling is not just designing tables, it's designing agreements between producers and consumers so the entire data platform remains stable as products evolve.

A data contract defines what data is *produced*, how it is *structured*, how *often* it arrives, how it should be *validated*, and what *guarantees* exist if something changes. It converts "best-effort data sharing" into a strict, enforceable API.

In interviews, the expectation is that you go beyond schemas and talk about *governance plus resilience*:

how your model stays reliable across schema changes, upstream releases, third-party disruptions, and new product features. Senior teams want to hear that you don't build pipelines around hope; you build them around contracts.

***What a Data Contract Includes***

*1. Schema Guarantees*

- Explicit datatype, nullability, allowed values, and semantic rules.
- Example: `event_timestamp` *must be an epoch in ms, never null.*

*2. Versioning Rules*

- Backward/forward compatibility expectations.
- Additive changes allowed; breaking changes require new version + deprecation window.

*3. Delivery Guarantees*

- SLA for freshness, volume, late data tolerance, and retry expectations.
- Example: *Events must arrive within 15 minutes; 1% late arrival tolerated.*

*4. Quality Expectations*

- Validations for duplicates, referential integrity, and business logic.
- Example: `order_amount > 0`, `order_status` *must match enum list.*

*5. Operational Expectations*

- How producers communicate schema changes.
- Release process: PR + contract review + test environment + rollout.

## ***Why This Matters in Modelling Interviews***

Teams want to see if you understand that good models *collapse without stable inputs*. A schema diagram means nothing if a source system suddenly:

- drops a column,
- changes boolean to string,
- starts sending corrupted IDs,
- floods the system with duplicates,
- stops sending late-arriving updates.

A strong candidate talks about *protecting the model at the boundary*, not just inside the database.

## ***Why Companies Now Require It***

Because modern data systems are:

- distributed,
- real-time,
- multi-team,
- cross-product,
- high-scale,
- globally compliant (GDPR/CCPA).

A schema-only approach collapses here.
A contract-based approach *scales*.

## ***How to Express This in Interviews***

❌ *Weak answer:*
"I'll create a star schema with fact and dimension tables."

✔️ *Strong answer:*
"My fact table depends on stable identifiers, so I'd enforce a data contract with the source team: strict schema, versioning guarantees, nullability rules, and validation logic. The pipeline will reject breaking changes at ingestion and notify producers before the downstream model breaks."

This instantly signals *architecture maturity*

# Modelling Workflow Mindset - The 5-Step Mental Framework Interviewers Expect

Before you even start drawing tables or picking keys, top companies expect you to think like an engineer who can design *repeatable, scalable workflows*. Data modelling is not about memorizing patterns, it's about showing that you can take an ambiguous business problem, break it into structured steps, and reason your way to a robust model.

FAANG interviewers don't care if you reach the "perfect" schema.
They care whether your *process* is systematic, logical, and production-ready.

This section teaches a ***5-step modelling workflow*** that every strong Data Engineer uses instinctively. It keeps you aligned with business goals, ensures modelling consistency across teams, and proves to interviewers that you think beyond tables and you think in systems.

This mindset is what separates average candidates who "draw schemas" from exceptional candidates who can design models that scale with real products, real traffic, and real ambiguity.

## 1. Clarify Business Requirements

Before touching schemas or talking about fact tables, top-tier interviewers judge how well you understand the *actual business problem*. FAANG companies expect data engineers to translate ambiguous goals into precise modelling requirements; not jump straight into designing tables.

A strong candidate starts by slowing down and asking sharp, scope-defining questions:

***What decision or metric will this model support?***

- ➢ Is it churn prediction, daily revenue accuracy, recommendation features, operational KPIs?
- ➢ The *business output* dictates the entire modelling strategy.

***Who are the consumers and how will they use this data?***

- ➢ BI analysts with ad-hoc SQL needs? A real-time service? ML training pipelines?
- ➢ Different consumers → different granularity, latency, and table design.

***What is the source of truth for each entity?***

- ➢ Which system owns customer attributes? Orders? Events?
- ➢ This avoids mixing inconsistent upstream definitions - a mistake FAANG interviewers catch instantly.

***What are the edge cases that break naive models?***

- ➢ Cancelled orders, reassignments, multi-device users, anonymized IDs, retries, late events.
- ➢ Interviewers want to see whether you *anticipate data messiness*, not just handle the happy path.

***What level of freshness and accuracy is required?***

- ➢ Is daily fine? Do we need near real-time? Is eventual consistency acceptable?
- ➢ This affects whether you choose batch models, streaming enrichment, or hybrid patterns.

By asking these questions upfront, you show three things interviewers cares about:

- clarity of thought,
- ability to reason under ambiguity, and
- understanding of downstream impact.

Only then should the modelling begin.

## 2. Identify entities

Once you understand what the business is trying to do, your next move is to lock down what actually exists in this world – the entities.

Think of entities as the *stable nouns* in the story that deserve their own tables because they have identity, lifecycle, and are reused across use cases.

In an interview, I'd walk through it like this:

- ***Start from the narrative, highlight nouns***

  *"In this problem I see clear candidates for entities:* `Customer`*,* `Order`*,* `Product`*,* `Store`*, and time-based* `Event` *like* `PageView` *or* `Transaction`*."*

  This shows you're translating business language into a data model.

- ***Separate three types of entities***

  I'd explicitly say:

  - *Core business entities* – customers, merchants, riders, drivers, accounts, subscriptions.
  - *Process / event entities* – orders, trips, sessions, payments, clicks. These often become facts.
  - *Reference entities* – countries, currencies, product categories, plans, statuses. These become dimensions / lookup tables.

- ***Decide: entity vs attribute vs relationship***

  Interviewers care that you don't blindly turn every noun into a table. I'd say something like:

  *"Loyalty tier feels like an attribute on* `Customer`*, not a separate entity. But* `Subscription` *is an entity because it has its own start/end dates, status, plan, and billing cycles."*

- ***Check cross-team reuse and ownership***

  A good modelling signal:

  *"Anything used across multiple domains – like `User`, `Organization`, `Product` – I'd model as first-class entities with stable IDs, because analytics, billing, and ML will all depend on them."*

- ***Tie it back to the interview case***

  For example, in a ride-hailing case:

  *"Here my main entities would be: `Rider`, `Driver`, `Trip`, `City/Region`, and `Payment`. `Trip` is the central process entity, `Rider` and `Driver` are core entities, and `City` is reference."*

If you speak in this way, you're not just "listing tables", you're showing that you can *systematically* extract the right entities from messy product requirements, which is exactly what Interviewers look for.

## 3. Define grain / granularity

When you move from “what are the entities?” to “how should they be stored?” ***granularity*** becomes the single most important modelling decision because it controls accuracy, performance, cost, and how future-proof your model will be.

In interviews, companies want to see whether you can lock the grain early, justify it, and understand its downstream impact. Candidates who skip this step usually design tables that can never scale or answer real queries.

### ***What “grain” actually means***

Grain is the exact level of detail represented by one row.

It answers:

> *“What does one record actually stand for?”*

Examples:

- *One order line per row*
- *One click event per row*
- *One daily aggregated metric per customer per row*

### ***How Interviewer expects you to reason***

Big companies look for whether you understand that the grain determines:

- Storage cost (event-level vs aggregated)
- Query cost (sum 3B rows vs sum 3M aggregated rows)
- Flexibility (fine-grained stores allow future use cases)
- Accuracy (wrong grain → duplicated metrics, inconsistent KPIs)
- Join complexity (grain mismatch = broken queries)

They don’t care about definitions, they want to see *trade-offs*.

## ***How to choose the correct grain - interview-ready reasoning***

*1. Start from the business question*

Ask: *"What questions must this data answer today and in the future?"*

If you need product-level insights, store item-level events.
If you only need daily summaries, store daily aggregates.

But in interviews, the safe answer is usually:

> "Keep the most granular atomic events unless there is a clear business constraint."

*2. Evaluate performance vs flexibility*

- Atomic grain = maximum flexibility, higher cost
- Aggregated grain = cheaper, but less reusable

Demonstrate this reasoning explicitly.

*3. Avoid mixed-grain tables*

A common mistake junior candidates make is allowing a table to store "sometimes event-level, sometimes aggregated" data. This destroys modeling integrity.

*4. Show you understand downstream impacts*

Explain how grain affects:

- deduplication
- SCD application
- fact table size
- clustering & partitioning
- dashboard performance
- ML feature generation

This shows senior maturity.

*5. State the grain clearly before designing the schema*

Interviewers want to see you say something like:

> *"Before designing anything, let me define the grain:*
> *One row will represent a single user event with a unique event_id.*
> *This gives us fine-grained traceability and supports both analytics and model training."*

## 4. Define Relationships

Once you know what the entities are and at what grain they live, the next step is to make their relationships explicit.

In interviews, don't just say "there's a customer and an orders table."

Say *how* they relate:

- one-to-many (one customer → many orders),
- many-to-many (user ↔ device), or
- one-to-one (user ↔ KYC profile), and
- whether the relationship is optional or mandatory.

This is where you show that you're thinking about *cardinality*, *optionality*, and real business rules, not just drawing lines between boxes.

For each relationship, explain what it means for the data and the queries. For example:

- "Customer to Orders is one-to-many. I'd keep `customer_id` as a foreign key on the fact table, which lets me aggregate by customer, segment, region, etc."

- "Users and devices are many-to-many. I'd introduce a bridge table `user_device_bridge` keyed by `(user_id, device_id)` so I can answer questions like 'how many active devices per user?' without duplicating attributes."

Good answers also call out modelling choices:

- ➢ when you denormalize relationships into the fact (e.g., store `store_id` and `merchant_id` directly on the transaction fact for performance),
- ➢ when you use a bridge table (multi-select dimensions like tags, interests, categories), and
- ➢ how relationships behave over time (e.g., a customer changing segment, store reassigned to another region).

Interviewer is listening for this kind of relational thinking: *"I understand who owns the relationship, where the foreign keys live, what the join paths are, and how that impacts correctness, performance, and future changes."*

## 5. Validate against query patterns

Once you've drawn the model, don't stop there.The last step is to pressure-test it against real query patterns and this is exactly what strong interviewers look for.

You take the core questions and KPIs you gathered earlier and walk them through the model:

- Can I get "daily active users by region and device" with a single fact table + a few dimensions, or do I need ugly subqueries and workarounds?

- If a product wants "cohort retention by signup month and marketing channel," does my grain (user-day, session, event, order…) make that natural, or does it explode into fan-out and double counting?

- For "last known customer segment as of a given date," do my SCD choices and date fields support point-in-time correctness, or will I misattribute changes?

Think in terms of SQL you'd actually write:

*"If I join Fact_Orders → Dim_Customer → Dim_Date and group by X/Y, is this straightforward, performant, and unambiguous?"*

If not, you adjust the model: maybe introduce a *bridge table*, a helper aggregate table, or change the fact grain.

In an interview, this is how you show maturity: you don't just draw tables, you prove your design by showing that it cleanly supports the real queries the business will run.

# Advanced Modelling Concepts (High-Level Theory)

So far we have seen how to think: clarify business context, lock grain early, design for SCD correctness, and reason from first principles.

Now let's go one step ahead and look concepts that separate a "good" data modeller from a FAANG-caliber one.

At top tech companies, interviewers don't test just whether you know facts, dimensions, or 3NF trade-offs, they test whether you understand the *theory behind modern, large-scale analytical systems*.

They want to know whether your models remain correct, scalable, and resilient when the real world becomes messy: multi-device users, cross-channel identity, late-arriving events, GDPR deletions, schema drift, high-cardinality attributes, and ML-driven feature stores.

Most candidates fail modelling rounds because they try to solve advanced problems using phase 1 fundamentals only.

FAANG teams expect more than that. They expect you to think like an architect.

