CHAPTER 03

Synthetic data, ETL from scratch, and how ETL meets TDM

The fourth technique, making test data from nothing, then extract-transform-load step by step, and how ETL and TDM overlap, differ and work together.

Test data classification and synthetic data generation

Two ways to get test data are to copy and cut down real data (the discovery and subsetting concepts chapter) and then hide the sensitive parts (the masking concepts chapter). The third way is to **make the data up**, on purpose, from rules. Before choosing a way, the better question is not "where do I get data?" but *what is this data for?*

A tester checking the login screen does not need "a lot of customers". She needs one normal customer, one customer whose account is locked, one with a very long surname, one with no phone number, and one whose date of birth makes them exactly 18 today. Each of those rows has a different job. Naming the jobs is called **classifying** test data, and it is the starting point for designing any data set, whether you copy it, mask it or generate it.

The chapter also covers the "Model, Design, Deploy, Manage" cycle and places the five data patterns (boundary conditions, sequential, random, edge cases, null) on a map. Tool detail is in the Informatica data generation chapter and the GenRocket chapters.

3.1 Words to know first

Test data

Any data the application is fed or finds waiting in its database during a test. It includes both the rows in the database and the values typed into screens or sent in files.

Test condition

One specific situation a test wants to check, such as "customer with no phone number" or "order with the maximum number of items". Each condition needs data that creates it.

Positive (valid) data

Data the application should accept and process normally.

Negative (invalid) data

Data the application should reject or handle gracefully, such as letters in a phone number.

Boundary value

A value sitting exactly on the edge of what is allowed. If customers must be 18 or older, the boundary values are 17, 18 and 19 years.

Edge case

A rare or unusual situation outside normal use, such as a surname containing an apostrophe. GenRocket describes it as a rare or unexpected scenario outside the typical usage pattern.

Synthetic data

Data made by a program following rules, instead of being taken from real events. GenRocket's help centre puts it as data manufactured by an algorithm rather than real-world events.

Generator

The small piece of logic that produces the values for one column, for example "count up from 1" or "pick a name from a list".

Seed

A starting number given to a random generator. The same seed gives the same "random" values every time, so a test data set can be rebuilt exactly.

Distribution

How values are spread: for example, 20% of customers have no orders, 40% have one. Real data is lumpy, and good test data copies the lumps that matter.

Referential integrity

The promise that every foreign key points at a row that exists. An order whose `customer_id` has no matching customer breaks it. See the data basics chapter.

Parent table / child table

The table being pointed at (CUSTOMERS) is the parent; the table holding the foreign key (ORDERS) is the child.

Coverage matrix

A grid that lists test conditions against the data that covers them, so you can see gaps at a glance.

3.2 Why classify test data at all?

Imagine a kitchen preparing for a cooking class. The chef does not just buy "food". She buys the exact ingredients each recipe needs, in the amounts each recipe needs, and she keeps the allergy-sensitive items (nuts) in a separate, labelled place. Test data works the same way. You prepare the exact rows each test needs, and you keep the dangerous ones (real personal data) away from places they must not be.

The analogy stops being accurate in one place: a chef can taste the soup, but nobody can "see" whether a table of 5 million rows has the right mix. That is why we write the classification down, as a plan, before building anything.

WHY IT MATTERS

Without classification, teams fall into two traps. Either they copy everything from production (huge, risky, and still missing the odd cases that have never happened yet), or they hand-type five "happy path" rows and discover the bugs in production. Classification gives you a short list of questions to ask for any data request: *where does it come from, how sensitive is it, and what must it prove?*

3.3 Three ways to classify test data

You can sort the same piece of data along three independent lines. A single customer row can be "masked production" by source, "restricted until masked" by sensitivity, and "boundary" by purpose. Keep the three lines separate in your head and in your documents.

By source	+	By sensitivity	+	By purpose

where did it come from?

how risky is it if it leaks?

what must the test prove?

3.3.1 By source

The source is how the data came to exist. Tool capabilities divide along this line: subsetting and masking work on production-derived data, while generation makes new data.

Source class	How it is made	Typical risk	Where it is covered
Production copy (raw)	Straight copy of the real database	Exposes real personal data; not allowed in test for regulated data	Why TDM exists
Production subset	A cut-down, consistent slice of production	Still real data until masked	the discovery and subsetting concepts chapter
Masked production	Production (or a subset) with sensitive values replaced	Weak masking can leak; masking must stay consistent across tables	the masking concepts chapter
Synthetic (generated)	Created from rules, no real person behind any row	May not look like the messy real world unless the rules are good	This section
Hand-crafted	A person types or edits specific rows	Slow, hard to repeat, tiny volume	This section
Hybrid	Masked or queried production plus generated rows added	Needs a clear plan for which part comes from where	Section 3.6

Hybrid is more than a theory. GenRocket's FAQ says it can query existing production data and merge it with synthetically generated data, and its G-Subset overview lists "test data augmentation" (adding a richer set of data to the destination) as one of its usage options. GenRocket General FAQ ↗

3.3.2 By sensitivity

Sensitivity is how much harm follows if the data leaks. the sensitive data and regulations chapter covers PII, PHI and the regulations in depth. For classification purposes you only need a simple ladder. The labels differ between companies; the ladder below is a common shape, not a standard.

Level	Meaning	ShopSphere example	Allowed in test?
Public	Harmless if seen by anyone	`PRODUCTS.product_name`	Yes
Internal	Business data, not personal	`ORDERS.total_amount` with no customer link	Yes
Personal	Identifies or relates to a person	`first_name`, `email`, `phone`, `zip_code`	Only masked or synthetic
Highly restricted	Identifiers regulators care most about	`ssn`, `tax_id`, `date_of_birth`	Only masked or synthetic, and strongly protected

Informatica's own introduction frames the same idea: you identify sensitive information in large volumes of production data, and its profiles can compare data against compliance standards, naming PII, PCI and PHI as examples (User Guide 10.5.10, p.24). The key point for classification is simple: **sensitivity decides what you are allowed to do to the data; source and purpose decide what you do with it.** Synthetic data has no real person behind it, which is why it needs no masking step.

3.3.3 By purpose

This is the line testers care about most. Each purpose answers "what must this row prove?". There are seven that cover almost every request.

Purpose	What it proves	ShopSphere example
Positive / valid	The normal path works	Asha Menon, 560001, valid email, two orders
Negative / invalid	Bad input is rejected with a clear message	Phone `98765abcde`; email with no `@`
Boundary	The edges of each rule behave exactly right	Customer aged 17, 18 and 19; surname of 1 and of the maximum length
Edge case	Rare or odd but legal situations	Surname `O'Brien`, hyphenated name, name with leading spaces, order with 50 items
Null / empty	Missing values do not crash anything	No phone; empty `zip_code`; customer with zero orders
Volume / performance	The system copes with realistic size	2 million customers, 10 million orders
Security	Hostile input is blocked	Name field containing `'; DROP TABLE ORDERS;--`

NEGATIVE DATA USUALLY DOES NOT LIVE IN THE TABLE

A database column with a 10-digit phone constraint will refuse `98765abcde` when you try to load it. Negative and security data are normally sent in through the screen, the API or an input file, not inserted straight into the table. Only load invalid values into a table if the table has no constraint and the test is about how the application reads bad stored data (for example a legacy table).

A vendor's view of the same list: in its 2022 overview video, GenRocket's CEO says masked copies of production leave "data gaps" and lists what its customers need instead: positive and negative data, *unique permutations* (every combination of values worth testing), *stateful data* (rows in a particular condition, such as a certain step of a complex workflow), referential integrity throughout, rules that condition the output, any volume from small to huge, and any file or database format. Most of this is the table above in other words. The additions to remember are *state* (a row's condition, not just its values) and *permutations* (all combinations, not single edge values). This is the vendor's pitch, not a measured finding.

3.4 The data patterns on the map

There are five data patterns: **boundary conditions, sequential, random, edge cases and null**. They are not a separate classification. They are *ways of shaping values*, and each one serves one or more of the purposes in 7.3.3. the GenRocket data patterns chapter shows them in GenRocket; here we place them on the map.

Pattern	What the values look like	Mainly serves purpose	ShopSphere example
Boundary conditions	Values on and just either side of every limit	Boundary	Ages 17, 18, 19; `quantity` 0, 1, max, max+1
Sequential	Values that count up (or down) in a fixed step	Valid, volume, unique keys	`customer_id` 2001, 2002, 2003 …
Random	Values picked unpredictably from a range or list	Valid, volume, realism	First names picked from a list; zip codes in a range
Edge cases	Unusual but possible values, placed on purpose	Edge case, negative	A negative age every fifth record; names over the length limit 25% of the time
Null	Missing, empty or blank values	Null / empty	5 customers with no phone

The two documented tools describe these same shapes in their own words. This is the evidence for the table above:

- **GenRocket** has a page on *Synthetic Data Replacement* that lists test scenarios for one column (a Social Security Number): sequential replacement (`001-00-0001` , `001-00-0002`), valid-format replacement, character sets that inject invalid characters, boundary values such as `000-00-0000` and `999-99-9999` , missing or null data, illegal-format testing and volume stress testing. What is SDR? ↗
- GenRocket's `EdgeCaseGen` page gives a story where the valid age range is 1 to 120, and a negative age is produced on every fifth record by cycling between a normal-range source and an edge-case

source. EdgeCaseGen ↗

- **Informatica** rules have an "exception test data" section where you pick how many rows should be null and how many should hold values that break the rule, each as a percentage with up to two decimal places (User Guide 10.5.10, pp.210 to 211). A rule can also use a sequence technique for ordered numbers or dates and a random technique for random strings, numbers and dates (pp.209 and 220).

WORKED EXAMPLE: ONE RULE, FIVE PATTERNS

Rule: "a customer must be at least 18 years old on the day of registration". Today, in the test, is 2026-10-23. Here is the same `date_of_birth` column shaped five ways.

EXAMPLE DATE_OF_BIRTH VALUES FOR THE 18+ RULE (EXAMPLE DATA)

Pattern	Values	Expected application result
Boundary	2008-10-24 (17 years, 364 days), 2008-10-23 (exactly 18), 2008-10-22 (18 years, 1 day)	Reject, accept, accept
Sequential	1980-01-01, 1980-01-02, 1980-01-03 ...	All accepted; useful for volume and unique ordering
Random	Any date between 1950-01-01 and 2008-10-22	All accepted; realistic age spread
Edge case	1900-01-01, 2024-02-29 (a leap day)	Application decides: accept, or reject as implausible
Null	empty	Reject with a "date of birth required" message

Notice the boundary row. A boundary test needs *three* values per limit: just inside, on, and just outside. Many bugs are off-by-one mistakes (using "greater than" where "greater than or equal" was meant), and only the exact boundary value catches them. Random data hits the exact boundary almost never. That is the main reason boundary data must be *designed*, not left to chance.