This phase teaches the *high-level modelling theory* required to design and defend systems at that level, the concepts that appear in virtually every real interview loop:

## 1. Event Modelling / Activity Schema Patterns

*How  Companies Expect You To Think About Clickstream, Sessions, Impressions & App Activity*

Modern high-scale products are event-driven by nature.

Every user interaction - a scroll, tap, click, swipe, play, pause, search, impression, purchase is an *event*. These events feed:

- Growth experimentation (A/B tests)
- Recommender systems
- Ad delivery + bidding systems
- User journey analytics
- Real-time monitoring
- Sessionization
- Feature stores

Event modelling is one of the *most heavily tested areas* in FAANG interviews because it reveals whether you understand:

- high-cardinality identifiers
- schema evolution
- late/out-of-order event handling
- write patterns at scale
- partitioning/clustering strategies
- deduplication logic
- session modelling
- event-level vs aggregated tables
- reproducibility + query performance

A junior candidate talks about “storing events.”

An experienced candidate talks about *event semantics, ingestion guarantees,* and *modelling decisions under real-world constraints.*

## *A. Event Schema Patterns (Clickstream, Sessions, Impressions, App Activity)*

### *1. Clickstream Events (Atomic Events)*

These are the raw behavioral breadcrumbs: every interaction a user performs.

**Examples:**

- page_view
- button_click
- add_to_cart
- checkout_start
- search_query
- product_impression
- scroll_depth

**What interviewers evaluate:**

- Can you define the *atomic grain*?
- Do you understand that each event must have a *unique event_id*?
- Can you ensure ordering using event_timestamp and ingestion_time?
- Do you model multiple devices per user?
- Can your design tolerate schema evolution?

*Typical Schema*

```
-- event_id (UUID)
-- user_id
-- device_id
-- session_id
-- event_type
-- event_timestamp
-- ingestion_timestamp
-- attributes (semi-structured)
```

*Insights*

Raw events must be immutable.
Corrections should be added as *new events* - not updates - to maintain replayability.

*2. Sessions (Sessionization Layer)*

Sessions are *analytical constructs*, not source events.

A session groups events into a logical window:

- “Actions performed by a user within 30 minutes of inactivity”
- “Series of events between app open → app close”
- “Streaming watch sessions (start → stop → pause → resume)”

*Interview Expectation:*

They want to see whether you know:

- sessions are *derived*, not produced
- sessionization rules vary by product
- event ordering must be deterministic
- session boundaries drive metrics like DAU/MAU, bounce rate, retention

*Typical Schema:*

```
-- session_id
-- user_id
-- device_id
-- session_start_ts
-- session_end_ts
-- event_count
-- session_duration
-- traffic_source
```

*Key modelling point:*

Session tables require *idempotent processing*, because late events can extend a session or split one into two.

*3. Impression Events (Advertising + Recommender Systems)*

Impressions appear in systems where items (ads, videos, products) are *shown* to users.

Characteristics:

- very high volume
- need strong dedupe logic
- often require prediction metadata (ranking score, model version, feature vectors)
- must support A/B test attribution

Interview angle:

They want to know whether you can handle:

- dedupe via impression_id or (user_id + item_id + timestamp_bucket)
- enrichment with model metadata
- point-in-time consistency for attribution

*4. App Activity (Device Events)*

These include:

- app_open
- app_close
- push_notification_received
- push_notification_opened
- background_foreground transitions

These events often drive:

- retention metrics
- churn prediction
- mobile performance monitoring

*Interview signal:*
Show that you understand different event semantics:
"app_open may not always correspond to a true user action, could be OS-level preloading."

Demonstrating semantic clarity is a senior-level skill.

## ***B. Choosing Grain: Event-Level vs Aggregated Rollups***

This is a **mandatory discussion** in interviews.
Interviewers expects you to justify your grain with business logic, cost considerations, and analytical flexibility.

### *1. Event-Level Grain (Atomic Events)*

"One row = one user action."

Pros

- Maximum flexibility
- Supports future metrics
- Perfect for ML feature generation
- Enables session reconstruction
- Enables time-travel and reprocessing
- Accurate for behavioral analytics

Cons

- Expensive at scale (billions of rows/day)
- Query cost can be very high
- Requires careful partitioning
- Needs dedupe + ordering guarantees

Interview-ready answer:

"I prefer atomic grain unless there's a hard storage or cost constraint because it allows full replayability, richer experimentation, and long-term analytical flexibility."

*2. Aggregated Rollups (Daily/Hourly/User-Level Summaries)*

“One row = one aggregated behavior snapshot.”

Examples:

- user_day_activity
- pageview_hourly_summary
- session_metrics

Pros

- Very fast for dashboards
- Much lower storage
- Good for product teams needing quick KPIs

Cons

- You lose event semantics
- Cannot re-derive new metrics later
- Harder to debug anomalies
- Impossible to reconstruct sessions from rollups

Interviewer expectation:

You must always emphasize **both**:

1. Keep atomic events for correctness
2. Create rollups for performance

This dual-layer model is exam gold.

*3. Mixed-Grain Tables (Anti-Pattern)*

Interviewers look for this mistake and expect you to call it out.

❌ One table storing some events at event-level and some aggregated metrics.
❌ Causes double counting, inconsistent metrics, incorrect joins.
❌ Hard to debug, version, or partition.

Always call it out explicitly in interviews.

## *C. Partitioning Strategies & Write Patterns (FAANG-Grade)*

Handling billions of daily events requires very intentional partitioning + ingestion design.

This is where senior candidates stand out.

### *1. Partitioning Strategies*

#### A. Time-Based Partitioning (Most Common)

Partition by:

- event_date
- event_hour

Why?

- Predictable arrival pattern
- Efficient pruning
- Balanced partitions
- Works well with out-of-order timestamps

Interview Nuance:
Partition by ingestion_time *not* event_timestamp when late/out-of-order data is common.

#### B. User-Based or Device-Based Partitioning (Only When Necessary)

Partition by:

- user_id bucket
- device_id bucket

Used when:

- workloads are ML-driven
- feature stores optimize reads by entity
- user-level distribution is balanced and hashed

Risk:

- uneven distribution
- partitions becoming hotspots
- complex compaction/clustering needed

### C. Hybrid Partitioning (Time + Hash Bucket)

Example:

- date = 2024-06-01
- user_bucket = hash(user_id) % 32

This achieves:

- temporal pruning
- balanced distribution
- scalable writes

## *2. Write Patterns*

### A. Append-Only (Correct for Event Logs)

All raw events should be appended, not updated.
This supports:

- immutability
- replayability
- idempotent processing

### B. Late Data Write Rules

A strong candidate discusses:

- how to insert late events
- how to avoid corrupting aggregates
- whether to maintain a DLQ (Dead Letter Queue)
- whether to maintain a late_event table
- whether to reprocess only impacted partitions (Delta/Iceberg/Hudi ability)

### C. Deduplication Patterns

FAANG events often arrive duplicated due to retries.

Common dedupe patterns:

- event_id UUID
- compound key (user_id, event_type, event_timestamp_bucket)

- deterministic HashKey on event payload

Show that you understand:

- dedupe at Bronze or Silver layer
- cost-effective dedupe
- idempotent upserts

### D. Compaction & Clustering

For high-write workloads:

- small-file compaction
- Z-ORDERing (Delta)
- clustering columns (BigQuery/Redshift/Snowflake)

*Interview Tip:* Mention how compaction reduces read cost and improves query pruning.

## 2. Identity Resolution & Customer 360 Modelling

What Interviewers Expect You To Model, Not Implement

Identity Resolution is one of the most important modelling concepts at FAANG-scale companies because no large system has a single, clean "*customer_id*."

Users appear under multiple identifiers across devices, channels, and systems.
Your job in an interview is to show that you can model a unified identity layer that is stable, consistent, and future-proof.

This section focuses only on modelling, not pipeline mechanics. If you wanna learn more about the System Design part of such problems, check out my FAANG Level System Design for Data Engineers guide.

## *A. MDM Basics (Modelling Perspective)*

Companies expect you to understand MDM (Master Data Management) not as a product but as a *conceptual modelling layer*.

What MDM means in a modelling interview:

- A canonical, company-wide definition of a customer/entity
- A single golden record that unifies all sources
- A system that keeps history, traceability, and cross-system identity mapping

Core modelling expectations:

1. You define a master entity table (dim_customer / customer_hub).
2. You keep stable surrogate keys for consistent joins.
3. You maintain source-specific identifiers for traceability.
4. You allow multi-record history for attributes that change (SCD2).
5. You model relationships between identifiers and the master entity.

A strong candidate shows awareness that identity is *messy* and modelling solves the mess.

## *B. Surrogate Key Strategies (The Heart of Identity Modelling)*

Interviewers want to know whether you can create stable, scalable identity keys even when natural identifiers change or disagree.

Surrogate key expectations:

- Generated by you, not by source systems
- Immutable, numeric or UUID
- Used for all joins in the warehouse/lakehouse
- Supports SCD2 history cleanly
- One surrogate per business entity (not per source system)

The interview-ready reasoning:

> *"I assign a surrogate customer_key for long-term stability. All incoming identifiers map to it, even if emails, phone numbers, or device IDs change."*

And:

> *"I never rely on email or phone as keys because they change and are not globally unique."*

This demonstrates senior-level thinking immediately.

### ***C. Cross-System Mapping (Identity Graph Modelling)***

Every big company has multiple systems that disagree on what a "customer" is:

- CRM
- Billing
- Marketing automation
- Mobile app backend
- Web identity provider
- Loyalty systems

Your model must stitch these systems together.

How to model cross-system mapping in interviews:

Create a "mapping table" or "bridge entity"

Example: customer_identity_bridge

Contains:

- surrogate_customer_key
- source_system
- source_customer_id
- effective_start_date
- effective_end_date
- is_current

Why interviewers expect this:

It allows:

- Traceability back to all source systems
- Multiple identifiers tied to one master user

- Historical tracking when mappings change
- Decoupling upstream chaos from downstream analytics

Answer pattern interviewers love:

> *"I model a bridge table so each source identifier maps to the surrogate customer key. This lets me unify data across systems without relying on any unstable natural key."*

## *D. Handling Many Identifiers (email, phone, device_id, cookie_id)*

*This is the most common modelling question.*

*Identity is fractured across channels and devices, and your model must represent that fragmentation cleanly.*

How Interviewers expects you to reason:

*1. Emails / Phones → PII + changeable*

- *Treated as attributes, NOT keys*
- *Managed with SCD2*
- *Stored in a customer_contact table:*

```
-- customer_contact_key
-- customer_key
-- contact_type (email/phone)
-- contact_value
-- is_primary
-- effective_start_date
-- effective_end_date
```

*2. Device IDs → many-to-one relationships*

*A user can have many devices → model as:*
*customer_device_bridge(customer_key, device_id, device_type, first_seen_date)*

*3. Cookie IDs → highly volatile, many-to-many*

*Cookies change frequently; model them separately:*
*web_identity_bridge(customer_key, cookie_id, first_seen, last_seen)*

*4. Identity stitching logic (modelling view only):*

- *Multiple identifiers must be able to map to the same customer_key*
- *The model must allow reconciliation when two identifiers are later discovered to belong to the same person*
- *Require surrogate keys and versioned mappings*

*Interview-ready line:*

> *"Emails, phones, cookies, and device IDs are attributes or secondary identifiers, never primary keys.*
> *They belong in bridge tables with versioning so they can be unified or split as identity evolves."*

## 3. Data Quality, Referential Integrity & Constraints

How companies Maintain Modelling Integrity Without Traditional RDBMS Constraints

Modern warehouses/lakehouses (BigQuery, Snowflake, Redshift, Databricks) do not enforce constraints the same way OLTP systems do.

Interviewers want to see whether you understand this shift and how modelling compensates for it.

Your job in the modelling round is to explain how you design data models that remain correct even when the engine does not enforce integrity for you.