3.5 What synthetic data is

Concept. Synthetic data is data created by a program following rules you set. No row in it was ever a real person's record.

Simple explanation. A film set looks like a street, but nobody lives in the houses. Synthetic data looks like customers, but no customer exists. It is built to be *good enough to exercise the application*, not to copy reality.

The analogy stops at one point: synthetic data must look right from every angle the application views it, with foreign keys that join, dates in order and totals that add up.

What the tools say.

- GenRocket: "Synthetic Test Data" is data manufactured or generated by an algorithm rather than real-world events. Its stated benefit is that companies can maximise test coverage by testing all permutations and combinations. Why Synthetic Test Data? ↗
- Informatica: data generation creates realistic test data for test environments, using generation rules that hold the logic. You import the metadata (table and column definitions) so the tool knows which columns to fill, assign rules to columns, then create a plan and run it (User Guide 10.5.10, p.190).

Both tools need only the *shape* of the target (the metadata) and a *rule* for each column, never a real customer.

> **A REAL-WORLD EXAMPLE FROM THE INFORMATICA GUIDE**
>
> The guide's own example is an airline that sells plane tickets. Rules load dictionary values such as names into the customer tables, create random number strings for ticket numbers, create a numeric sequence for identification numbers, and use a reference lookup for values such as airport codes (User Guide 10.5.10, p.191). For ShopSphere: a name dictionary for `first_name`, a sequence for `customer_id`, and a lookup of valid `product_code` values.

3.6 Synthetic data versus masked production data

Neither is "better". They solve different problems, and mature teams use both. The table compares them on the dimensions that matter. The ratings are general experience, so each column heading carries the "General concept" badge rather than a product claim.

Question	Masked production data	Synthetic data
Realism	High. Real shapes, real mess, real correlations between columns	As good as your rules. Looks real only for the properties you modelled
Coverage of rare or future cases	Only cases that already happened in production	Any case you can describe, including ones that never happened, and boundary or invalid values on demand
Privacy	Depends on masking quality. Weak masking, or unmasked columns, can still identify people	No real person behind any row, so there is nothing to re-identify
Setup effort	Discovery, masking rules, consistency checks	Modelling the tables, writing rules for each column and relationship
Cost to refresh	Re-run extract and mask, which grows with production size	Re-run the generator; speed depends on the rows asked for, not on production size
Referential integrity	Comes from the real data, but masking must keep keys matching across tables	Must be designed: parent rows first, children point at existing parents (section 3.8)
Distributions	Real distributions come free	You must state them (for example 20% of customers with no orders)
Repeatability	Same snapshot until refreshed	Same output every run if you use a seed
Volume on demand	Limited by what production holds	As many rows as the target can hold
Main risk	A leaked or unmasked column	A model that is too tidy and misses real-world mess

GenRocket's SDR page makes one claim worth noticing, which is the vendor's view: synthetic data "can produce edge cases that do not currently exist in production" and "can yield more accurate data patterns than exist in production". That is a statement about coverage, and it is true for any generator that

supports designed edge cases. It does not mean synthetic data is automatically more realistic than production, which it is not.

3.6.1 Which one should you choose?

1. **Is a realistic mix of messy data essential, such as for a new analytics report or a migration rehearsal?**
   Lean towards masked production, because the real mess is the thing being tested.

   Expected result: Result: use the subsetting and masking techniques from the subsetting and masking concept chapters.

2. **Is the system new, so there is no production data yet, or are you testing a feature with no history?**
   Generate. There is nothing to copy.

   Expected result: Result: use rules from this section.

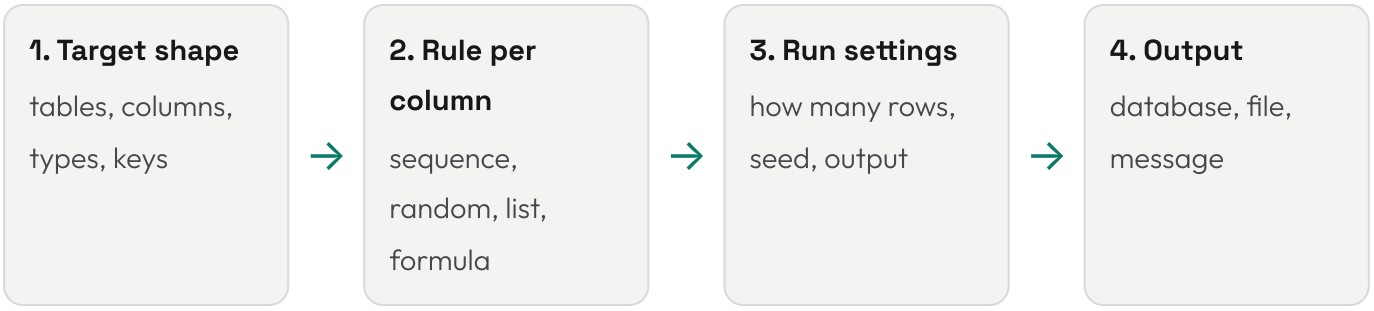
3. **Do you need exact boundary, negative or very large data sets?**
   Generate. Production rarely holds the 51st item in an order or a customer aged exactly 18 today.

4. **Do you need both realism and designed edge cases?**
   Use a hybrid: masked subset as the base, generated rows added on top, with generated keys chosen so they never clash with the real ones.

> **GENERATED IS NOT THE SAME AS SAFE BY DEFAULT**
>
> If a generator copies a real value (for example, picks names from an unmasked production table as its dictionary) the result is no longer purely synthetic. Always check where a generator's source list came from. Dictionaries should be made-up or public lists, never an export of real customers.

3.7 How generation works: rules, generators, seeds and distributions

Concept. A generation job has three layers. The *target* says which tables and columns to fill. A *rule* (Informatica's word) or an assigned *generator* (GenRocket's word) says how to fill each column. A *run* says how many rows and where to put them.



3.7.1 The common kinds of generator

Almost every column in a business table can be filled by one of about eight ideas. The table shows each idea, then the names the two tools use for it. Only names the documentation states are listed.

Idea	ShopSphere column	Informatica technique	GenRocket generator named in its docs
Count up in steps	`customer_id`	Sequence (numeric or date)	`RangeGen` (FAQ: starts a sequence from 1)
Random in a range	`zip_code`, `quantity`	Random	`RandomGen` (seeding page: random numbers 1 to 25)
Pick from a list of real-looking words	`first_name`	Dictionary	`NameGen`, `ListGen`
Pick from a small fixed set	`order_status`	Set of Values	`ListGen` (list generators)
Follow a pattern	An order reference such as 4 digits plus "St."	Random with a regular expression pattern	`SimplePatternGen` (default fallback, appends a number to the attribute name)
Combine other columns	`email` from first name plus domain	Advanced rule, or Expression	`EmailGen`, `ConcatGen`; "Linked Generators" chain several
Copy from another table	`ORDERS.customer_id`	Reference Lookup	`ReferenceGen` (with `ProxyReferenceGen` at domain level)
Depends on a condition	`age_group`: "adult" or "minor"	Conditional	Conditional Logic Generators, G-Rules

Sources: Informatica technique list (User Guide 10.5.10, p.209) and ad hoc, standard and global rule types (pp.192 to 193); GenRocket "Generators Overview", "How are Generators Assigned to Domain Attributes?", "What is Seeding?", the General FAQ, and the "What is the difference between the ProxyReferenceGen and ReferenceGen?" page. Both tools offer more techniques than these. The Informatica data generation chapter and the GenRocket generators chapter list them.

SAME IDEA, DIFFERENT PACKAGING

Informatica separates the *rule* (reusable, named, stored in the Policies area) from its *assignment* to a column (User Guide 10.5.10, pp.191 to 193). GenRocket attaches a *generator* directly to each attribute and picks a default for you from the attribute and domain name, the project's industry type and usage patterns (GenRocket "How are Generators Assigned to Domain Attributes?"). In both, you can replace what was picked.

3.7.2 Seeds: making ”random” repeatable

Concept. Computers do not produce truly random numbers. They run a formula that produces a long, random-looking list, starting from a number called the *seed*. Same seed, same list.

Why it matters. Suppose a test fails on Tuesday because customer 2003 had a zip code with a leading zero. If the data is regenerated on Wednesday with different random values, you may never see that customer again. With a fixed seed you can rebuild exactly Tuesday's data and debug it. Change the seed on purpose when you want fresh data.

GenRocket describes this directly: seeding generates the same set of random data for the same seed value every time, a seed can be set on the domain (a global seed variable) or in an attribute's generator configuration, and two attributes using the same generator and same seed produce the same random series. What is Seeding? ↗

The snippet below illustrates the principle in plain Python (not a vendor tool). It builds the 50 customers and orders designed in section 3.10. Run it twice with seed 42 and you get identical data; run it with seed 7 and you get different data.

```
import random

def generate(seed):
    rnd = random.Random(seed)              # the seed lives here
    first = ["Asha","Ravi","Meera","Kiran","Divya","Arjun","Sana","Vikram"]
    last  = ["Menon","Kumar","Nair","Iyer","Rao","Shah","Das","Pillai"]
    customers = []
    for i in range(50):
        cid = 2001 + i                     # sequential: 2001, 2002, ...
        customers.append((cid, rnd.choice(first), rnd.choice(last),
                          "%06d" % rnd.randint(560001, 560099)))   # random
    # exact distribution: 10 customers get 0 orders, 20 get 1, 15 get 2, 5 get
5
    counts = [0]*10 + [1]*20 + [2]*15 + [5]*5
    rnd.shuffle(counts)                    # random WHO, exact HOW MANY
    orders, oid = [], 9001
    for cust, n in zip(customers, counts):
        for _ in range(n):
            orders.append((oid, cust[0], rnd.randint(1, 4)))  # parent id,
items
            oid += 1
    return customers, orders

c1, o1 = generate(42)
c2, o2 = generate(42)
c3, o3 = generate(7)
print(len(c1), len(o1), sum(x[2] for x in o1))   # 50 75 179
print(c1 == c2 and o1 == o2, c1 == c3)           # True False
```

Expected result: Expected result: the first print shows `50 75 179` and the second shows `True False`. The first three customers with seed 42 were, under Python 3.13, `(2001, 'Ravi', 'Menon', '560095')`, `(2002, 'Divya', 'Iyer', '560029')` and `(2003, 'Meera', 'Kumar', '560087')`. The item total 179 and these exact names depend on the random generator inside your Python version, so yours may differ. What must hold on your machine is `True False`.

3.7.3 Distributions: how values are spread

If every customer got exactly two orders, the data would be repeatable but unrealistic. Real shops have many customers with none, most with one or two, and a few with many. The snippet above states this on purpose: 10, 20, 15 and 5 customers get 0, 1, 2 and 5 orders. That is a *distribution*.

Both tools let you state percentages rather than writing a loop:

- Informatica's random technique lets a regular expression create a fixed percentage of the generated values, and pattern percentages must total 100 (User Guide 10.5.10, p.220). Null and invalid-value rows are also set as percentages (pp.210 to 211).
- GenRocket's `MultiWeightGen` assigns weights (a percentage and a value) so that, for example, 50% of parents have 2 children, 20% have 3 and 30% have 1. With `exactPercentage` set to true, ten parents give exactly five, two and three parents of each type. Generate Children based on Weights ↗

> **EXACT OR APPROXIMATE?**
> With "10% of rows null", you can ask for roughly 10% (each row has a 10% chance) or exactly 10% (5 of 50). For small data sets such as 50 customers, always choose exact, otherwise a run can produce zero nulls and your null test silently tests nothing.

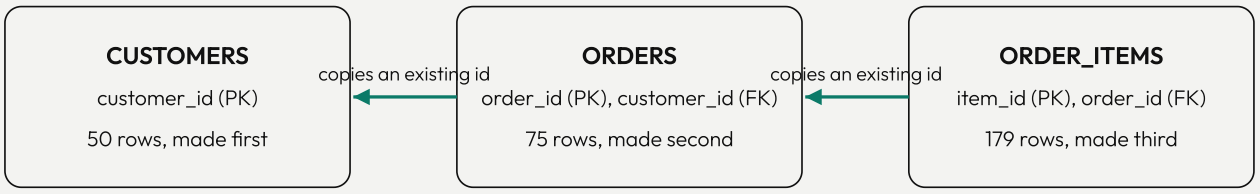
3.7.4 Rules that link columns

Some values only make sense in combination. A shipped date must come after the order date. A flight cannot leave and arrive at the same airport. Informatica's guide describes a *conditional constraint*: an expression such as `To_City ≠ From_City` that restricts values in one column based on another (User Guide 10.5.10, pp.201 to 202). When the rule's own limits make the constraint impossible (for example a modified date that must be later than a created date of 31 July 2014, the last date the rule allows), the rule wins and the tool generates data with warning messages. The lesson for your own designs: when two rules fight, decide which one is the boss, and expect warnings from the other.

3.8 Generating parent and child tables so every key is valid

Concept. A child row needs a parent that exists. If you generate ORDERS first and pick `customer_id` values at random, some will point at customers that do not exist, and the database (or the application's joins) will fail.

Simple explanation. You cannot write a library card for a member who has not been registered. So you register members first, then issue cards that carry a registered member's number.



Generation order: left to right. Arrows show which table each foreign key points at.

Original diagram: parents are generated before children, and every foreign key copies an id that already exists. Row counts are from the design in section 3.10.

3.8.1 Two common ways to keep keys valid

A. Child picks from existing parents

Generate the whole parent table first. Then, for each child row, choose a parent id from the finished list.

Informatica calls this a *reference lookup* rule: you name the parent table and assign parent columns to child columns (User Guide 10.5.10, p.229).

Good when the parent table already exists, for example loading new orders against real or masked customers.

B. Children are made inside each parent

Loop over parents. For each parent, decide how many children it has, then create exactly that many, stamping the parent's id on each.

GenRocket works this way: a parent domain's loop count and the child's count per parent drive the output, and it generates data sequentially along the relationships (see its "Parent, Child, and Sibling Relationships" page).

Good when both tables are being created together, and when you want a controlled number of children per parent.

GenRocket's page also says every domain has an id or primary attribute that is used to guarantee referential integrity between domains, and its FAQ says it can generate synthetic data with full referential integrity. It introduces a "sibling" idea: a table that merely describes another (the page's example is a role table describing a user table) is marked as a sibling instead of a parent, to avoid generating unnecessary data. Parent, Child, and Sibling Relationships ↗

3.8.2 How many children per parent?

Informatica's plan settings for data generation include a **minimum and maximum number of child records for each parent record, each with a default of 1**, and percentages for "parent records without children" and "child records without parent" for self-cyclic tables and reference lookup rules (User Guide 10.5.10, p.285). Those last two settings exist because sometimes you *want* a customer with no orders (a null-style condition) or, rarely, a deliberately orphaned child (for a data-quality test). Do this only on purpose and label it, so nobody mistakes it for a bug.

> **SELF-REFERENCING TABLES**
>
> A table whose foreign key points at itself (an employee table with a manager id) has no "parents first" order. The tool needs a special rule: the first rows have no parent, later rows point back at earlier ones. Informatica's setting names self-cyclic tables specifically (p.285). ShopSphere has none.

3.9 The lifecycle: Model, Design, Deploy, Manage

"Model, Design, Deploy, Manage" is GenRocket's own methodology name in its help centre, and in its 2022 overview video GenRocket's CEO presents it as the company's method. The GenRocket Methodology ↗

In that video the CEO describes the steps like this. **Model:** a customer uploads a data model, meaning the table and column definitions and not the real rows, to the GenRocket cloud, where an "intelligent data warehouse" (the vendor's name for its store of generator knowledge) and some machine-learning methods speed up setting up a *test data project*. **Design:** a test data engineer designs the data each test requirement needs, in a suite of test data cases (the demo shows unit, negative, load and integration), with the row count per table set separately and rules applied to the output. **Deploy:** a developer plugs the data into a CI/CD pipeline (an automated build-and-test pipeline) or a single test case and generates it on demand. The video does not unpack "Manage" in that passage; elsewhere the CEO says a project can be versioned and life-cycle managed to stay current with the application or database, and the 2020 webinar likewise mentions updating the model and managing scenarios as things change.

The four words also describe a cycle that any generation effort goes through, in any tool. The table below is a tool-neutral mapping of the phrase onto ShopSphere work.

Phase	Question it answers	ShopSphere activity	What you produce
Model	What does the data look like?	List the tables, columns, types, keys and relationships (CUSTOMERS, ORDERS, ORDER_ITEMS, PRODUCTS)	A data model the tool can read, often imported from an existing database definition
Design	Which values, how many, and why?	Choose a rule per column, set the distributions, decide the boundary and null rows (section 3.10)	Rules, counts and a coverage matrix
Deploy	How do the rows reach the test system?	Run the generation, write to the QA database or to files, check keys and counts	Loaded, verified test data
Manage	How do we keep it right and reusable?	Version the rules, record the seed, handle requests for changes, retire old data sets	A catalogue of data sets and a change history

The docs agree on who does what. GenRocket's General FAQ gives the complex tasks to a small group of trained "Test Data Engineers" who *model* and *design* the data, while testers and developers search for data and *deploy* it into their environment. The architecture page adds that modelling and design happen in a cloud component and generation runs locally, on premises. GenRocket Architecture ↗

GenRocket also names five building blocks that the phases work on. Its "Five Key Components" page: a *domain* is a noun such as a table or file; an *attribute* is a characteristic of a domain, such as a column; a *generator* produces the values for an attribute; a *receiver* turns generated data into a usable format such as SQL, CSV or JSON; and a *scenario* holds the instructions for what to generate and how much. GenRocket Five Key Components ↗ Placed on the cycle, domains and attributes are mostly Model work, generators and scenarios are Design work, receivers carry the Deploy step. The GenRocket overview chapter takes this further.

WHERE INFORMATICA FITS ON THE SAME CYCLE

Informatica's task flow has the same shape: import metadata into a project (model), create generation rules and assign them to columns (design), create a plan, generate a workflow and run it (deploy), then watch it in the Monitor view (User Guide 10.5.10, p.192). The guide does not use the words Model, Design, Deploy, Manage; the mapping is mine.

3.10 Worked example: 50 ShopSphere customers and their orders

The request. The QA team testing ShopSphere's Customer Orders app asks the TDM engineer for a new data set in a QA database: 50 customers, with orders and items, covering the registration rules, the order history page and a customer-search feature. No production data may be used. We follow Model, Design and Deploy; Manage is covered by the lab at the end.

3.10.1 Model

The tables are the ShopSphere tables from the data basics chapter. The relationships are the only thing the generator needs to respect: `ORDERS.customer_id` points at `CUSTOMERS`, `ORDER_ITEMS.order_id` points at `ORDERS`, and `ORDER_ITEMS.product_code` points at `PRODUCTS`. PRODUCTS is a small reference table of four rows that we load by hand (P-100 Laptop Stand, P-200 USB Cable, P-300 Phone Case, P-400 Desk Lamp), because it is better to control the catalogue than to invent products.

The business rules for this example are what the test conditions will aim at: *customers must be 18 or older; first and last names are at most 30 characters; phone is 10 digits when present; zip code is 6 digits when present; email is optional; each order has between 1 and 4 items.*

3.10.2 Design: a rule for every column

GENERATION RULES FOR CUSTOMERS (EXAMPLE DESIGN; THE 'SSN' PREFIX 900 AND 'EXAMPLE.TEST' DOMAIN ARE CHOSEN SO VALUES CANNOT BE MISTAKEN FOR REAL ONES)

Column	Rule	Sensitivity
customer_id	Sequential from 2001, step 1	Internal
first_name	Random pick from a made-up list of 200 names	Personal (synthetic)
last_name	Random pick from a made-up list of 200 surnames	Personal (synthetic)
email	first name + "." + customer_id + "@example.test", lower case	Personal (synthetic)
phone	"9" followed by 9 random digits	Personal (synthetic)
ssn	"900-" then 2 random digits, "-" and 4 random digits; unique	Highly restricted (synthetic)
date_of_birth	Random between 1950-01-01 and 2008-10-22	Highly restricted (synthetic)
zip_code	Random 6 digits from 560001 to 560099	Personal (synthetic)
tax_id	Pattern of 5 letters, 4 digits, 1 letter; unique	Highly restricted (synthetic)

Next, the customers are split by purpose. The counts below must add up to 50, and each group overrides the default rules for the columns it targets.

CUSTOMER GROUPS BY PURPOSE

Group	Rows	What differs from the default rules
Typical (positive)	33	None
Boundary	6	1 customer exactly 18 today (born 2008-10-23); 1 aged 18 years and 1 day (2008-10-22); first_name of 1 character; first_name of 30 characters; last_name of 30 characters; zip_code 560001 (lowest in range)
Null / empty	5	phone NULL (2 rows); email NULL (1); zip_code empty string (1); tax_id NULL (1)
Edge case	6	last_name O'Brien; hyphenated last_name; first_name with a leading and trailing space; accented letters in first_name; date_of_birth on a leap day (2000-02-29); one customer sharing another customer's last_name and zip_code (to test search)
Total	**50**	

Invalid and security values (letters in a phone number, an email with no "@", a name that contains SQL text) are *not* in the table. They go into a separate four-row input file that the tester types or posts through the registration screen, because the database would refuse to store them. That is the point made in the tip in 7.3.3.