### ***A. How Constraints Differ in Warehouses (Modelling Perspective)***

Traditional OLTP systems enforce:

- Primary keys
- Foreign keys
- Unique constraints
- Check constraints
- Not-null rules

But analytical warehouses often:

- *allow* constraints to be declared
- *do not enforce* them physically
- rely on the modelling layer + pipeline logic to ensure correctness
- prioritize scan performance over strict referential guarantees

What interviewers expect you to say:

> "In analytical warehouses, constraints are rarely enforced at the engine level.
> So modelling decisions must assume the warehouse won't protect integrity for us."

This shows you understand real-world architecture, not textbook SQL.

## ***B. Soft vs Hard Constraints (Core Modelling Concept)***

This distinction is **heavily tested** in FAANG interviews.

*1. Hard Constraints (Enforced by the Engine)*

Examples in OLTP:

- Primary key must be unique
- Foreign key must exist in parent table
- Column must be NOT NULL

These guarantee correctness by force, which is great for transactions but expensive for analytical workloads.

Most warehouses *do not enforce* them because:

- foreign key checks slow down massive writes
- distributed engines can't enforce relational integrity cheaply
- analytical workloads accept "eventual consistency"

*2. Soft Constraints (Enforced by Modelling + Logic)*

Soft constraints are *not enforced by the database*, but by:

- model design
- ingestion rules
- schema contracts
- dedupe logic
- SCD strategies
- business rules
- validation layers

Examples FAANG expects you to mention:

- ensuring surrogate keys are generated by the model
- enforcing referential integrity during ingestion (reject or quarantine bad data)
- enforcing uniqueness via dedupe logic in fact tables
- modelling relationships so invalid references are impossible (e.g., using mapping tables)

*Interview-level explanation:*

> *"In warehouses, constraints are soft by default.*
> *We enforce correctness through modelling patterns, not database locks."*

## *C. How Teams Design for Integrity Without RDBMS Constraints*

This is *the key part*.

Interviewers want to see how your modelling choices keep data trustworthy without PK/FK checks.

Here are the modelling patterns teams expect you to understand:

*1. Surrogate Keys for Stable Identity (Prevents orphan facts)*

A surrogate key in dim_customer ensures:

- consistent identity
- no PII-based keys
- stable FK references from fact tables

This acts as a “soft primary key.”

*2. Bridge / Mapping Tables (Prevents broken relationships)*

When multiple systems produce IDs:

- you model a mapping table
- every fact table references the canonical surrogate key
- even if upstream IDs are messy, facts never break referential integrity

Interview statement:

> “We preserve referential integrity by modelling a canonical entity and mapping all source system identifiers to it.”

*3. SCD2 Dimensions (Prevents temporal joins from breaking)*

Modelled effective date ranges maintain integrity across time:

- fact.timestamp between start_date and end_date
- ensures correct point-in-time joins
- prevents mismatched foreign keys for historical facts

This is a *temporal integrity constraint*, enforced by modelling.

*4. Immutable Fact Tables (Prevents data corruption)*

Facts are append-only:

- no destructive updates
- no foreign key rewrites
- no referential gaps from reassignments

*5. Data Contracts + Schema Validation (Warehouse substitute for CHECK constraints)*

Warehouse engines don't enforce:

- allowed values
- nullability
- enum rules

So teams rely on:

- schema registries
- data contracts
- column-level semantic rules
- "reject, quarantine, or warn" strategies

This preserves modelling integrity through contract-based governance.

*6. Partition + Clustering Strategy (Prevents duplicate or missing rows)*

This is not about performance alone.
It also protects modelling quality by controlling where data lands:

- event_date partition ensures completeness checks
- Z-ORDER/clustering makes dedupe + consistency checks efficient
- partition boundaries prevent cross-contamination of fact batches

*7. Identity Resolution Tables (Prevents duplication across systems)*

Customer 360 models create:

- a single surrogate key
- traceable mappings
- versioned identifiers

This prevents the “two customers that are actually the same person” problem - the biggest integrity challenge at big tech.

# Architecture Patterns for Modelling

If Phase 1 taught you the fundamentals and Phase 2 taught you advanced modelling theory, Phase 3 now moves into architecture-level modelling patterns, the frameworks companies use to structure entire analytical ecosystems.

Interviews at senior levels evaluate whether you understand *not just tables*, but architectural modelling philosophies:

- Why a company picks Kimball for BI-heavy workflows
- Why Data Vault becomes essential in large, multi-source, fast-evolving systems
- When strict 3NF is the right choice
- How scale, schema evolution, and real-time workloads shape these decisions

This phase is not about memorizing definitions.
It's about demonstrating when and why a pattern works, what trade-offs it introduces, and how it influences downstream modelling, lineage, governance, and performance.

A strong candidate shows they understand architecture as a modelling choice, not an implementation detail.

By the end of this phase, you'll be able to defend your modelling architecture the same way senior DEs do in design reviews: clearly, rationally, and grounded in scalability.

## 1. Kimball vs Data Vault vs 3NF

### ***A. Where Each Modelling Approach Fits***

Interviewers don't want definitions - they want contextual reasoning. Here's how to frame it:

1. Kimball (Dimensional Modelling - Star/Snowflake)

Best Fit:

- BI analytics
- Dashboarding
- Metric computation
- Slice-and-dice queries
- Stable business processes

Why companies use it:

- Simple for analysts
- Fast for aggregations
- Clear grain + surrogate key patterns
- Low cognitive load

Interview-friendly summary:

> *"Kimball is ideal when the business has well-defined processes and the primary workload is analytical reporting."*

*2. 3NF (Normalized Modelling)*

Best Fit:

- Operational/OLTP systems
- Transactional workloads
- High write concurrency
- Systems of record (CRM, billing, user account services)

Why it exists:

- Avoids anomalies
- Ensures consistent writes
- Minimizes data duplication

Interview summary:

> *"3NF is perfect when the system cares about strict consistency and fast updates - not analytical flexibility."*

*3. Data Vault (Hubs, Links, Satellites)*

Best Fit:

- Large-scale, multi-source enterprises
- Rapid schema evolution
- Merging conflicting identifiers
- Long-term auditability
- High-volume ingestion
- Federated data ownership

Why companies adopt it:

- Highly modular
- Handles changing source schemas
- Captures full history (satellites)
- Easy to extend without breaking existing models
- Decouples identity (hub), relationships (link), attributes (satellite)

Interview-ready line:

> *"Data Vault shines in environments with many sources, messy identifiers, and frequent schema changes because hubs, links, and satellites evolve independently."*

## ***B. Why Data Vault Is Common in Large-Scale Systems***

large-scale data is:

- distributed
- inconsistent
- fast-growing
- multi-source
- audit-heavy
- schema-evolving

This makes Data Vault extremely attractive.

Key reasons:

*1. Identity Chaos → Hubs*

Different systems define "customer" differently → hubs unify identity.

*2. Many-to-many relationships → Links*

Links elegantly represent complex relationships without breaking grain.

*3. Constant schema evolution → Satellites*

Add or modify attributes without altering core entities.

*4. Full history capture → Auditability*

Every attribute change is recorded.
Crucial for fraud, ads, finance, privacy compliance.

*5. Future-proof extension*

Add new systems/data sources without refactoring existing models.

Interview phrase:

> *"Data Vault lets you scale horizontally, new systems become new hubs/links/satellites, not new redesigns."*

## *C. Anti-Patterns Interviewers Expect You to Avoid*

Interviewers watch for these mistakes, calling them out shows maturity.

*1. Using Only Kimball for Rapidly Changing Schemas*

❌ Star schemas break when upstream teams add/change attributes.
❌ Huge dimensions become unmanageable.
❌ SCD2 explodes in size.

*2. Using 3NF for Analytics Warehouses*

❌ Too many joins.
❌ Poor query performance.
❌ Hard for analysts to use.
❌ Not optimized for columnar storage.

*3. Mixing Kimball + 3NF in the Same Layer*

❌ Creates grain mismatch.
❌ Unclear ownership of tables.
❌ Hard to maintain lineage.

*4. Using Natural Keys in Distributed Models*

❌ Emails change
❌ Phone numbers change
❌ Device IDs reset
❌ Systems disagree on customer identity

Always use surrogate keys.

*5. Building Wide "Kitchen Sink" Dimensions*

❌ Hundreds of columns
❌ Volatile attributes
❌ Unrelated domains shoved together

Interviewers expect you to mention this as a modelling anti-pattern.

*6. Treating Data Vault as a BI Schema*

❌ Data Vault is raw modelling, not for end-user queries
❌ Querying directly is slow and difficult
❌ Must be transformed into dimensional marts

Interview bonus point:

> "Vault → Marts → BI is the standard pattern. Vault isn't the final analytical layer."

How to Summarize This in an Interview (30-second version)

> "3NF is best for operational systems, Kimball is best for analytical reporting, and Data Vault is best for large-scale, multi-source architectures with frequent schema evolution.
>
> Data Vault wins at scale because hubs, links, and satellites let you grow without breaking existing models.
>
> Anti-patterns include using Kimball for rapidly changing schemas, using 3NF for analytics, mixing modelling styles, and relying on natural keys."

## 2. Designing for Scale

### *A. Handling Big Dimensions (100M+ Rows)*

Large dimensions are common at FAANG scale: users, devices, merchants, items, SKUs, ads, content catalogs.

Interviewers want to see whether you understand how modelling must change at large cardinalities, not how to tune ETL.

*1. Use Surrogate Keys (Always)*

- Ensures compact joins
- Avoids PII-based keys
- Minimizes storage + clustering cost
- Supports SCD2 cleanly

*Interview line:*

> "At 100M+ rows, surrogate keys are mandatory for join efficiency and evolution."

*2. Normalize When Dimensions Become Too Large*

Very large dimensions should not be fully denormalized.

Examples:

- product → product, product_category, product_attributes
- location → country, region, city, postal-unit

Why?

- Better compression
- Faster selective queries
- Avoids column bloat
- Reduces I/O scanning
- Reduces SCD2 explosions

*Interview line:*

> "When a dimension becomes extremely wide or includes hierarchical attributes, I snowflake it for scalability."

*3. SCD2 Strategy Must Be Efficient*

For huge dimensions, naive SCD2 can explode.

Better patterns:

- Only version attributes that truly change (satellite-style modeling)
- Partition SCD2 tables by effective_end_date or current_flag
- Avoid copying large column sets on every change

*Interview line:*

> "I separate frequently-changing attributes into their own SCD2 satellites to avoid table explosion."

*4. Separate Core Identity from Attributes*

Use a *hub-style* pattern:

- dim_customer (identity + stable attributes)
- customer_attributes_satellite (volatile attributes)
- customer_contact_satellite (email/phone)

This is Data Vault thinking applied in a modelling interview.

## ***B. High Cardinality Issues (session_id, device_id, event_id, cookie_id)***

High-cardinality fields create 3 problems:

- Huge fact tables
- Weak clustering/partition pruning
- Slow joins

Candidates should be able to explain how modelling addresses these.

*1. Avoid Using High-Cardinality Keys as Join Paths*

Do not join facts on:

- session_id
- event_id
- cookie_id
- device_id

These keys are:

- volatile
- unbounded
- high cardinality → poor performance

Instead:

- Join on surrogate keys where possible
- Treat high-cardinality fields as attributes, not join drivers

*Interview line:*

> "I never design join paths around volatile, high-cardinality identifiers. They belong as descriptive attributes."

*2. Use Hash Bucketing for High Cardinality Columns*

For clustering or distribution:

- Hash(session_id)
- Hash(user_id)
- Hash(device_id)

This gives:

- Even distribution
- Predictable performance
- Better pruning

*3. Avoid Storing High-Cardinality IDs in Dimensions*

Examples:

- session_id is not a dimension
- cookie_id is not a dimension
- event_id is not a dimension

Model them as:

- Fact-level attributes
- Bridge tables when necessary

*4. When High Cardinality Is Inevitable*

Create skinny dimension tables (mini-dimensions):

Example: dim_session

- session_id
- user_id
- session_start
- session_end

Lightweight, partition-friendly, low-number-of-columns.