Now the children. Orders per customer follow the distribution below, drawn from the Python snippet in 7.7.2.

ORDERS PER CUSTOMER

Orders per customer	Customers	Share	Orders created
0	10	20%	0
1	20	40%	20
2	15	30%	30
5	5	10%	25
Total	**50**	100%	**75**

Each order has 1 to 4 items picked at random, each item has a random product code from PRODUCTS, a quantity of 1 to 3, and a `unit_price` copied from that product's `price` (so prices always agree with the catalogue). With seed 42 that gave 179 items. `total_amount` on each order is calculated as the sum of its items, so totals always match. Without that last rule the data would be internally contradictory, and tests of the order total report would fail for the wrong reason.

3.10.3 The coverage matrix

The matrix shows which data group serves which test area. An empty column is a gap. A tick means a row exists that exercises that area.

COVERAGE MATRIX: DATA GROUPS AGAINST TEST AREAS (✓ = COVERED)

Data group	Registration validation	Customer search	Order history page	Order total report	Privacy export
Typical (33)	✓	✓	✓	✓	✓
Boundary (6)	✓	✓			
Null / empty (5)	✓	✓			✓
Edge case (6)	✓	✓	✓		✓
Customers with 0 orders (10)			✓	✓	✓
Customers with 5 orders (5)			✓	✓	
Negative input file (4 rows, not loaded)	✓				

Look at the empty cells. Boundary customers have no tick for the order history page, because the design does not say whether they have orders. A good review catches that. Fix it by requiring that at least two boundary customers and two null-phone customers are among the 40 customers who have orders, so the order history page is also tried with them.

3.10.4 Deploy and verify

After the run, check three things before you hand the data over. The SQL checks below work on most databases.

```
-- 1. No orphan orders (expected: 0)
SELECT COUNT(*) FROM orders o
LEFT JOIN customers c ON c.customer_id = o.customer_id
WHERE c.customer_id IS NULL;

-- 2. The orders-per-customer distribution matches the design
SELECT n AS orders_per_customer, COUNT(*) AS customers
FROM (SELECT c.customer_id, COUNT(o.order_id) AS n
      FROM customers c
      LEFT JOIN orders o ON o.customer_id = c.customer_id
      GROUP BY c.customer_id) t
GROUP BY n ORDER BY n;

-- 3. The null and empty rows really exist (expected: 2, 1, 1, 1)
SELECT SUM(CASE WHEN phone IS NULL THEN 1 ELSE 0 END)    AS null_phone,
       SUM(CASE WHEN email IS NULL THEN 1 ELSE 0 END)    AS null_email,
       SUM(CASE WHEN zip_code = '' THEN 1 ELSE 0 END)    AS empty_zip,
       SUM(CASE WHEN tax_id IS NULL THEN 1 ELSE 0 END)   AS null_tax_id
FROM customers;
```

Expected result: Expected result: check 1 returns 0. Check 2 returns four rows: (0, 10), (1, 20), (2, 15), (5, 5); the snippet in 7.7.2 gives exactly this. Check 3 returns 2, 1, 1, 1 once the null and empty overrides from the design table are applied. The snippet in 7.7.2 generates only the typical shape, so check 3 is a template to run on a real design.

3.11 Test data as a service

Concept. "Test data as a service" means test data is something teams can *ask for and receive*, like a ticket to a service desk, instead of something each team builds in secret. A small group owns the know-how and the rules; many teams consume the results. The phrase itself does not appear in the tool documentation; it labels what the three tools describe in their own words.

Why it matters. Without it, every team writes its own scripts, nobody can reproduce another team's data and the same sensitive columns get masked ten different ways. With it, a request is a small form ("50 customers with orders, boundary included"), and the answer is a catalogued, repeatable data set.

Tool	What its docs say that fits the idea
GenRocket	G-Portal is described as part of a distributed self-service platform in which a small group of Test Data Engineers in a Center of Excellence supports large distributed teams. Requesters search for existing test data, create requests, track them and download components. G-Portal Overview ↗
GenRocket	Test Data Cases (G-Cases) change the components inside a scenario so the same scenario can produce many permutations for different test cases, organised in suites with categories such as Beta, Unit or Load. G-Cases Overview ↗
Informatica TDM	A test data warehouse stores data sets; the guide says multiple teams using different machines that host a Test Data Manager Service in a domain can access a single test data warehouse (User Guide 10.5.10, p.24).
Delphix	The terminology page describes Delphix Self-Service as a user interface designed for project teams, application developers and testers to manage their virtualised datasets. Delphix masking terminology ↗

WHAT A SHOPSPHERE REQUEST MIGHT LOOK LIKE

Requester: QA engineer. **Need:** customers with 0, 1, 2 and 5 orders; 5 null-phone customers. **Environment:** QA2. **Refresh:** weekly, same seed. **Owner:** TDM engineer. **Delivered as:** a named data set "SHOPSPHERE-CUST-50-v3" with the rule version and seed recorded. This record is the "Manage" phase of 7.9 in miniature.

3.12 Hands-on lab: design a smaller data set on paper

Lab: model and design 20 customers

Goal: practise turning a request into a Model, a Design and a coverage matrix. You need pen and paper or a text editor; no tool is required.

1 **Write the request.**

"20 customers with orders for the order history page. Customers must be 18 or older. Zip code is 6 digits."

Expected result: You should have one sentence naming the tables, the count and the rules.

2. **Split the 20 customers by purpose.**
   Use typical, boundary, null or empty and edge case. The groups must add up to 20.

   Expected result: A table with four rows and a total of 20.

3. **Pick an orders-per-customer distribution.**
   Choose counts that add up to 20 customers and state the total orders.

4. **Draw a coverage matrix** with your groups down the side and the test areas "registration" and "order history" across the top. Mark ticks and find one gap.

5. **Write two SQL checks** you would run after generation.

▶ **Show a worked answer**

3.13 Common mistakes and gotchas

MISTAKES BEGINNERS MAKE

- **Generating children before parents.** The foreign keys then point at nothing. Always parents first, or have the child rule read from the finished parent table.
- **All-random data and no designed edges.** Random values almost never hit an exact boundary, a null, or a name with an apostrophe. Put those rows in on purpose.
- **Not recording the seed.** A failed test cannot be reproduced if the data cannot be rebuilt.
- **Requesting "about 10% null".** In a 50-row data set that can come out as zero. Ask for an exact count.
- **Totals that do not add up.** Derive `total_amount` from the items instead of generating it separately.
- **Loading invalid values into constrained columns.** The load fails or the constraint is silently dropped. Keep negative data in the input channel.
- **Treating synthetic as automatically realistic.** It only mirrors what you modelled. Compare a few generated distributions against real, masked, aggregated figures.
- **Reusing real values as the "dictionary".** A name list exported from production defeats the purpose.

3.14 Recap

RECAP

- Classify test data along three lines: **source** (production-derived or synthetic), **sensitivity** (what harm a leak causes) and **purpose** (what the row must prove).
- The seven purposes are valid, invalid, boundary, edge case, null or empty, volume and security. The data patterns (boundary, sequential, random, edge cases, null) are ways of shaping values to serve them.
- Synthetic data is created from rules. It gives privacy and designed coverage; masked production gives real-world mess. Hybrids use both.
- Generation needs a target shape, a rule or generator per column, and run settings. A **seed** makes it repeatable and a **distribution** makes it realistic.
- Generate parents first and make every foreign key copy an existing parent id; control how many children each parent gets.
- GenRocket's cycle is Model, Design, Deploy, Manage. Test data as a service means requestable, catalogued, repeatable data owned by a small expert group.
- A coverage matrix shows gaps before the testers find them.

ETL from scratch: extract, transform, load

Every tool in this guide moves data. Informatica TDM copies a slice of a database, changes some values and writes it somewhere else. Delphix reads a source, masks it and delivers it to a target. GenRocket creates rows and writes them to a database or a file. Underneath, all three do the same three things: **get data from somewhere, change it, put it somewhere else**. The industry has had a name for that pattern for decades: **ETL**, which stands for Extract, Transform, Load.

This section covers ETL on its own, without test data management, so the pattern is recognisable wherever it appears. It builds a small but complete ETL job for ShopSphere in plain SQL, then describes the machinery that real ETL tools wrap around those SQL steps: schedulers, logs, error handling and reusable components. The ETL and TDM chapter lays the TDM tools on top. Tables and keys are explained in the data basics chapter.

SQL DIALECTS

The SQL here is standard in spirit and was written for SQLite, but functions for text, dates and "insert or update" differ between databases (Oracle, SQL Server, PostgreSQL, MySQL), so a few function names may need adjusting. The main ones are pointed out as they come up.

3.15 Words to know first

ETL

Extract, Transform, Load: copy data out of one or more sources, reshape and clean it, and write it into a target.

Staging area

A temporary landing zone, usually a set of tables or files, where extracted data sits unchanged before it is transformed.

Operational database (OLTP)

The database an application uses to run the business: place an order, update an address. Built for many small, fast reads and writes.

Data warehouse

A separate database built for reporting and analysis across lots of history. Built for a few big reads.

Full extract / incremental extract

Reading everything every time, versus reading only what is new or changed since last time.

Watermark

A saved marker, such as "last order id loaded was 5005", that tells the next run where to start.

Change data capture (CDC)

Any technique that detects which rows were inserted, updated or deleted in the source so only those are processed.

Upsert

"Update if the row exists, insert if it does not." Short for update plus insert.

Slowly changing dimension (SCD)

A way of keeping history in a warehouse table when descriptive data (an address) changes over time.

Mapping

The definition of how source columns become target columns, including every transformation in between.

Workflow, session, job

The runnable wrappers around mappings. A job or workflow runs one or more sessions or steps on a schedule.

Mapplet

Informatica's word for a reusable group of transformations that you build once and drop into many mappings.

Metadata

Data about data: table definitions, column types, mapping definitions, run history.

Reject (error) row

A row the job could not process, written aside with a reason instead of stopping the whole run.

ELT

Extract, Load, Transform: load raw data into the target first and transform it there using the target database's own power.

3.16 What ETL is and why it exists

Concept. ETL is a repeatable process that moves data from where it is created to where it is needed, fixing it on the way.

Simple explanation. Think of a school canteen. Vegetables arrive muddy from several farms (*extract* brings them to the back door), staff wash, cut and combine them in the prep room (*transform*), and trays go to the serving counter (*load*). Each stage has its own place and rules.

The analogy stops being accurate here: an ETL job runs many times and must give the same kind of result each time, even if interrupted and restarted. Cooks can improvise; ETL jobs must not.

> **WHY IT MATTERS**
>
> Data in real companies lives in many places and many shapes. ShopSphere's orders sit in one database, product prices in a spreadsheet maintained by buyers, and a partner sends a daily file where dates are written day first. A manager who asks "how much did we sell last week, per product?" needs one clean answer. Someone has to collect, clean and combine. ETL is the name for doing that on purpose, automatically, and with a record of what happened.