## *C. Pre-Aggregations (Analytical Optimization Layer)*

Pre-aggregations are not ETL topics - they are modelling-layer decisions to make analytical workloads feasible at scale.

Pre-aggregations shine when:

- Fact tables hit billions of rows
- Dashboards query the same metrics repeatedly
- BI teams need sub-second latency

### *1. Build Aggregated Marts*

Examples:

- user_day_activity
- session_metrics
- order_item_daily
- impression_hourly

Grain choices:

- daily
- hourly
- weekly
- per-user/per-product

*Interview phrase:*

“I keep atomic facts for correctness and build aggregated marts for performance-critical workloads.”

### *2. Pre-Aggregations Must Mirror Business Logic*

You must define:

- metric definitions
- windows
- filters
- attribution rules

Why?
Because inconsistent pre-aggregation logic breaks KPIs.

*Interview line:*

> “Pre-aggregations are only correct if the business logic is stable and versioned.”

*3. Use Pre-Aggregations for High-Cardinality Filters*

Instead of scanning billions of events to compute:

- DAU
- MAU
- impressions per campaign
- retention curves

Pre-aggregate to daily or session-level.

Saves cost, time, cluster usage.

*4. Pre-Aggregations Must Be Idempotent + Rebuildable*

FAANG expects you to say this (even in modelling rounds):

> “All aggregated tables must be derived from atomic facts so they can be regenerated if logic changes.”

This is a modelling discipline, not ETL.

## 3. Modelling for Streaming Use Cases

### *A. Real-Time Updates (Modelling for Continuously Changing Data)*

Companies expect you to understand that *streaming does not change the modelling principles*, but it *does change the constraints around correctness and temporal logic.*

*1. Facts Must Be Immutable*

Streaming facts should be append-only, never updated in-place.

This ensures:

- idempotent processing
- replay safety
- consistent event history

Interview line:

> *"In streaming pipelines, facts remain immutable; corrections appear as new events, not updates."*

*2. Use "Upsertable" Dimensions*

Dimensions in streaming systems must support:

- fast updates
- SCD2 history
- late corrections

Model dimensions so that:

- the surrogate key remains stable
- attribute changes produce new SCD2 rows
- current version is easy to query (is_current flag)

*3. Maintain a "current-state" + "historical-state" Structure*

For streaming workloads, interviewers expect a pattern like:

- state table (latest snapshot for fast lookup)
- history table (full SCD2 trail for correctness)

This hybrid design is extremely common at Big Companies.

## *B. Schema Evolution (How to Model Evolving Event Schemas)*

Streaming systems evolve rapidly:

- new event types
- new attributes
- deprecated fields
- new device/platform data
- new ML metadata

A strong candidate shows they can design models that won't break when schemas change weekly.

### *1. Use Semi-Structured Columns for Flexible Attributes*

Example models:

- event_attributes (JSON / VARIANT)
- metadata
- context

This preserves flexibility without forcing schema redesigns.

Interview line:

> *"I model stable fields as columns and put highly volatile fields into a semi-structured attribute column."*

### *2. Version Your Event Types*

Events evolve → version them.

Model fields like:

- event_version
- schema_version

This enables:

- backward compatibility
- proper interpretation of old vs new payloads
- safe schema expansion

*3. Maintain Backward-Compatible Schemas*

Modelling rule:
*Additive changes only.*

If a breaking change is required → introduce a *new event type* or *new version*.

Interview-ready line:

> *"Streaming models must be additive-first. Breaking changes require new versions, not mutated schemas."*

## *C. Late-Arriving Data (Core Modelling Problem in Streaming Systems)*

All Big Data systems have late events due to:

- mobile offline behavior
- retries
- network buffers
- partner feeds
- cross-region replication
- timezone misalignment

Your model must handle this without corrupting metrics or breaking SCD logic.

*1. Store Both Event Time and Ingestion Time*

Two timestamps are essential:

- event_timestamp → real-world user action
- ingestion_timestamp → when system received it

This enables:

- correct ordering
- accurate reprocessing
- watermarking logic
- backfilling only affected partitions

Interview phrase:

> "I always model both event_time and ingestion_time. It's the only way to handle lateness cleanly."

*2. SCD2 Must Consider Late Data*

If a dimension attribute changes and the event arrives late, your model should still:

- insert the correct historical row
- update effective_end_date boundaries
- reassign facts to the correct version

This is a temporal modelling requirement, not an ETL one.

*3. Partitioning Must Not Break Late Data*

Modelling-friendly partitions:

- ingestion_date (tolerant of late/out-of-order events)
- event_date with small correction windows

Bad choice:

- strict event_date partitioning with no correction logic → late data gets misplaced.

*4. Maintain a "Late Events" Store*

At FAANG, this is common:

fact_events_late_arrivals

- event_id
- event_timestamp
- ingestion_timestamp
- reason
- original_partition

This allows downstream logic to:

- reprocess only impacted windows
- avoid corrupting dashboards
- apply corrections safely

*Short Interview Summary (Use This Script)*

> “For streaming models, I keep facts immutable and use SCD2 dimensions with stable surrogate keys. I model event_time and ingestion_time separately to handle late events and maintain temporal correctness.
>
> For schema evolution, I store stable fields as columns and use semi-structured attributes plus versioning for volatile fields.
>
> Pre-aggregated or state tables give fast lookups, while history tables ensure accuracy. This makes the model real-time friendly, evolvable, and resilient to late or out-of-order data.”

# Interview Techniques

**How to Think, Communicate, and Defend Your Model Like a Senior Engineer**

You now understand the fundamentals (Phase 1), the advanced modelling theory (Phase 2), and the architectural patterns used at scale (Phase 3).

Phase 4 focuses on something equally important: how you communicate your reasoning in an interview.

Companies don't only evaluate *what* you model, they evaluate how you think out loud, how you structure your answers, how you justify trade-offs, and how you handle ambiguity under pressure.

This phase teaches the communication playbook senior Data Engineers use in design reviews and modelling interviews:

✔ *Ask clarifying questions that show maturity*

Interviewers judge you more on *what you ask before modelling* than what you draw.
Strong candidates immediately clarify latency, freshness, grain, data sources, consumers, and edge cases because modelling makes sense only when the problem is fully scoped.

✔ *Present your model in a structured, business-aligned way*

A 30-second summary of your design is a required skill at FAANG.
You must explain:

- the chosen grain,
- why you grouped or split dimensions,
- how your model handles history,
- and what trade-offs you made (storage vs accuracy, simplicity vs flexibility, atomic vs aggregated).

This signals leadership-level modelling capability.

✔ *Avoid the classic traps designed to expose shallow understanding*

FAANG interviewers intentionally ask simple-looking problems, order schema, transaction schema, customer model, to see whether you apply:

- SCD correctly,
- point-in-time accuracy,
- identity resolution,
- normalization vs denormalization logic,

- or Data Vault thinking where necessary.

Your job in this phase is to recognize these traps instantly and answer with clarity.

**1. Clarifying Questions to Always Ask**

The most important skill in data modelling interviews before you draw anything

At FAANG-level interviews, your first 60 seconds determine the trajectory of your entire answer. Strong candidates do not start modelling immediately.

They slow down, clarify the scope, and force the interviewer to define the business problem, constraints, and expectations.

Weak candidates start drawing tables.
Strong candidates start drawing boundaries.

This is exactly what senior engineers do in real design reviews.

Below is the ultimate clarifying question playbook, structured to match a real interview flow and focused strictly on modelling, not ETL.

***A. Clarifying Questions Around Business Logic (The Most Important Category)***

These questions show you understand modelling starts with *why*, not *schema*.

*1. "What business metric or decision will this model support?"*

This shows you're designing the model around *use cases*, not guessing.

Examples:

- revenue attribution
- churn forecasting
- real-time recommendations
- order reconciliation
- growth analytics

FAANG loves when you anchor the entire model to business outcomes.

*2. “Who are the consumers?”*

Different consumers = different grains + different modelling choices.

Consumers may be:

- BI analysts
- Product analytics
- ML feature stores
- Real-time API services
- Finance teams
- Experimentation frameworks

Each one imposes different modelling constraints.

*3. “What are the ambiguous or edge-case conditions?”*

This is where senior candidates shine.

Examples you should surface:

- cancelled orders
- returned items
- partial shipments
- cross-device identity
- retries / duplicates in event streams
- missing or null identifiers
- multi-country rules
- GDPR deletes

FAANG intentionally hides edge cases to see if *you* bring them up.

## ***B. Clarifying Questions Around Data Sources & Truth Ownership***

These questions prove you understand modelling ≠ dumping data → it's about source-of-truth clarity.

*1. "What is the system of record for each entity?"*

Example:

- Customer identity → Identity service
- Orders → Order service
- Payments → Billing system
- Events → Clickstream pipeline

This prevents you from mixing mismatched definitions.

*2. "Do different systems define the entity differently?"*

A must-ask for Customer 360, identity resolution, or MDM-style questions.

Interviewer signal:

*You understand cross-system inconsistency is the real modelling challenge at scale.*

## *C. Clarifying Questions Around Grain & Cardinality*

Grain is the most critical modelling decision.

*1. "At what level of detail do we care about the data?"*

Examples:

- event-level
- session-level
- order-line-level
- user-day grain
- daily summaries
- per-product snapshots

This one question alone can save you from 90% of modelling mistakes.

*2. "Should this model support multiple grains?"*

Example:

Atomic clickstream events *and* daily aggregates.

Senior-level signal:
You think in layers, not flat schemas.

## *D. Clarifying Questions Around Latency & Freshness*

These questions show you understand real data systems have timing constraints.

*1. "What is the expected freshness of the data?"*

Common interview answers:

- T+0 near real-time
- T+15 minutes
- T+1 hour
- Daily batch

Different freshness → different models:

- real-time → immutable facts + streaming-friendly SCD
- daily → simpler SCD + aggregates

*2. “Is eventual consistency acceptable?”*

Interviewers expect this question because most distributed systems are eventually consistent.

## *E. Clarifying Questions About History & Updates*

This category exposes whether you understand SCD, point-in-time correctness, and data lineage.

*1. “Do we need historical accuracy or only latest state?”*

This question alone determines:

- Type 1 vs Type 2
- modelling version history
- change tracking
- audit requirements

*2. “How important is point-in-time correctness?”*

If the answer is “very,” then:

- SCD2 required
- effective date ranges
- temporal joins
- backfill-friendly design

*3. “Do updates come as full snapshots or deltas?”*

This determines:

- how you design dimensions
- how you choose update strategies
- how mapping tables behave

This is modelling, not ETL because your schema structure changes based on snapshot vs delta updates.

## *F. Clarifying Questions About Data Quality & Contracts*

These questions show you think like a senior engineer.

*1. "Are there data quality guarantees from upstream?"*

- uniqueness rules
- nullability rules
- schema stability
- identifier stability

*2. "Do we expect schema evolution?"*

If yes → you model flexible/semi-structured attributes.

*3. "What are the constraints around nulls, duplicates, and late events?"*

Constraints define modelling guardrails.

## *G. Clarifying Questions That Instantly Impress FAANG Interviewers*

Ask these and you immediately sound senior:

*1. "What are the primary read patterns and query patterns?"*

Models serve queries, not tables.

*2. "Do downstream users require real-time corrections or backfills?"*

Signals understanding of late-arriving data and reprocessing.

*3. "Do we ever reassign ownership, identity, or relationships?"*

Examples:

- orders reassigned to another store
- subscription migrated to another account
- user merges/splits

This determines if your model needs:

- identity resolution
- versioned relationship tables
- link tables (Data Vault style)

## 2. How to Present Your Model

When you present your model in an interview, the goal is not to walk through every table, it's to communicate your thinking with clarity, structure, and intention. Interviewers evaluate your ability to summarize the design, justify your modelling decisions, and demonstrate that everything you built is tied to the business problem, not guesswork.

Here is the presentation style that consistently performs well:

### *A. Start With a 30-Second Summary*

Begin by giving a crisp, high-level overview of what you designed and why.

A good summary sounds like:

> *"This model captures user actions at the event level, ties them to stable dimensions like customer and device, and preserves full history through SCD2 where needed. The structure supports analytics, experimentation, and ML use cases while staying flexible for schema evolution."*

The interviewer should immediately understand the shape of your solution, even without diagrams.

### *B. Explain Why You Chose This Grain*

After the summary, call out the grain deliberately.

Grain is the centre of every modelling decision, so explain your reasoning in one or two clean lines:

> *"I kept the atomic event grain because the business cares about behavioural analysis, attribution, and session reconstruction. Aggregates can be derived later, but atomic events give maximum flexibility and prevent future metric gaps."*

Or for a case where coarse grain is correct:

> *"Daily grain is sufficient because all KPIs are reported at day-level and there's no requirement for event-level detail."*

This shows you are intentional, not reactive.

### *C. Explain Why You Split or Grouped Dimensions*

Instead of listing tables, articulate the logic behind your structure.

A good explanation sounds like:

> *"I separated customer, device, and identity mappings because each evolves differently and mixing them would create inconsistent history. Product and category live in separate smaller dimensions because their hierarchies change frequently and shouldn't bloat the main dimension."*

Or:

> *"I kept a single dimension because attributes are stable, rarely change, and analysts prefer a one-stop table."*

One or two sentences like this demonstrates architectural awareness.

### *D. Give a Small Trade-Off Analysis*

This is where you sound senior. You show that you considered alternatives and made conscious trade-offs.

It can be as short as:

> *"I'm choosing SCD2 here for historical accuracy, knowing it increases storage, but it preserves point-in-time correctness which this use case depends on."*

Or:

> *"I normalized this dimension to keep it scalable at 100M+ rows, trading off a slightly more complex join for better performance and smaller table growth."*

Or:

> *"I kept this fact narrow instead of adding redundant attributes to keep writes fast, even though it means analysts need to join more dimensions."*

You only need one or two thoughtful lines, not a long analysis to show maturity.

### ***E. Wrap With a Line That Connects Back to the Business***

Interviewers love when a candidate closes with intent:

> *"Overall, this model keeps the business metrics accurate, supports future growth, and stays easy for downstream teams to use."*

This leaves a strong impression that you understand *why* you built what you built.

*Short "Interview Script" Version*

> *"I'll start with a quick summary of the model, then clarify the grain and why it fits the business questions. After that, I'll explain why the dimensions are split the way they are, and point out the main trade-offs I chose. The goal is always a model that's correct, scalable, and aligned to real analytical needs."*

## 3. Common Interview Traps

The subtle modelling pitfalls interviewers use to separate seniors from everyone else

FAANG data modelling interviews are deliberately structured around *trick simplicity*.

The questions look harmless:

*"design an orders schema," "model transactions," "track user updates"*

but each is designed to reveal whether you truly understand modelling fundamentals like SCDs, historical correctness, identity resolution, normalization, and grain clarity.

Below are the traps interviewers commonly set, why they matter, and how to avoid falling into them.

### ***A. The "Order / Transaction Schema" Trap***

What the interviewer shows you:

"Design a schema for orders," or "Model a transaction system."

Simple on the surface but the hidden trap is that they want to see whether you immediately raise the *real* modelling issues:

- Do cancelled orders overwrite or generate new history?

- Do order updates need SCD2 or audit tracking?
- Are refunds separate events or state changes?
- Are line items stored independently or merged into orders?
- Does the business need point-in-time metrics for revenue and attribution?

The mistake many candidates make is jumping straight to a star schema without addressing these subtleties.

*The correct response pattern:*

You acknowledge that orders evolve over time and that your model must capture historical accuracy, not just the current state.

A strong line is:

> *"Orders are not static records, they change state, and attribution depends on when the state changed, so I'd model order headers, order line items, and use SCD2 or effective dating to track state changes instead of overwriting them."*

This instantly signals senior-level thinking.

### B. The "How Do You Handle Updates?" Trap

What interviewers look for:

They're not testing SQL skills, they're checking whether you think in terms of history vs latest state.

Updates in data systems fall into three categories:

1. *New facts* (append-only events)
2. *Attribute changes* (requires SCD strategy)
3. *Snapshots vs deltas* (affects modelling shape)

Most candidates answer at the pipeline level ("use MERGE"), which is a red flag in a modelling interview.

What they actually want:

- Do you choose Type 1 vs Type 2 correctly?
- Do you preserve point-in-time correctness?
- Do you know when updates should generate new rows?
- Do you know when overwriting is acceptable?
- Do you understand the risks of losing historical context?

Your explanation needs only one or two lines, like:

> *"If history matters for metrics or attribution, I model updates as Type 2 with effective dates. If the business only cares about the latest state, I'll use Type 1 to keep the model lean."*

Simple, logical, and shows awareness of real-world implications.

### ***C. The "Large Dimension Problem" Trap***

What interviewers want to test:

Whether you know when to abandon a clean star schema and move toward more scalable patterns like normalization, satellite-style splits, or Data Vault elements.

Large dimensions (customer, product, merchant, device) quickly break naive Kimball models because:

- they change frequently,
- they store hundreds of attributes,
- they contain complex hierarchies,
- and they grow to 50M–200M+ rows.

Most candidates fail because:

They keep one giant dimension and try to cram everything into it.

What a senior-level answer sounds like:

> *"If a dimension gets too large or attributes change at different frequencies, I'll split it into smaller components i.e core identity in one table, volatile attributes in a satellite, and hierarchical attributes in a reference table. This keeps the model scalable and avoids SCD2 explosions."*

This shows you understand scalability, not just schema shapes.

## *D. The "Everything in One Fact" Trap*

Interviewers watch for candidates who push attributes into facts because it feels easier.

This leads to:

- duplication
- fan-out errors
- broken measures
- huge storage overhead
- slow scans
- grain inconsistency

The moment you store descriptive attributes (like user_city or product_category) directly in your fact table, the interviewer knows you're not thinking about *dimensional integrity.*

The subtle requirement is:

- keep facts narrow
- use dimensions for descriptive context
- guard the grain strictly

## *E. The "Ignoring Identity Resolution" Trap*

Any question involving users, customers, accounts, devices, or sessions has an identity trap built in.

Interviewers expect you to ask:

- Are users cross-device?
- Do emails/phones change?
- Do multiple systems disagree on customer_id?

And then model accordingly:

- surrogate keys
- identity bridge tables
- separate tables for device_id, cookie_id, email, phone

This trap tests maturity.
If you don't bring up identity fragmentation, they assume you haven't worked at scale.

## *F. The "Schema Looks Good but Query Patterns Fail" Trap*

A common mistake is designing a pretty schema that collapses under real queries.

Interviewers expect you to check:

- Does the model support the KPIs?
- Are joins ambiguous?
- Is point-in-time correctness possible?
- Does this grain match real questions?

Good candidates *validate their model* before finishing.

A great closing line is:

> *"Before finalizing the model, I want to walk through the core queries to ensure the join paths and grain produce correct results."*

This shows end-to-end modelling maturity.

## *G. Short Summary to Add to Your Guide*

> FAANG interviews are built around subtle modelling traps: orders that evolve over time, updates that require history, large dimensions that can't stay denormalized, identity fragmentation across systems, and schemas that fail under real query patterns.
>
> Strong candidates recognize these patterns immediately and adjust the model based on grain, SCD strategy, identity resolution, and business logic, not surface-level shapes.

# Wrap-Up: Turning Knowledge Into Interview Performance

By this point, you've gone through the full modelling playbook, from fundamentals, to advanced concepts, to architectural patterns, to communication techniques. If this feels like a lot, that's because modelling at FAANG levels is not a memorization exercise, it's a reasoning exercise.

And this is the most important message of this entire guide:

You are **NOT** expected to remember every concept, every trick, every edge case, every SCD nuance, or every architecture pattern.

**No one can.**
**Not even the author.**
**Not even senior engineers in the actual interview panels.**

Interviews are designed to evaluate how you *think,* not how much raw information you can recall under pressure.

What matters is your ability to bring up *a handful of relevant concepts* that show clarity, maturity, and structured thinking.

If you can surface even **5–6 strong points** out of everything in this guide and tie them logically to the case study you're solving, you will already perform better than 95% of candidates.

Because great modelling interviews are not about perfection.

They are about demonstrating:

- that you understand grain,
- that you ask the right clarifying questions,
- that you know when history matters,
- that you can identify tricky edge cases,
- that you can justify your choices,
- and that you think like someone who has built real systems.

The guide I've built is intentionally deep, intentionally comprehensive and not to overwhelm you, but to give you the confidence that whenever you choose *any* point to bring up, it will be the right one.

You don't need all of it.

You just need a **subset that fits the problem in front of you.**

And if you can articulate those few points clearly and calmly, you'll already stand out as someone who understands modelling the way senior Data Engineers do i.e  with clarity, business alignment, and long-term thinking.

This wrap-up should sit at the end of Phase 4 as a grounding reminder:

> *Don’t cram. Don’t force it. Use what fits.*
> *Even a small portion of this knowledge, applied thoughtfully, is enough to nail your interview.*

# Case Studies

## Case Study 1: Design a Data Model for a Product Catalog & Category Hierarchy System (Easy)

**Interviewer:**

Let's start the interview.

We're a large-scale e-commerce marketplace. We need a *Product Catalog & Category Hierarchy data model*.

- Products have multiple variants (SKUs).
- Categories follow a 3-level hierarchy.
- Products can belong to multiple categories.
- Attributes evolve over time.

Please begin.

**Interviewee :**

Thanks. Before I design anything, I want to clarify scope and constraints:

*Business Scope*

1. What are the top business questions this model should answer? (Search ranking? category analytics? inventory? pricing?)
2. Are we modelling for analytics only or also serving systems?

*Scale*

3. How many products and SKUs do we have today?
4. How frequently do attributes change?

*Hierarchy*

5. Is the category hierarchy always 3 levels?
6. Can categories move under new parents over time?

*Mappings*

7. Can a product belong to multiple categories?
8. Can this mapping change historically?

*Variations*

9. Are variants (size/color) modelled as independent SKUs?

*History + SCD*

10. Do we need to reconstruct historical views of products/categories?

*Attributes*

11. Do different categories have totally different attributes?
    (e.g., TV has “screen_size”, Clothing has “fabric”?)

*Users of the Model*

12. Who are consumers - BI, Search, ML, Merchandising?

That’s all I need to select grain + modelling patterns.

**Interviewer:**

Good questions.

- Business focus: search, product detail pages, category analytics, ML features
- Scale: ~3M products, ~10M SKUs
- Attributes change monthly
- Hierarchy is fixed 3 levels (Department → Category → Subcategory)
- Categories can move to different parents
- Products can be in multiple categories
- Variants should be SKUs
- Yes we need full history
- Yes, attributes differ heavily by category
- Consumers: BI + Search + ML

Go ahead and design the model.

**Interviewee :**

Before I go into the actual tables, I want to briefly talk about the *modelling style* I’m choosing here and the trade-offs.

For this use case, I’d go with a *snowflaked dimensional* model rather than:

- a single wide denormalized table, or
- a fully normalized 3NF OLTP-style model, or
- a pure Data Vault implementation.