ETL appears in data warehouses and reports (the classic case), system migrations, keeping two systems in step, feeding analytics, and test data provisioning, the topic of the next chapter.

3.17 Operational database versus data warehouse

ETL usually connects those two kinds of database, so the difference is worth knowing.

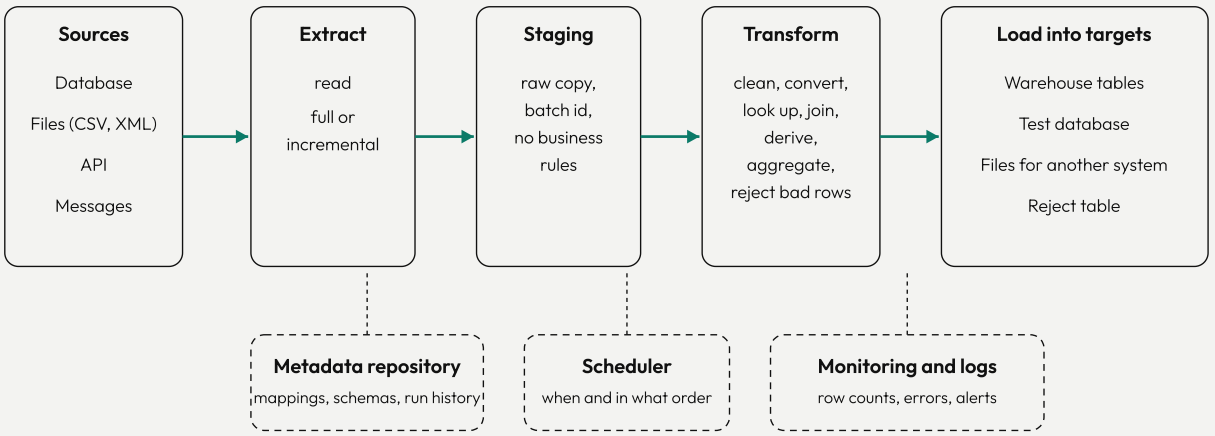
Question	Operational database (OLTP)	Data warehouse
Main purpose	Run the business, one transaction at a time	Answer questions across lots of history
Typical work	"Insert this order", "update this address"	"Total sales per product per month for three years"
Table shape	Many small tables, tightly normalised to avoid duplication (CUSTOMERS, ORDERS, ORDER_ITEMS)	Fewer, wider tables arranged around facts and dimensions
History	Mostly current values; old values are overwritten	History kept, often with dated versions of each row
Who writes	The application, all day	The ETL job, on a schedule

A common warehouse shape is the **star schema**: a central *fact* table that records events or measurements (one row per order line, with quantity and amount), surrounded by *dimension* tables that describe the who, what and when (customer, product, date). A fact table is mostly numbers and

keys; a dimension table is mostly descriptive text. We will build a tiny star for ShopSphere later in this section.

3.18 The big picture

Here is the whole pattern on one page. Read it left to right, then read the three boxes under the main flow, which sit around the flow and keep it running.



Original diagram: the ETL data path (top) and the supporting parts that control and observe it (bottom, dashed).

3.19 Extract: getting data out

Concept. Extraction reads data from a source system and hands it to the ETL process, without changing the source.

Why it matters. Extraction is where ETL jobs first meet the real world: slow networks, locked tables, files that arrive late or half-written, and sources that must not be slowed down because customers are using them. A good extract takes only what it needs, takes it consistently, and leaves the source as it found it.

3.19.1 Kinds of source

Source kind	How it is read	Watch out for
Relational database	SQL query over a database connection	Load on the source; reading while rows are being changed
Flat files (CSV, fixed-width, delimited text)	Read line by line using a layout definition	Different date formats, stray quotes, missing columns, a file that is still being written
Structured documents (XML, JSON)	Parse, then flatten nested parts into rows	Optional elements, repeating groups that become child tables
Web services (APIs)	HTTP requests, often in pages	Rate limits, authentication, changing response shapes

The Informatica guide shows the same range in test data work: it mentions relational sources, flat files and XSD (XML) sources as things that data subset, masking and generation plans can run on (User Guide 10.5.10, pp.274 to 277). GenRocket's FAQ describes its targets in the same vein (databases, flat files, mainframe formats, NoSQL). GenRocket General FAQ ↗

3.19.2 Full versus incremental extraction

Full extract

Read every row, every run.

Good: simple; no state to remember; picks up deletions automatically because the target is rebuilt.

Bad: slow and heavy as data grows; reads unchanged data again and again.

Use for: small reference tables such as PRODUCTS, and first-time loads.

Incremental extract

Read only rows that are new or changed since the last successful run.

Good: fast; kind to the source.

Bad: needs a reliable way to know "what changed"; misses deleted rows unless you plan for them.

Use for: large, growing tables such as ORDERS.

3.19.3 The watermark technique

The simplest incremental method is a **watermark**: store the highest value you have processed, and next time ask only for rows beyond it. It needs a column whose value only ever grows, such as an increasing order number or a "last modified" timestamp.

ShopSphere's order feed has an increasing `order_id`, so we keep a small control table with one row per job:

```
CREATE TABLE etl_control (
  job_name      TEXT PRIMARY KEY,
  last_order_id INTEGER
);
INSERT INTO etl_control VALUES ('orders_daily', 0);

-- Extract only new orders (the feed holds text, so convert for the
comparison)
INSERT INTO stg_orders
SELECT 1, o.*
FROM   feed_orders o
WHERE  CAST(o.order_id AS INTEGER) >
       (SELECT last_order_id FROM etl_control WHERE job_name =
'orders_daily');

-- After the whole run succeeds, move the watermark forward
UPDATE etl_control
SET    last_order_id = (SELECT MAX(CAST(order_id AS INTEGER)) FROM
stg_orders)
WHERE  job_name = 'orders_daily';
```

Expected result: Expected result: on the first run (watermark 0) five orders (5001 to 5005) were staged and the watermark moved to 5005. On the second run, with orders 5006 and 5007 newly added to the feed, only those two were staged and the watermark moved to 5007.

THREE WAYS A WATERMARK GOES WRONG

- **Moving the watermark too early.** If you advance it before the load finishes and the job fails, those rows are never retried. Advance it as the very last step, in the same transaction as the load when possible.
- **A column that is not strictly increasing.** If IDs can be assigned out of order by two servers, a late row with a smaller number is skipped forever. A "last modified" timestamp has the same trap when clocks differ.
- **Changes and deletes are invisible.** A watermark on `order_id` sees new orders, not edits to old ones, and never sees a deleted row. For edits use a last-modified column; for deletes use CDC or a periodic full comparison.

3.19.4 Change data capture at concept level

Change data capture (CDC) is the umbrella name for ways to learn *what* changed. There are four common approaches. Each product implements them differently, so check your tool's documentation before relying on one.

Approach	How it works	Catches deletes?	Cost to the source
Timestamp or version column	Query rows where `last_modified > watermark`	No	Low, but the table needs the column
Trigger-based	A database trigger writes every change into a side table that ETL reads	Yes	Adds work to every write
Log-based	Read the database's own transaction log to see inserts, updates and deletes in order	Yes	Low on the source, but needs special access and tooling
Compare snapshots	Keep yesterday's copy; compare with today's; the difference is the change	Yes	High: reads and compares everything

3.19.5 Good extraction habits

1. **Record the run.**
   Give every run a batch id and a timestamp, and write both next to the extracted rows. When something looks wrong later, you can say exactly which run brought it in.

2. **Read a consistent snapshot.**
   If you extract CUSTOMERS and then ORDERS five minutes later, new orders may point at customers you did not capture. Extract related tables in one transaction or point in time, or cut the orders to a time you know customers cover.

   Expected result: What you should see: every order in staging has its customer in staging, or the job knows why not.

3.20 The staging area

Concept. The staging area is a temporary landing zone between the source and the final target. Extracted data is stored there in nearly the same form as it was found.

Simple explanation. In the canteen, it is the delivery bay where crates are put down before anyone peels anything.

Why it matters. Staging gives you four things:

- **A frozen copy.** You can transform, check and re-run without hitting the source again.
- **Separation of duties.** Extraction only reads; transformation only reshapes; neither depends on the other being fast.
- **Safe restarts.** A failure in the transform step does not need a new extract.

Informatica's plans give you the same idea in their connection options. When you create a plan, you can choose a *staging connection*, "a connection to a database for staging data", and the default shown comes from the Preferences view (User Guide 10.5.10, p.276). That is a documented example of a staging area as a configured part of a real tool.

3.20.1 How to design staging tables

- **Mirror the source, with all columns as text.** A staged row should never fail because a date has a strange format. Conversion is the transform step's job.
- **Add audit columns.** At least `batch_id`; often a load timestamp and the source file name.
- **No keys, no foreign keys, no constraints.** Staging must accept bad data so that the transform step can decide what to do with it.
- **Empty it each run (or partition by batch).** Old staged data is usually not worth keeping beyond debugging.

```
CREATE TABLE stg_customers (
  batch_id    INTEGER,
  customer_id TEXT,
  first_name  TEXT,
  last_name   TEXT,
  email       TEXT,
  zip_code    TEXT
);
```

> **SHOPSPHERE STAGING, RUN 1**
>
> The extract wrote these rows into `stg_customers`, and notice it kept the mess exactly as found: the leading space in " Asha ", the upper-case "MENON", the trailing space in the email, the duplicate row for 1002 and the missing email for 1003.

STG_CUSTOMERS AFTER EXTRACT (BATCH 1)

batch_id	customer_id	first_name	last_name	email	zip_code
1	1001	" Asha "	MENON	"Asha.Menon@Example.com "	560001
1	1002	Ravi	Kumar	ravi.k@example.com	600017
1	1002	Ravi	Kumar	ravi.k@example.com	600017
1	1003	Meera	Nair	NULL	682001

The order feed is staged the same way. Two of its rows are traps handled later: order 5004 has a total of "abc", and order 5005 belongs to customer 9999, who does not exist.

3.21 Transform: cleaning and reshaping

Concept. Transformation turns staged raw data into the clean, consistent, correctly typed shape the target needs.

Why it matters. This is where business rules live. If the transform is wrong, the data is wrong everywhere downstream, and nobody can tell from the target alone. Transformations are therefore the part of ETL most worth documenting and testing.

There are nine everyday kinds of transformation: cleaning, type conversion, mapping, deduplication, lookups, filtering, joining, derived columns and aggregation. We cover each in turn using the ShopSphere staging rows from the previous section. Each subsection shows the idea, the SQL, and the before and after.