*Why a snowflaked dimensional model?*

- The workload is analytical: search analytics, category performance, SKU-level reporting, ML features.
- We have clear business entities: Product, SKU, Category, plus mappings.
- Dimensions like category and sometimes brand or attributes naturally form hierarchies / sub-entities, which fit well into a slightly normalized ("snowflaked") dimensional design.

So the core structure is still dimensional (product/sku/category as dimensions), but not a perfectly flat star, it's intentionally snowflaked to keep large, hierarchical dimensions scalable and clean.

*🆚 Why not a single wide table?*

A single flat "product_catalog" table with everything denormalized would:

- Make writes and updates expensive i.e every attribute change forces full-row rewrites.
- Cause SCD2 explosion: category change or attribute updates would duplicate a lot of data.
- Mix different grains (product, SKU, category) in one place, which is dangerous for analytics.

It's attractive for quick querying, but long-term, it becomes hard to maintain and scale, especially with 10M+ SKUs.

*🆚 Why not strict OLTP-style 3NF?*

A fully normalized 3NF design is great for transactional systems, but for analytics:

- You end up with too many joins for every basic query.
- BI and ML teams suffer from complex, fragile query patterns.
- It's not optimized for columnar warehouses and scan-heavy workloads.

So 3NF is overkill for this analytical catalog model.

*🆚 Why not pure Data Vault?*

Data Vault is very powerful for:

- tons of upstream systems,
- constantly changing schemas,
- regulatory/audit-heavy environments.

In this case, we *do* borrow some ideas (like splitting stable vs volatile attributes, bridge-style mapping), but going full Vault would:

- introduce extra complexity (hubs, links, satellites everywhere),
- make it harder for BI/ML teams to query directly,
- be more than we need for a relatively bounded domain (Products + SKUs + Categories).

So I'd rather use dimensional modelling with some Vault-inspired patterns instead of full DV.

*So to sum it up :*

*"For this problem, I'd use a snowflaked dimensional model: Products, SKUs, and Categories as dimensions, plus a bridge for many-to-many mappings.*

*It's simpler and more query-friendly than 3NF, more manageable than a single wide table, and less heavy than full Data Vault, while still giving us history, scalability, and clean hierarchies"*

I'll propose the complete schema now, starting with grain, then entities, then keys.

*A. Grain Selection*

- **Product** grain: one row per *parent* product
- **SKU** grain: one row per *variant*
- **Category** grain: one row per *category node*
- **Mapping** grain: one row per *(product, category)* at a given time

This gives maximum flexibility.

*B. Core Entities + Keys (FULL MODEL)*

### 1. dim_product (SCD2)

Purpose: Product-level identity + brand/title/description
Surrogate Key: product_key

Columns:

- product_key (PK)
- product_id (natural key)
- product_title
- brand_name
- description
- created_date
- effective_start_date

- effective_end_date
- is_current

### 2. dim_sku (SCD2)

Purpose: Variant-level details
Surrogate Key: sku_key
FK: product_key

Columns:

- sku_key (PK)
- sku_id
- product_key
- color
- size
- package_type
- is_active
- effective_start_date
- effective_end_date
- is_current

### 3. dim_category (SCD2)

Purpose: Hierarchical category structure
Surrogate Key: category_key

Columns:

- category_key
- category_id
- category_name
- category_level (1/2/3)
- parent_category_key
- effective_start_date
- effective_end_date
- is_current

### 4. bridge_product_category (SCD2)

Purpose: Handle many-to-many mapping
Grain: one row per (product_key, category_key)

Columns:

- product_key
- category_key
- primary_flag
- effective_start_date
- effective_end_date
- is_current

### 5. dim_sku_attributes (Semi-Structured)

Purpose: Flexible attributes per SKU (JSON or name-value)

Columns:

- sku_key
- attributes_json
- schema_version
- ingestion_timestamp

### 6. fact_product_snapshot_daily (Optional, Derived)

Grain: (date_key, sku_key)

Columns:

- date_key
- sku_key
- product_key
- primary_category_key
- current_price
- inventory_status

**Interviewer:**

Why did you choose SCD2 for product, SKU, and category?
Why not Type 1 overwrites?

**Interviewee :**

Because all three have business requirements tied to historical correctness.

- Product details may change (e.g., title update)
- Categories may reorganize
- SKUs may get rebranded or repackaged

If we overwrite, dashboards and ML training sets become inconsistent.

SCD2 ensures:

> *"What was true at the time remains true when we query historical data."*

**Interviewer:**

Why not flatten product + SKU into one wide table?

**Interviewee :**

I avoided flattening for three reasons:

1. Identity vs Variant separation
   Product is brand-level identity; SKUs are sellable units with their own lifecycle.

2. SCD churn
   SKU attributes change more frequently.
   If flattened, product rows would explode unnecessarily.

3. Data quality
   Dimensions stay clean, maintainable, and stable.

**Interviewer:**

Categories change quarterly, how does your model adapt?

**Interviewee :**

Because dim_category is SCD2, hierarchy changes simply create a new versioned row.

bridge_product_category is also SCD2, so product → category membership history is preserved.

This gives perfect point-in-time reconstruction.

**Interviewer:**

Suppose tomorrow the merchandising team adds 20 new attributes to Clothing (e.g., collar_type, fabric_gsm, pattern). What changes in your model?

**Interviewee :**

Nothing in the core schema.

Those attributes go into:

- dim_sku_attributes.attributes_json

This avoids schema churn and keeps the model maintainable.

If any of those attributes become widely used across multiple categories, we can “promote” them to structured columns later.

**Interviewer:**

Let’s say we scale to 50 million SKUs, and attributes update weekly.
Your SCD2 tables may explode.

How do you prevent bloat?

**Interviewee :**

I’d split product into:

- dim_product_core → stable identity
- dim_product_attributes → SCD2 for volatile attributes

Same for SKUs.

This isolates churn and keeps dimension sizes manageable.This is essentially a Data Vault satellite pattern, embedded inside a dimensional model.

**Interviewer:**

Good. Please summarize your final model.

**Interviewee :**

The final Product Catalog model consists of:

- dim_product (SCD2) - stable product identity
- dim_sku (SCD2) - sellable variants
- dim_category (SCD2) - 3-level hierarchy, historical
- bridge_product_category (SCD2) - many-to-many mapping with history
- dim_sku_attributes (Variant/JSON) - flexible category-dependent attributes
- fact_product_snapshot_daily - optional historical snapshot

This model supports search, browsing, ML features, and analytics while being scalable and history-safe.

**Interviewer:**

Excellent. That’s exactly how a senior engineer should structure this problem.

## Case Study 2 - Design a Clickstream & Session Modelling for a Video Streaming Platform (Easy)

**Interviewer:**

Let's move to the next case.

We are a large-scale video streaming platform (think Netflix or YouTube).
We want to build a data model for user engagement analytics based on clickstream events: play, pause, seek, stop, impressions, etc.

Source data is a continuous stream of JSON events from web, mobile, and smart TV clients.

**Task:**

Design the core analytical data model for our warehouse to support user engagement analytics and downstream ML use cases.

Walk me through your thought process.

**Interviewee:**

Great, this is a classic but very real-world problem.

Before proposing any schema, I'd like to ask a few clarifying questions so I don't optimize for the wrong thing.

Clarifying Questions

1. *Business Goals:* What are the top 3–5 questions this model must answer?

   For example:

   - total watch time per user/video/day,
   - completion rate,
   - drop-off points in a video,
   - engagement by device type?

2. *Scale*: Rough order of magnitude:

   - how many daily active users,
   - and how many events per day?

3. *Latency / Freshness :*

   Are we okay with T+1 daily analytics, or do we also need near-real-time (say, under 15 minutes) views?

4. *Consumers*

   Who will use this model? BI analysts (Tableau/Looker), Data Scientists, or Product teams needing ad-hoc exploration?

5. *Event Semantics*

   Do we have a unified event schema (play, pause, seek, impression, etc.), or do different clients/platforms send slightly different JSON payloads?

6. *Sessions*

   Do we have a defined notion of "session" already (e.g., 30 minutes of inactivity = new session), or is that something we derive downstream?

7. *Identity*

   What identifiers are available?

   - user_id,
   - device_id,
   - anonymous_id / cookie_id?

   And do we care about anonymous users?

8. *History & Retention*

   How long do we store raw events? 1 year, 3 years?

These will drive the grain and table structure.

**Interviewer:**

Good questions. Here's more context:

- Business goals:
    - watch time per user/video/day,
    - completion rate,
    - drop-off position in videos,
    - engagement by device and platform,
    - basic funnel like "impression → click → play".
- Scale: About 10 billion events per day from ~500 million daily active users.
- Freshness: Dashboards can be T+1, but product teams want an hourly near-real-time view as well.
- Consumers: All of them: BI, DS, and PMs.
- Events: Mostly unified schema, but mobile sometimes adds fields earlier than web (so mild schema drift).
- Sessions: We don't have a hard definition yet; your model can define it (something reasonable like 30 min inactivity).
- Identity: We have user_id for logged-in users and device_id for anonymous activity.
- Retention: Keep raw events for at least 2 years.

Go ahead.

**Interviewee:**

Before going into actual tables, I'll quickly explain the type of model I'll use and why.

For this use case, I'd choose a dimensional model with a ***star schema*** at the core:

- A central event fact table at atomic grain
- with several supporting dimensions: user, video, device, date, time, location, and optionally session.

It will be lightly "snowflaked" where it makes sense (e.g. separate location or platform dims), but conceptually still a star.

*Why not a single wide table?*

A giant denormalized clickstream table would:

- duplicate video/user metadata billions of times,
- make SCD handling messy,
- and be painful to maintain as attributes evolve.

*Why not strict 3NF?*

We're in an analytical warehouse, not an OLTP system.
3NF would introduce too many small tables and joins for every BI query.

*Why not pure Data Vault?*

We *could* model events in Vault (hubs/links/satellites), but for this level and the need for BI/DS teams to query quickly, a dimensional model is more practical.
We can still borrow ideas like separating stable vs volatile attributes if needed.

So I'll proceed with a star schema centered on a **fact_event** table, plus a derived session fact for convenience.

***Step 1: Business Process & Grain***

The core business process here is:

> *"User interactions with video content over time."*

I'll define the *primary grain* of the main fact table as:

> *One row per user event (a single action: play, pause, seek, stop, impression, like, share).*

This gives maximum flexibility for both analytics and ML feature engineering.

On top of that, I'll derive a *session-level fact table later* (one row per user session per device).

***Step 2: Fact Table Design***

**fact_video_event**

Grain: one row = one event (play/pause/seek/etc.)

Key columns:

- event_id (surrogate PK, unique per event)
- user_key (FK → dim_user; nullable for anonymous)
- video_key (FK → dim_video)
- device_key (FK → dim_device)
- session_key (FK → dim_session; nullable initially, can be derived)
- date_key (FK → dim_date) – from event_timestamp
- time_key (FK → dim_time)
- location_key (FK → dim_location, if available)

*Event attributes (measures/flags):*

- event_type (play, pause, seek, stop, impression, like, share, etc.)
- event_timestamp (actual time of event)
- ingestion_timestamp (when we received it – for late data handling)
- playhead_position_sec (position in video)
- watch_duration_sec (for play/stop segments)
- is_autoplay, is_muted, is_fullscreen (optional flags)

This table stays **append-only**; we don't update events in-place.

***Step 3: Dimension Tables***

Now I'll walk through the key dimensions and their roles.

**dim_user (SCD2)**

- user_key (PK)
- user_id (natural key)
- account_created_date
- subscription_tier (Free, Basic, Premium, etc.)
- country, language
- effective_start_date, effective_end_date, is_current

We use SCD2 because subscription tier and region may change, and we want historical analysis like:

*“How did engagement change after upgrade to premium?”*

### dim_video (SCD2)

- video_key (PK)
- video_id
- title
- genre
- duration_sec
- content_partner
- age_rating
- effective_start_date,
- effective_end_date,
- is_current

SCD2 lets us analyze engagement vs metadata changes (e.g., different thumbnails, categories).