3.21.1 Cleaning

Idea. Make values consistent so that equal things look equal. "MENON", "Menon" and "menon " are the same surname to a person, but three different strings to a database.

```
SELECT
  UPPER(SUBSTR(TRIM(first_name),1,1)) || LOWER(SUBSTR(TRIM(first_name),2))
AS first_name,
  UPPER(SUBSTR(TRIM(last_name),1,1))  || LOWER(SUBSTR(TRIM(last_name),2))
AS last_name,
  COALESCE(LOWER(TRIM(email)), 'not provided')
AS email
FROM stg_customers;
```

CLEANING, CUSTOMER 1001 AND 1003

	first_name	last_name	email
Before (1001)	" Asha "	MENON	"Asha.Menon@Example.com "
After (1001)	Asha	Menon	asha.menon@example.com
Before (1003)	Meera	Nair	NULL
After (1003)	Meera	Nair	not provided

Function names differ by database (for example `SUBSTR` is `SUBSTRING` in SQL Server). Replacing a missing email with "not provided" is a business choice; some teams keep NULL.

3.21.2 Type conversion

Idea. Staging holds everything as text. The target needs real dates and numbers so that sorting, comparing and adding work properly. As text, "9" sorts after "10", and "20/10/2026" sorts before "21/09/2026" only by accident.

```
SELECT
  CAST(order_id AS INTEGER) AS order_id,
  SUBSTR(order_date,7,4) || '-' || SUBSTR(order_date,4,2) || '-' ||
SUBSTR(order_date,1,2) AS order_date,
  CAST(total_amount AS REAL) AS total_amount
FROM stg_orders
WHERE total_amount NOT GLOB '*[^0-9.]*';   -- only rows whose amount is
numeric
```

TYPE CONVERSION, ORDER 5002

	order_id	order_date	total_amount
Before (text)	"5002"	"20/10/2026"	"250.50"
After (typed)	5002	2026-10-20 (date)	250.5 (number)

Oracle and PostgreSQL would use `TO_DATE(order_date, 'DD/MM/YYYY')`; the string is sliced here because SQLite has no date parser with a format string. Confirm with the data owner whether dates are day-first or month-first, and reject dates that do not parse instead of guessing.

3.21.3 Mapping and standardising

The word **mapping** has two meanings in ETL, and both matter. In the broad sense a *mapping* is the whole definition of how source columns become target columns (we meet this in 8.11). In the narrow sense, *value mapping* means translating codes: the order system stores `S`, `P` and `C`, and reports want the words.

```
CREATE TABLE status_map (status_code TEXT PRIMARY KEY, status_name TEXT);
INSERT INTO status_map VALUES ('S','Shipped'),('P','Pending'),
('C','Cancelled');

SELECT o.order_id, COALESCE(m.status_name, 'Unknown') AS order_status
FROM   stg_orders o
LEFT JOIN status_map m ON m.status_code = o.status_code;
```

VALUE MAPPING OF STATUS CODES

order_id	status_code (before)	order_status (after)
5001	S	Shipped
5003	P	Pending
5006 (run 2)	C	Cancelled
5007 (run 2)	Z	Unknown

The unmatched code `Z` becomes "Unknown" because of the `LEFT JOIN` plus `COALESCE`. An inner join here would silently drop order 5007 altogether, which is one of the most common ETL bugs: **a**

lookup that loses rows when it fails to match.

3.21.4 Deduplication

Idea. Keep one row for each real-world thing. Customer 1002 arrived twice in the staging data.

```
SELECT customer_id, first_name, last_name, email, zip_code
FROM (
  SELECT s.*,
         ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY customer_id)
AS rn
  FROM stg_customers s
) t
WHERE rn = 1;
```

`ROW_NUMBER()` numbers the rows inside each customer id (1, 2, ...); keeping only number 1 removes the extras. The `ORDER BY` inside the window decides *which* duplicate survives. Here the two copies were identical so it did not matter. In real data the copies often differ, and you must choose a rule such as "keep the latest by modified date" or "keep the one with the most filled-in columns".

`DEDUPLICATION RESULT (BATCH 1)`

	rows for customer 1002	total rows
Before	2	4
After	1	3

3.21.5 Lookups

Idea. A *lookup* enriches a row by finding a related value in a reference table. Order items carry only `product_code`. A lookup in PRODUCTS adds the product name.

```
SELECT i.item_id, i.product_code,
       COALESCE(p.product_name, 'UNKNOWN PRODUCT') AS product_name
FROM   stg_order_items i
LEFT JOIN products p ON p.product_code = i.product_code;
```

PRODUCT LOOKUP, WITH ONE MISS

item_id	product_code	product_name (after lookup)
1	P-100	Laptop Stand
4	P-400	Desk Lamp
8 (run 2)	P-999	UNKNOWN PRODUCT

Choose deliberately what a miss does: use a default such as "UNKNOWN PRODUCT" and report it, or reject the row. Never let it vanish. Real ETL tools provide a ready-made *lookup transformation* for this and handle caching, because looking up a million rows one at a time would be slow. In the Informatica guide, lookup transformations appear among the restrictions on mapplets used for Hadoop masking (User Guide 10.5.10, p.137), which confirms that lookup is a standard kind of transformation in that product.

3.21.6 Filtering, validating and rejecting

Idea. Some rows should not reach the target. *Filtering* removes rows that are valid but unwanted (for example, test orders). *Validating* checks rows against rules and *rejects* those that fail, keeping them aside with a reason so someone can fix them.

```
CREATE TABLE reject_orders (batch_id INTEGER, order_id TEXT, reason TEXT);

INSERT INTO reject_orders
SELECT 1, o.order_id,
       CASE WHEN o.total_amount = '' OR o.total_amount GLOB '*[^0-9.]*'
            THEN 'total_amount is not a number'
            ELSE 'customer ' || o.customer_id || ' not found' END
FROM   stg_orders o
WHERE  o.total_amount = '' OR o.total_amount GLOB '*[^0-9.]*'
   OR  NOT EXISTS (SELECT 1 FROM dim_customer d
                   WHERE d.customer_id = CAST(o.customer_id AS INTEGER));
```

REJECT_ORDERS AFTER RUN 1

batch_id	order_id	reason
1	5004	total_amount is not a number
1	5005	customer 9999 not found

The valid orders carry on. This is the key design choice: **one bad row must not stop the whole run, and must not vanish**. It goes to a reject table with a batch id and a human-readable reason.

3.21.7 Joining and derived columns

Joining combines rows from different inputs. To build the sales fact table we join order items to their orders for date, customer and status. Ask of every join: "what happens to a row with no match?" An inner join drops it; a left join keeps it with empty columns. That is a business decision, not a technical one. **Derived columns** calculate values that do not exist in the source, such as `line_amount` as quantity times unit price (item 2: 2 × 100.00 = 200.0).

3.21.8 Aggregating

Idea. Summarise many rows into fewer, such as a daily sales table for dashboards.

```
SELECT order_date,
       COUNT(DISTINCT order_id) AS orders_count,
       SUM(quantity)            AS units,
       SUM(line_amount)         AS revenue
FROM   fact_order_line
GROUP BY order_date;
```

`AGG_DAILY_SALES AFTER RUN 1`

order_date	orders_count	units	revenue
2026-10-20	2	4	1749.5
2026-10-21	1	1	75.0

Hand check: on 20 October, 1 + 2 + 1 = 4 units and 1499 + 200 + 50.5 = 1749.5 revenue.

3.22 Load: writing into the target

Concept. Loading writes the transformed rows into the target tables, in a way that is correct, repeatable and fast enough.

Why it matters. A load that half-finishes leaves the target in a state nobody understands: some rows from tonight, some from yesterday. Loads must either complete or leave things as they were, and they must be safe to run again.

3.22.1 Load strategies

Strategy	What it does	Good for	Danger
Truncate and load (full refresh)	Empty the target table, then insert everything	Small tables, reference data, test databases	Target is empty or partial during the load; slow for big tables
Append (insert only)	Add new rows, never touch old ones	Event or fact data that never changes	Running twice duplicates rows
Incremental	Load only the rows extracted incrementally	Large tables	Depends on a correct watermark
Upsert (merge)	Update rows that exist, insert those that do not	Dimensions and master data	Needs a reliable key to match on
History-keeping load (SCD)	Close the old version of a row and insert a new one	Dimensions where history matters	More complex; the table grows

What the tools say about these ideas, all in their own terms:

- Informatica's plan has update strategy options: treat source rows as *Insert* or *Update*, and an "Update as" choice of *Update*, *Insert* or *Update else Insert*, which updates a row if it exists in the target and inserts the rest (User Guide 10.5.10, p.284). That is the upsert strategy, named.
- Informatica lets you run SQL before and after writing to a target. Its example is `TRUNCATE TABLE $OWNERNAME.$TABLENAME;`, which deletes all rows from the target table first (pp.277 to 278). That is truncate and load.
- Delphix on-the-fly masking jobs have a *Truncate* option that sets whether target database tables are emptied before the masked data is loaded, and a *Batch Update* option for loading in batches sized by the commit size. Create, view, edit, delete, verify jobs ↗
- GenRocket's FAQ says that for very large volumes you do not want to connect through JDBC; you generate data in the database's native bulk load format using bulk load receivers.

GenRocket General FAQ ↗

3.22.2 Upsert in SQL

Suppose Asha Menon's zip code changes in the next feed. A plain insert would collide with the existing row. An upsert updates it:

```
-- SQLite and PostgreSQL syntax
INSERT INTO dim_customer_simple (customer_id, full_name, email, zip_code)
VALUES (1001, 'Asha Menon', 'asha.menon@example.com', '560002')
ON CONFLICT (customer_id)
DO UPDATE SET full_name = excluded.full_name,
              email     = excluded.email,
              zip_code  = excluded.zip_code;

-- Oracle and SQL Server use MERGE instead
```

UPSERT EFFECT ON ONE ROW

	customer_id	full_name	zip_code
Before	1001	Asha Menon	560001
After	1001	Asha Menon	560002

Notice the property that makes upserts valuable: **running the same load twice gives the same result**. A property like this is called *idempotence*, and it is what lets you safely re-run a failed job.

3.22.3 Keeping history: a slowly changing dimension

An upsert overwrites. After it, the warehouse says Asha always lived in 560002, so a report on October 20 sales would attribute them to the wrong zip. If history matters, keep both versions with validity dates. This pattern is commonly called a **Type 2 slowly changing dimension**.

The pattern needs two statements, run in Lab 2: first *close* the current row of any customer whose data changed (set `valid_to` and `is_current = 0`), then *insert* a new current row for every customer who has none. The table gets a *surrogate key* (`customer_key`), an artificial id, because the business key `customer_id` now appears more than once.