### dim_device (Type 1 or light SCD2)

- device_key
- device_type (mobile, web, TV)
- os (iOS, Android, etc.)
- app_version

Usually Type 1 is enough; we don’t always need historical device attribute changes for analytics.

### dim_session (Derived)

Sessions are derived from event streams with rules like “30 min inactivity → new session”.

- session_key (PK)
- session_id (if we generate or infer it)
- user_key
- device_key
- session_start_ts
- session_end_ts
- total_events
- total_watch_time_sec

This acts as a convenience dimension/fact hybrid for session-level reporting.

**dim_date & dim_time**

Standard slowly changing date/time dimensions for efficient partitioning and time rollups.

**dim_location (Optional)**

- location_key
- country
- region
- city

If we can geo-resolve IPs or have location metadata.

**Interviewer:**

This is a solid design. Now, why did you choose an event-level fact instead of an aggregated daily fact?

**Interviewee:**

Good question.

I chose event-level grain because:

- We need to analyze drop-off positions inside a video (e.g., people leaving at 20% vs 80% progress). That needs playhead-level detail.
- Data Scientists want raw sequences of events to build sequence models and features.
- Aggregates like daily watch time can always be computed from atomic events, but the reverse is not true.

To keep performance manageable, I'd complement this with pre-aggregated tables later (e.g., fact_video_user_day), but the source of truth stays at event grain.

**Interviewer:**

Why a star schema with dimensions instead of one giant flat events table with user and video info repeated?

**Interviewee:**

A flat table would be simpler to query initially, but:

- It would repeat user metadata and video metadata billions of times, increasing storage and scan cost.
- Handling SCD for user/video would become messy – we'd have to duplicate dimension logic in the fact.
- Any metadata correction (e.g., fixing genre) would require rewriting huge chunks of data.

With a star schema:

- fact_video_event is lean and purely behavioral.
- dim_user and dim_video handle history cleanly using SCD2.
- Joins are straightforward and columnar warehouses are optimized for this pattern.

**Interviewer:**

At 10B events/day, how would you physically model this fact table in a warehouse like BigQuery or Snowflake from a modelling perspective?
Partitioning, clustering?

**Interviewee:**

From a modelling perspective, I'd design:

- Partitioning by event_date (derived from event_timestamp).
  This keeps daily chunks manageable and aligns with most reporting needs.

- Clustering (or sort keys) by something like:

    - video_key, then
    - user_key,

      so queries filtered by video or user get good pruning.

In many warehouses, I'd also keep both event_timestamp and ingestion_timestamp so we can reason about late-arriving data.

**Interviewer:**

Suppose European events are delayed by 48 hours due to some network issue.
We have dashboards partitioned by event_date. How does your model cope with late events?

**Interviewee:**

Because the fact model has separate timestamps, we can handle this cleanly:

- event_timestamp → the actual user action time
- ingestion_timestamp → when the warehouse received the event

Partitioning is by event_date (based on event_timestamp), so late events are still written into the correct date partition, even if they arrive 2 days later.

From a reporting perspective, we usually define a “watermark” or SLA window (e.g., T+2 days) before considering metrics final, and can run backfill jobs for aggregates over the last few days.

The modelling choice (storing both timestamps) is what enables that correctness.

**Interviewer:**

Mobile adds a new field is_background_play which the web doesn’t send. How does your model adapt?

**Interviewee:**

This is where the event model’s flexibility matters.

Two options:

1. Add is_background_play as a nullable column to fact_video_event.
    - For platforms that don’t send it, it stays null.
    - This is fine if we don’t have dozens of such fields.
2. If many such platform-specific fields appear, we can add a semi-structured column like event_properties (JSON) storing extra attributes.

I’d start with option 1 for a few fields, and move to a JSON-based event_properties if drift becomes heavy.

The core grain and table structure remain unchanged.

**Interviewer:**

Let’s wrap up. Summarize your final data model.

**Interviewee:**

At the end of this design, the core analytical model looks like this:

- fact_video_event – event-level fact, one row per user event (play/pause/seek/etc.), with FKs to user, video, device, date, time, location, and optionally session.
- dim_user (SCD2) – user identity and subscription tier history.
- dim_video (SCD2) – video metadata history (genre, duration, content partner).
- dim_device – device/app context (platform, app version).
- dim_session – derived session table aggregating events into sessions.
- dim_date, dim_time, dim_location – standard analytical dimensions.

This star schema:

- supports detailed engagement analytics and ML,
- is scalable for 10B events/day,
- handles schema drift and late data via timestamps and flexible attributes,
- and keeps user/video history clean via SCD2.

## Case Study 3 - Customer 360 Identity Resolution (Medium Difficulty)

**Interviewer:**

Let's go to the next scenario.

We are a consumer-facing platform (think Amazon, Uber, Spotify).
We want a *Customer 360 model that unifies user identities across multiple systems*:

- Marketing system → email, device_id, cookie_id
- Transaction system → user_id
- Mobile app → device_id
- Web analytics → cookie_id
- CRM → email + phone
- Support tickets → email or phone or user_id

Users can log in on some devices, stay anonymous on others, and change emails/phones over time.

Your task:

Design a core analytical Customer 360 model that resolves identities correctly and supports analytics & ML.

**Interviewee:**

Before designing the model, I want to clarify a few things so I choose the right grain and identity strategy.

*Business Goals*

1. What are the main analytics + ML use cases?
   (Churn prediction? LTV modelling? Engagement tracking?)

*Identifiers*

2. Which identifiers are considered authoritative?
   user_id? email? phone?
   Or none?

3. Can users:

    - change email?
    - change phone?
    - log in with social accounts (Google/Apple)?

*Anonymous Activity*

4. Should anonymous activity (device_id, cookie_id) be linked to known users once they log in?
   e.g., user browses anonymously → then logs in → should we unify that history?

*Cross-System Mapping*

5. Are identifiers consistent across systems, or do we see different formats?
   (e.g., marketing emails vs CRM emails)

*Scale*

6. What's the scale?

    - number of users
    - number of devices per user
    - How many identifiers are typically linked?

*Historical Needs*

7. Do we need to preserve identity history?

   Example:

    - old email vs new email
    - merging multiple accounts
    - splitting identities that were incorrectly merged

*Data Quality*

8. How common are duplicates?
   (Same email with different user IDs? same phone shared by multiple accounts?)

Consumers

9. Who uses this? BI? Marketing? ML? Fraud detection?

*Event Attribution*

10. Should events always use the latest identity or the identity at event time?
    (This affects point-in-time correctness.)

That gives me the constraints needed to design the Hub/Golden Record.

**Interviewer:**

Good questions.

- Business goals: churn prediction, LTV, unified engagement metrics, attribution
- Authoritative ID: user_id from our auth system
- Email/phone can change and we need to preserve history
- Anonymous → known linking: yes, when a user logs in, all past anonymous device_id/cookie_id activity should attach to them
- Scale: ~150M users, ~400M devices, ~3–5 identifiers per user
- Historical needs: yes, full history required
- Duplicates: happens frequently (multiple emails, merged accounts)
- Consumers: ML, marketing, BI, and fraud
- Event attribution: must support both
    - identity at event time
    - current unified identity for lifetime metrics

Go ahead.

**Interviewee:**

Before detailing tables, I want to explain *how* I'll model this.

I'll use a *hybrid model:*

✔️ Dimensional backbone (dim_user, mapping tables)
✔️ Data-Vault-inspired identity resolution layer (Hubs/Links/Satellites)

Why?

*Why not a plain star schema?*

Because identity is messy and star schemas are bad at many-to-many identifier mapping and historical linking.

*Why not full Data Vault?*

DV is great for ingestion, but BI/ML teams shouldn't query raw hubs/links, too many joins.

**So hybrid = best tradeoff**

- Identity resolution in link tables
- User profile in SCD2 dim_user
- Clean "golden record" for analytics
- Preserves history, supports merging, supports anonymous → known conversion

I'll now walk through the full model.

## *A. Core Golden Record*

**dim_user** (SCD2 - Golden User Record)

Surrogate key: user_key

Columns:

- user_key (PK)
- user_id (auth-system ID)
- canonical_email
- canonical_phone
- account_create_ts
- marketing_opt_in_flag
- status (active, suspended, deleted)

- SCD2 fields:
    - effective_start_date,
    - effective_end_date,
    - is_current

This is the customer 360 profile.

***B. Identity Mapping Layer***

This is where the real identity resolution happens.

**link_user_identifier** (Many-to-many mapping, SCD2)

Grain: one row per (user_key, identifier)

Identifiers include:

- email
- phone
- device_id
- cookie_id
- external login ids (Google/Apple)

Columns:

- user_key (FK → dim_user)
- identifier_type (email/phone/device_id/cookie_id)
- identifier_value
- source_system (CRM, marketing, mobile app, web analytics)
- is_verified_flag
- effective_start_date
- effective_end_date
- is_current

This lets 1 user → many identifiers
and identifiers → can map to different users over time.

***C. Device Modeling***

**dim_device**

- device_key (PK)
- device_id
- device_type
- os
- app_version

Device_ids link to users through link_user_identifier.

***D. Event Table (to show usage)***

**fact_user_event**

Not the primary design focus, but needed to show usage.

- event_id
- user_key (FK) – identity at event time
- device_key
- event_timestamp
- event_type
- payload

**Interviewer:**

How do you handle anonymous browsing?
Someone uses the app with device_id, then logs in 2 days later with user_id.

**Interviewee:**

This is where the identity link table shines.

When anonymous events come in:

- device_id gets inserted as an identifier in link_user_identifier but with no user mapped yet
  → we store them as *unassigned identifiers*

When the user logs in:

- We merge all past identifiers (device_id, cookie_id) onto user_key
- We version out old mappings using SCD2
- All future events use the new unified identity
- Past events remain historically correct (device-only events still point to that device)

If we need retroactive identity:

- we can run a reconciliation job to remap old events to the user_key.

This is exactly how companies handle anonymous → known identity stitching.

**Interviewer:**

What if we discover that two user_ids actually belong to the same person?
E.g., user created 2 accounts with different emails.

**Interviewee:**

We do an identity merge, similar to MDM merge operations.

Mechanism:

- Decide which user_id is the “survivor”
- In dim_user:
    - Close old SCD2 record
    - Move identifiers to the surviving user_key
- In link_user_identifier:
    - Reassign identifiers
    - Maintain historical versions
- Fact tables:
    - We keep event data tied to the original identity for PIT correctness
    - For lifetime metrics, we can aggregate by the new unified user_key
    - This dual-view is a conscious modelling choice

This lets BI and ML teams choose:

- historical identity or
- current unified identity

**Interviewer:**

Users can change email and phone numbers.
How do you capture history for both?

**Interviewee:**

Email/phone history belongs in both:

dim_user (canonical version)

- stores the current best version of email/phone
- but not all historical versions (too noisy)

link_user_identifier (full history, SCD2)

- stores every email/phone used by the user
- stores verification status, source system, and validity dates
- allows historical ML analysis like

*"Did using a certain email domain correlate with churn?"*

- preserves identity merges/splits cleanly

This separation avoids polluting dim_user while keeping all needed history.

**Interviewer:**

Suppose we ingest data from marketing and CRM.
Marketing uses lowercase emails; CRM uses uppercase and often strips dots.
How do you handle inconsistent formats?

**Interviewee:**

This is an identity resolution classic.

I do:

*Normalization Layer*

- lowercase
- strip spaces
- collapse dots for Gmail
- remove country codes for phone

*Hashing*

Store a fingerprint hash of each identifier to simplify joins.

*Confidence Scoring*

When linking identifiers to users, we maintain a **confidence score**:

- strong match → verified email login
- weak match → same phone but unverified

This prevents accidental merges.

The model stays the same processing logic handles normalization.

**Interviewer:**

What if the same phone number is used by multiple users (family accounts)?
How does your model avoid merging them incorrectly?

**Interviewee:**

This is why the link table has:

- identifier_type
- identifier_value
- is_verified_flag
- source_system

If multiple user_keys map to the same phone:

- We keep them separate unless we have a verified login linking them
- The link table's SCD2 + source metadata ensures we never blindly merge

This is exactly how robust identity systems work in practice.

**Interviewer:**

Summarize your final Customer 360 model.

**Interviewee:**

The Customer 360 model consists of:

*1. dim_user (SCD2)*

- stable golden customer record
- account metadata, subscription status, country, etc.

*2. link_user_identifier (SCD2)*

- maps users to all identifiers (email, phone, device_id, cookie_id)
- supports history, merges, anonymous → known transitions
- the backbone of identity resolution

*3. dim_device*

- additional device attributes for analytics/ML

*4. fact_user_event*

- shows how identity is used downstream
- supports both historical and unified identity attribution

This hybrid dimensional + identity-mapping model:

- handles messy real-world identity
- supports cross-device attribution
- preserves history
- scales to hundreds of millions of users
- and enables ML + BI + marketing use cases cleanly

# Case Study 4 - Real-Time Ride / Trip Modelling (Hard Difficulty)

**(Uber / Lyft / DoorDash–style system)**

**Interviewer:**

Alright, let's do a data modelling case study

We operate a *global ride-hailing network.* Trips are event-driven. Nothing is clean.
Clients send partial updates, out-of-order messages, retries, device pings, driver reassignments, destination edits, surge adjustments, cancellations, the whole mess.

Your task: *Design a data model that represents a trip accurately.*

I'll stop you frequently.
Start however you want.

**Interviewee:**

Okay… in that case, I'll start with the thing that's unclear to me:

Do you produce a "trip object" in the backend, or is the trip purely reconstructed from events?

Because that changes the entire modelling approach.

**Interviewer:**

Good question.
We don't emit a clean trip object.
Everything comes in as events from clients and services.
Your model must reconstruct truth.

And no, events aren't guaranteed to be ordered.
Clock skew is real.
Mobile clients batch events under poor network conditions.

Please proceed.

**Interviewee:**

Alright, so I have to assume "event-first" modelling.

Let me think aloud for a moment.

If the truth is a sequence of events but consumers want final trip-level analytics then the model has to do both:

1. Retain the raw, messy sequence

2. Derive a coherent, stateful representation

So my instinct is:

- Model events as the ground truth
- Reconstruct a versioned “trip state”
- Expose a final trip entity for analytics

But let me not jump too far.
First, I need more clarity around drivers and rider identities.

Can a trip have multiple drivers over time?

**Interviewer:**

Yes.
Driver A gets assigned, cancels mid-way, driver B takes over.
It happens thousands of times a day.

And you don’t get a “clean breakpoint.”
We infer transitions from events.
Sometimes incorrectly.

**Interviewee:**

Alright, so this isn’t even a clean many-to-one mapping.
It’s many-to-many-with-temporal-validity.

So I need to track:

- trip_id
- driver_id
- effective time window

Otherwise historical queries break.

Got it.

Another question:
Do you need point-in-time correctness?
For example: If surge changed mid-trip, must we reconstruct the fare based on surge “as of that moment,” not the final surge?

**Interviewer:**

Absolutely.
This model supports pricing experiments and fraud detection.

Point-in-time correctness is mandatory.

**Interviewee:**

Okay.
If point-in-time correctness matters, I cannot store one “trip row” that overwrites itself in-place.

I need a versioned representation of the trip state.

But before I propose tables, let me ask something subtle:

Is cancellation a terminal state or a branching state?
Meaning:

- When a rider cancels before assignment - that’s one scenario.
- When the driver cancels after arriving - different scenario.
- When the system auto-cancels due to timeout - different again.

Do all these need independent modelling?

**Interviewer :**

Yes.
And we want to understand *at which stage* cancellation happened.
It determines:

- refund
- driver compensation
- fraud heuristics
- supply/demand balancing

So yes, cancellation type + timing must be represented.

**Interviewee:**

Okay, so we have:

- A messy, unordered stream of events
- Multiple actors (rider, driver, backend)
- Temporal lapses

- Non-terminal “branching” cancellation paths
- Dynamic surge
- Device pings
- Driver reassignments
- Destination edits
- And we must reconstruct truth

Let me now commit to my modelling spine.
I’ll explain my design choices as I go

I’m going to treat the trip like a time-evolving entity, not a static row.

My model will have three core layers:

***1. Raw Events (truth layer)***

Every action - good, bad, duplicate, out of order - goes here.
This table is immutable.
It is the canonical truth of “what we received.”

***2. Trip State History (derived but versioned)***

This is a reconstruction of what the trip looked like *after each meaningful event*.
Think of it like an SCD2 record but driven by event timestamps.

***3. Final Trip Snapshot (cleaned analytics surface)***

This is what BI wants: one row per trip.
But it is derived, not directly sourced from upstream.

This tri-layer architecture reflects how companies like Uber engineer their lakehouse.

Does that sound aligned so far?

**Interviewer :**

Yes, so far you’re on track.
But I want details now.
Let me challenge you.

How do you reconstruct the state if events arrive out of order?

**Interviewee:**

I’ll rely strictly on event_timestamp, not ingestion_timestamp, when reconstructing the state.

But ingestion timestamp is still needed to detect anomalies.

So the reconstruction algorithm sorts events by event_timestamp and applies the transitions sequentially.

If I receive:

- pickup at 10:31
- then driver_arrived at 10:29

The algorithm reorders them.
But I keep ingestion order too, so I can flag:

- late arrival
- duplication
- missing events
- possible device batch uploads
- fraud signals

The model stores both timestamps.

**Interviewer :**

Good,now give me the actual schema like tables. columns. Keys and other details.
And don't just list them but explain *why* each exists.

**Interviewee:**

***1. Raw Events Table***

**trip_event**

Every message from rider/driver/backend lands here. This is our source of truth.

It includes:

- event_id - unique surrogate
- trip_id - assigned when rider requests
- rider_id
- driver_id (may be null early on)
- event_type - request, driver_assigned, driver_arrived, pickup, etc.
- event_timestamp - logical timestamp
- ingestion_timestamp - when we saw it

- payload - full JSON, unflattened, containing GPS, surge, route, etc.

Why store payload?
Because new pricing algorithms may want fields we didn't previously model.

This table is immutable and append-only.

***2. Trip State History***

**trip_state_history**

This is where the trip's lifecycle is reconstructed.

Each row is a snapshot of the trip after an event. Not every event produces a snapshot, only meaningful transitions.

Columns include:

- trip_state_id
- trip_id
- rider_id
- driver_id
- trip_status - requested, driver_assigned, en_route, pickup, dropoff, canceled_xxx
- surge_multiplier
- estimated_fare
- estimated_pickup_time
- route_version
- effective_start_ts
- effective_end_ts
- is_current

This is essentially SCD2 driven by event timestamps.

Why?
Because pricing and fraud rely on time evolution, not final states.

***3. Final Trip Snapshot***

**dim_trip**

This is what BI queries:

- trip_id
- rider_key
- driver_key
- pickup_ts
- dropoff_ts
- wait_time_sec
- travel_time_sec
- total_distance
- final_fare
- surge_multiplier_at_pickup
- cancellation_status
- cancellation_stage

This row is computed from trip_state_history once the trip is terminated (completed or canceled).

***4. Driver Assignment History***

**trip_driver_assignment**

Because a trip can have many drivers:

- trip_id
- driver_id
- assigned_at
- unassigned_at

Without this, driver attribution breaks.

***5. GPS Ping Table***

**trip_gps_point**

A trip may have thousands of GPS emissions. Storing them inside events bloats the pipeline.

So I keep a separate table:

- trip_id
- driver_id
- timestamp
- lat, lon, speed, bearing

Used for route reconstruction, fraud detection, and ETA simulation.

**Interviewer :**

Let's see if your design holds up under pressure.

Scenario 1:

A trip receives pickup and dropoff events, but never sends driver_arrived.How does your model handle missing transitions?

**Interviewee:**

Great question.

Trip_state_history doesn't require all transitions.
It models *what happened*, not *what should have happened*.

If a driver_arrived event is absent:

- No row is created corresponding to that state
- The state transitions jump directly from "driver_en_route" to "pickup"

This actually becomes a data-quality signal.
In trip_state_history, the missing transitions become detectable patterns.

I do not backfill or infer missing events, I surface them.

**Interviewer :**

Scenario 2:

A driver emits duplicate events, two pickups at the same timestamp.

How do you detect and handle duplicates?

**Interviewee:**

Duplicate detection happens at two layers:

Layer 1 - Raw Events:

I keep all the duplicates. Without altering raw truth ..

Layer 2 - State Reconstruction:

When reconstructing state, I examine:

- same trip_id
- same event_type
- same event_timestamp
- identical or near-identical payload

If duplicates are detected:

- They do NOT create new state versions
- They increment a duplicate_count field in the state snapshot

This allows detection without polluting the state timeline.

**Interviewer :**

Scenario 3:

Rider changes the destination mid-trip - twice. Surge also changes twice during the trip. Driver takes a detour.

Where do these changes live? How do you model them?

**Interviewee:**

Destination change → event
Surge change → event
Route change → event
Detour → emerges from GPS pings, not explicit events

In trip_state_history, every time any of these changes occur:

- a new versioned state row is produced

- effective_start_ts = change timestamp
- previous state gets effective_end_ts
- surge multiplier, estimated fare, route_version all update accordingly

So the timeline might look like:

- state 1: initial destination + surge x1
- state 2: destination changed + surge x1
- state 3: destination same + surge x1.5
- state 4: destination same + surge x2.0
- state 5: pickup
- state 6: mid-route state
- state 7: detour (route_version incremented from GPS)
- state 8: dropoff

The model doesn't collapse these, it preserves every change as part of the timeline.

**Interviewer :**

Scenario 4:

Device pings come in 3 minutes late during poor network. But ETA models rely heavily on near-real-time GPS.

Does your model break?

**Interviewee:**

No, because GPS is modeled separately in trip_gps_point.

There I store:

- event_timestamp
- ingestion_timestamp

ETA models can discard stale pings by comparing the two.

Trip_state_history only references the current or last-known route, so delays in GPS don't corrupt the state timeline.

In other words, GPS affects the trip indirectly, not by altering the main event flow.

**Interviewer :**

Surge pricing changed at 10:01, 10:05, 10:10, and 10:14.
The trip started at 10:00.
Pickup happened at 10:13.
Dropoff at 10:20.

What surge multiplier do you record as the “surge at pickup”?

**Interviewee:**

I record the surge multiplier from the trip_state_history row whose effective_start_ts is the latest value before or equal to 10:13 (pickup_ts).

Meaning:

- If surge changed at 10:10 → that’s the active multiplier
- The 10:14 update doesn’t apply yet

This is exactly why state versioning exists.

**Interviewer :**

Good.
Summarize your model in one minute - as if explaining to a VP.

**Interviewee:**

A trip isn’t a row - it’s a timeline.

Real systems don’t give us perfect events, so the model must preserve:

- every event as emitted
- a time-evolving sequence of states
- a final cleaned trip suitable for analytics
- all supporting components like driver assignments and GPS pings

So the architecture is:

1. Raw events - the messy, immutable truth.
2. Trip state history - a versioned reconstruction of the lifecycle, aligned by event timestamps.
3. Final trip snapshot - a clean, BI-friendly surface.
4. Supporting tables - driver history, GPS, location semantics.

This supports pricing, fraud, ETA, routing, experimentation, real-time ops, and historical analysis, even when events are late, duplicated, out-of-order, or contradictory.