DIM_CUSTOMER AFTER RUN 2 (OUTPUT FROM THE SQLITE RUN, 1002 AND 1003 OMITTED)

customer_key	customer_id	zip_code	valid_from	valid_to	is_current
1	1001	560001	2026-10-23	2026-10-24	0
4	1001	560002	2026-10-24	9999-12-31	1

A fact row can then be joined to the customer version valid on the order date. The date 9999-12-31 is a common convention for "still current".

3.22.4 Making a load safe

- **Transactions and commit size.** Rows are written in groups; each group is committed. Informatica's plan lets you set a source-based or target-based commit (target-based is the default) and a commit interval whose default is 10,000 rows (User Guide 10.5.10, p.285). Delphix masking jobs have a *Commit Size* field with the same meaning.
- **Recovery.** Decide in advance what a failed load does: roll back and stop, skip the bad row, or resume from a checkpoint (see 8.13).

3.23 Batch, streaming, and ETL versus ELT

3.23.1 Batch versus streaming at a glance

Batch collects data for a period (an hour, a night) and processes it as one group. It is easy to reason about and to re-run, and results are always "as of the last run". Most ETL, and nearly all test data provisioning, is batch. **Streaming** processes each event within seconds of arrival. Results are fresh, but you must handle duplicates and out-of-order events, so it is used where the answer must be current, such as fraud alerts. A middle path, *micro-batching*, runs a small batch every minute or so.

3.23.2 ETL versus ELT

In **ETL**, transformations run in the ETL engine, on its own servers, before data reaches the target. In **ELT**, raw data is loaded into the target first (often a powerful cloud warehouse) and transformed there using its SQL engine.

	ETL	ELT
Where transforms run	Separate ETL engine	Inside the target database
Strength	Sensitive data can be cleaned or masked before it lands in the target	Uses the target's power; raw data can be re-transformed later
Weakness	The engine can become a bottleneck	Raw (maybe sensitive) data sits in the target

Look at the last row, because it matters for test data. If raw data holds real personal details, ELT would place them in the target before masking. That is why test data work usually follows the ETL order: mask first, then load. The ShopSphere example in this section is really a hybrid: staging is a lightweight "EL" step and the transformation is done by SQL. Many practical systems mix both styles.

3.24 ETL architecture

An ETL tool is more than a place to run SQL. Look back at the diagram in 8.4. Here is each part and what a real product adds to the plain SQL of our example.

Part	What it does	In our SQL example
Sources	The systems and files that hold the original data	The customer, order and item feeds
Integration (ETL) engine	The program that actually reads, transforms and writes rows	The SQLite engine running our statements
Metadata repository	A database that stores the definitions of mappings, connections and run history	The SQL script files and the `etl_control` table
Scheduler	Starts jobs at the right time or after another job finishes	You, running the script by hand
Monitoring and logging	Shows progress, row counts, rejects, errors; sends alerts	The reject table and the reconciliation query
Targets	The systems that receive the result	dim_customer, fact_order_line, agg_daily_sales

3.24.1 A documented example: the pieces in Informatica TDM

Informatica TDM's architecture section lists these parts, and they line up with the general model. All quoted from the User Guide chapter 1, pages 25 to 27 and 29.

- The **PowerCenter Integration Service** is the engine. It "runs sessions and workflows that you run from Test Data Manager" (p.26) and runs the data subset, data generation and data masking workflows (p.27).
- The **PowerCenter Repository Service** manages the PowerCenter repository, the metadata store. That repository holds "information required to extract, transform, and load data", along with permissions and privileges for users (p.26).
- The **PowerCenter Client** is where you "design mappings, mapplets, and sessions to load the data" (p.25).
- The **Informatica Administrator** is a web application to "manage, monitor, deploy, and undeploy data flows" (p.25).

PowerCenter, an ETL product, is the engine under TDM. The next chapter builds on that link.

3.25 Mappings, sessions, workflows and jobs

Every ETL tool has a few layers of "thing you can define and run". Names vary, so learn the layers and then the names.

1. **Transformation.**
   One operation on rows: filter, lookup, aggregate. The smallest piece.

2. **Mapping.**
   A diagram or definition that connects a source to a target through a chain of transformations. It describes *what* happens to the data, but does not run by itself.

3. **Session (or task).**
   A runnable instance of one mapping, with the run settings: which connections, commit size, error rules.

4. **Workflow (or job, or pipeline).**
   A set of sessions and other steps with an order and conditions: "run session A; if it works, run B and C; if it fails, send an email".

5. **Schedule or trigger.**
   When the workflow starts: a time, a file arriving, another workflow ending.

THESE LAYERS IN THE INFORMATICA TDM GUIDE

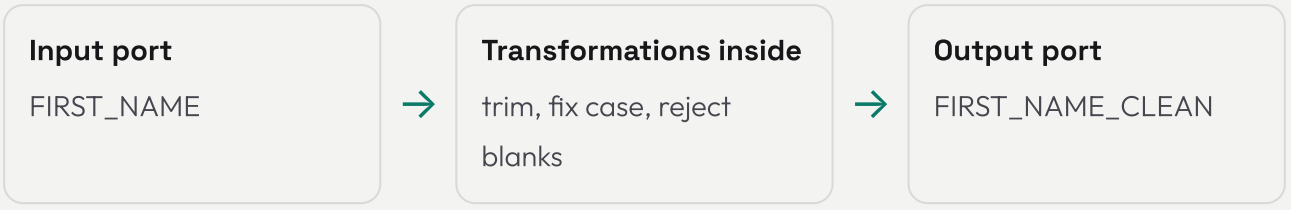
- When you generate a workflow from a plan, "the PowerCenter Repository Service generates mappings for the workflow"; when you start it, the PowerCenter Integration Service does the subset, masking and generation tasks defined in the plan (p.308).
- You can view the sessions that run from each workflow, with statistics for affected rows, rejected rows and rows written to the target (p.275, "Session Details" tab).
- A plan can run a chosen workflow before ("pre workflow") and after ("post workflow") the main one: its example copies files from production to test area first, then masks, then runs statistics on the tables (p.277).
- The guide also warns that if you edit components generated from Test Data Manager in the PowerCenter Client, "the session might generate unexpected results" (p.150).

3.26 Reusable components and the mapplet

Concept. A reusable component is a piece of ETL logic you build once and use in many places, like a function in programming.

Why it matters. Without reuse, the same rule (clean a phone number, standardise a country name) is copied into twenty mappings. When the rule changes, someone must find and fix twenty copies, and will miss one.

In Informatica's world, the reusable group of transformations is called a *mapplet*. A mapplet has input and output ports (named connection points), contains one or more transformations, and can be placed into many mappings. Think of it as a small mapping with sockets at each end. The LearningMart video "How to create mapplet in informatica" defines a mapplet the same way: a reusable object created in the Mapplet Designer, for example one that joins first and last name into a full name for many mappings. A step-by-step build is in the mapplet walkthrough in the Informatica masking chapter.



What the TDM guide says about mapplets (the rest of the product is not needed to follow this):

- Test Data Manager can create a masking rule *from a mapplet*. The mapplet holds the logic that masks the input columns and returns data to the target columns (User Guide 10.5.10, p.137).
- You import the mapplet from an XML file exported from the PowerCenter repository or the Model repository, and the mapplet can contain "any passive transformations" (p.137). A passive transformation is one that does not change the number of rows.
- A mapplet can have multiple input and multiple output columns; at least one input and one output must be marked required. When you assign the rule to a column you map the mapplet's column names to the real table columns (p.137).
- The TDM repository stores the mapplet logic when imported, and you cannot change the mapplet inside Test Data Manager. The guide also says that if you change the name or type of a field of a custom masking rule, you must import the mapplet again (p.144).
- Data generation uses the same idea: a *custom generation rule* is based on PowerCenter mapplet logic, for example one that joins first and last name into a full name (pp.196 and 211).

The Informatica data masking chapter builds a mapplet.

3.27 Error handling

Concept. Error handling is the plan for what the job does when something goes wrong, decided *before* it goes wrong.

Why it matters. Something will go wrong. A file will arrive empty, a database will be down for maintenance, a date will say 31 February. The question is whether you find out from an alert at 2 a.m. with a clear row count, or from an angry tester at 10 a.m. with a half-loaded database.

3.27.1 Three kinds of error

Kind	Example	Usual response
Data error (one row is bad)	Amount "abc"; customer 9999 does not exist	Reject the row to an error table; carry on
Process error (the whole run cannot continue)	Target database down; source file missing; disk full	Stop, roll back, alert, retry later
Logic error (runs fine but is wrong)	Join duplicates rows so revenue doubles	Found only by checks and reconciliation (8.14)

3.27.2 Strategies

- **Reject and continue** for data errors, as in 8.7.6. Each reject row has a batch id and a reason.
- **Threshold stop.** Continue unless more than N errors occur. A file where 40% of rows fail is a broken file, not 40% bad customers.
- **All-or-nothing.** Load inside a transaction so a failure undoes everything.
- **Checkpoint and resume.** Remember how far you got, and on restart continue from there.
- **Retry with a limit.** Temporary glitches (a dropped connection) often pass on a second attempt.

REAL SETTINGS IN THE INFORMATICA TDM PLAN

Under "Error and Recovery Options" in a plan (User Guide 10.5.10, p.285) the guide lists these, which are a concrete menu of the strategies above.

- **Stop on Error:** how many non-fatal errors the session tolerates before it stops. Zero means the session never stops for non-fatal errors; zero is the default.
- **Suspend on Error:** pause the workflow when an error occurs so you can fix it and restart; default No.
- **Rollback Transactions on Error:** roll back at the next commit point on a non-fatal write error; default Yes.
- **Recover Strategy:** resume from the last checkpoint, restart the task, or fail the task and continue the workflow (the default).
- **Enable High Availability Recovery:** default No.
- **Commit type and interval:** target-based commit by default; interval default 10,000 rows.

Delphix gives a second example of the same ideas for masking jobs. Data that does not fit what an algorithm expects (non-conformant data) can, at job level, make the job fail, leave the value unmasked, or set it to null (jobs page; terminology page). It also warns that when a masking job fails, the target database or files may be left in a partially masked or unmasked state (job monitoring page). Job monitoring ↗ That sentence is a good reminder: after a failed job, never assume the target is usable.

3.27.3 Reprocessing rejected rows

Run 1 rejected orders 5004 and 5005, and the watermark moved past them, so nothing picks them up again. Rejected rows need their own path: group the reject table by reason, have the owner fix the cause (the "abc" amount; whether customer 9999 arrives tomorrow), then re-stage the corrected rows as a new batch and run transform and load again.

IDEMPOTENCE AND RESTARTS

A safe job can be run twice with no harm. Achieve this with upserts, with "delete the target rows of this batch, then insert them" loads, and with watermarks moved last. Without it, every failed run needs a careful human cleanup, and humans make mistakes at 2 a.m. In the SQL here, the fact load used `INSERT OR REPLACE` keyed on `item_id` for this reason (other databases: `MERGE` or delete-then-insert).

3.28 Data quality and reconciliation

Concept. Data quality checks are tests on the data itself, run as part of the job, that prove the result is what you expected.

Why it matters. The most dangerous failure is a job that "succeeds" and loads the wrong thing. Only checks find logic errors.

Quality dimension	Question	Example check
Completeness	Is anything missing?	Rows read = rows loaded + rows rejected
Validity	Do values follow the rules?	Amounts are numeric and not negative; dates parse
Uniqueness	Is anything duplicated?	One current row per customer; no repeated item_id
Consistency	Do related values agree?	Every fact row has a customer; line amounts add up to order totals
Timeliness	Is it fresh enough?	Latest order date in the target is yesterday or later

3.28.1 Reconciliation queries

Reconciliation means comparing counts and totals between two stages to show nothing was lost or invented. These are the queries to run after run 2 of the ShopSphere job.

```
-- Completeness: orders read = orders loaded + orders rejected
SELECT (SELECT COUNT(*) FROM feed_orders)                      AS
read_total,
       (SELECT COUNT(DISTINCT order_id) FROM fact_order_line)  AS
loaded_orders,
       (SELECT COUNT(DISTINCT order_id) FROM reject_orders)    AS
rejected_orders;

-- Consistency: no fact row without a customer version (expect no rows)
SELECT DISTINCT f.customer_id
FROM   fact_order_line f
LEFT JOIN dim_customer d ON d.customer_id = f.customer_id
WHERE  d.customer_id IS NULL;
```

Expected result: Expected result: the first query returns 7, 5 and 2 (7 = 5 + 2); the second returns no rows.

> **KEEP THE CHECKS WITH THE JOB**
>
> Write checks once and run them every time, with results stored in a table. Then the question "was last Tuesday's load complete?" has an answer. The Informatica guide shows the same idea at the product level: TDM can be used with the Data Validation Option so that target data can be validated and compared against production to show the transformation process did not introduce errors (User Guide 10.5.10, p.24).

3.29 Performance ideas

Performance tuning is its own craft, and the right fix depends on measurement. These are the ideas that help most often, in the order to try them.

1. **Think in sets, not rows.**
   One SQL statement that updates a million rows is much faster than a program that updates them one by one. Many ETL tools give you a row-by-row feeling in their diagrams, but good ones process rows in streams and batches.

2. **Use bulk loading for big loads.**
   See 8.8.1 for the documented examples from GenRocket's FAQ.

3. **Tune commit size.**

A commit after every row is slow; a commit after 50 million rows risks a huge rollback. Informatica's default of 10,000 rows is a hint at the ballpark, not a rule.

4 **Handle indexes and constraints during the load.**
Maintaining an index for every inserted row costs time. Some loads drop indexes first and rebuild afterwards. Delphix exposes this as the "Drop Indexes" job option and warns that multiple update threads can cause deadlocks, advising fewer threads if that happens (jobs page).

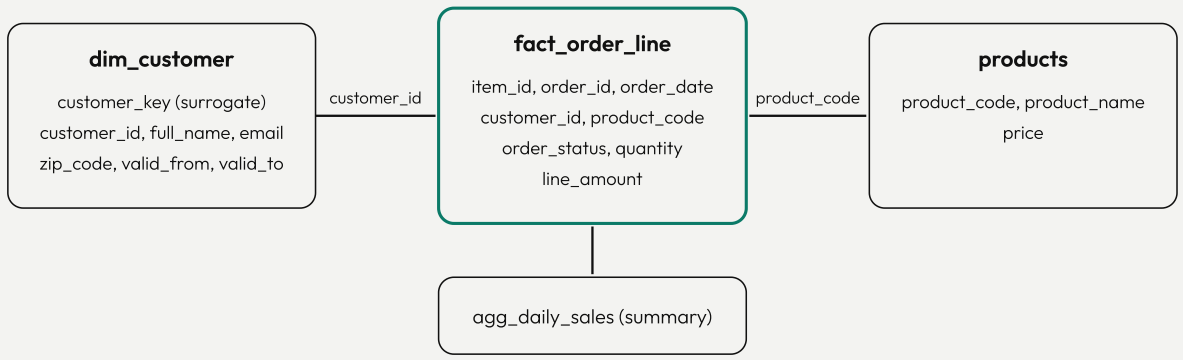
5 **Run in parallel.**
Independent tables can load at the same time. Delphix jobs have "Number of Streams", described as the number of parallel streams, so several tables are processed at once; some streams finish before others, so the number of active streams falls during the job (jobs page). Informatica allows up to eight concurrent workflows (p.276).

3.30 Common ETL tools

Informatica PowerCenter is the ETL product under Informatica TDM, and the TDM notes cover how TDM uses it (8.10 and 8.11). You will also meet Microsoft SQL Server Integration Services, Talend, cloud services such as AWS Glue and Azure Data Factory, and schedulers such as Apache Airflow. Nothing here goes beyond the general ideas of this section. Plain SQL scripts plus a scheduler such as cron, as in this section, are a real way to do small ETL. TDM tools are ETL-style tools specialised for producing safe test data; the next chapter compares the two.

3.31 The complete ShopSphere ETL job on one page

Sections 3.19 to 8.8 showed the pieces. Here is how they fit into one job, the way you would document it for a colleague. The target is a small star schema for sales reporting.



Original diagram: the target star schema. The fact table holds one row per order line; the dimensions describe who and what; the summary table is built from the fact table.

3.31.1 What the two runs did

These row counts come from two SQLite runs of the full job. Run 1 is the first load; run 2 is the next day, after Asha's zip code changed and two new orders (5006 and 5007) arrived.

RUN LOG (COUNTS AT EACH STEP)

Step	Run 1	Run 2
Watermark before	0	5005
Extracted: customers / orders / items	4 / 5 / 6	3 / 2 / 2
Customers after dedup	3	3
Customer versions closed / inserted	0 / 3	1 / 1
Orders rejected	2 (5004, 5005)	0
Fact rows loaded	4	2
Daily summary rows	2	3
Watermark after	5005	5007

Order 5007 has an unknown product (P-999) and an unmapped status (Z); it still loads, as "UNKNOWN PRODUCT" and "Unknown", because the lookups use left joins. The 2026-10-23 summary row (2 orders, 3 units, 220.0) includes that line. Whether unknown products count as revenue is a decision to raise, not to bury.

3.32 Hands-on labs

You need any SQL database. SQLite is free and has no setup: install the SQLite command-line tool or use an online SQLite playground, open a new empty in-memory database, and paste each block. If you use another database, adjust the function names as noted in section 3.21.

Lab 1: ETL by hand with SQL

Goal: run extract-staged data through clean, deduplicate, reject, convert, look up, derive, aggregate and reconcile, and compare the results with the worked answer. The staged data is the ShopSphere batch 1 from section 3.20.

1

Create the staging and target tables.

Staging holds the raw text; the targets are empty until the transform runs.

```
CREATE TABLE stg_customers (customer_id TEXT, first_name
TEXT, last_name TEXT, email TEXT, zip_code TEXT);
INSERT INTO stg_customers VALUES
 ('1001', ' Asha ', 'MENON', 'Asha.Menon@Example.com ',
'560001'),
 ('1002', 'Ravi',   'Kumar', 'ravi.k@example.com',
'600017'),
 ('1002', 'Ravi',   'Kumar', 'ravi.k@example.com',
'600017'),
 ('1003', 'Meera',  'Nair',  NULL,
'682001');

CREATE TABLE stg_orders (order_id TEXT, customer_id TEXT,
order_date TEXT, total_amount TEXT);
INSERT INTO stg_orders VALUES
 ('5001', '1001', '20/10/2026', '1499.00'),
 ('5002', '1002', '20/10/2026', '250.50'),
 ('5003', '1001', '21/10/2026', '75.00'),
 ('5004', '1003', '21/10/2026', 'abc'),
 ('5005', '9999', '22/10/2026', '303.00');

CREATE TABLE stg_order_items (item_id TEXT, order_id TEXT,
product_code TEXT, quantity TEXT, unit_price TEXT);
INSERT INTO stg_order_items VALUES
 ('1', '5001', 'P-100', '1', '1499.00'),
 ('2', '5002', 'P-200', '2', '100.00'),
 ('3', '5002', 'P-300', '1', '50.50'),
 ('4', '5003', 'P-400', '1', '75.00'),
 ('5', '5004', 'P-200', '1', '100.00'),
 ('6', '5005', 'P-300', '6', '50.50');

CREATE TABLE products (product_code TEXT PRIMARY KEY,
product_name TEXT, price REAL);
INSERT INTO products VALUES
 ('P-100', 'Laptop Stand', 1499.00), ('P-200', 'USB Cable',
100.00),
 ('P-300', 'Phone Case', 50.50),     ('P-400', 'Desk Lamp',
75.00);

CREATE TABLE dim_customer (customer_id INTEGER PRIMARY KEY,
full_name TEXT, email TEXT, zip_code TEXT);
CREATE TABLE reject_orders (order_id TEXT, reason TEXT);
CREATE TABLE fact_order_line (item_id INTEGER PRIMARY KEY,
order_id INTEGER, order_date TEXT,
  customer_id INTEGER, product_code TEXT, product_name TEXT,
```

```
quantity INTEGER, line_amount REAL);
CREATE TABLE agg_daily_sales (order_date TEXT, orders_count
INTEGER, units INTEGER, revenue REAL);
```

Expected result: No output. Check with `SELECT COUNT(*) FROM stg_customers;` , which should return 4.

2 **Clean, deduplicate and load customers.**

Trimming and case fixes make equal things equal; `ROW_NUMBER()` keeps one row for customer 1002.

Expected result: The customer table has three rows: Asha Menon, Ravi Kumar, and Meera Nair with email "not provided".

3 **Reject bad orders.**

Bad rows are parked with a reason, not dropped.

Expected result: Two reject rows: 5004 "total_amount is not a number" and 5005 "customer 9999 not found".

4 **Build the fact table.**

This converts text to dates and numbers, looks up the product name and derives the line amount.

Expected result: Four fact rows (items 1 to 4). Item 2 has line_amount 200.0 (2 times 100.00).

5 **Aggregate, then reconcile.**

The summary shows revenue per day, and the reconciliation proves nothing was lost.

Expected result: Revenue 1749.5 for 2026-10-20 and 75.0 for 2026-10-21. Reconciliation returns 5, 3 and 2: five orders read, three loaded, two rejected.

Run steps 2 to 5 as one script:

```
-- STEP 1: clean, deduplicate and load customers
INSERT INTO dim_customer (customer_id, full_name, email, zip_code)
SELECT CAST(customer_id AS INTEGER),
       UPPER(SUBSTR(TRIM(first_name),1,1)) ||
LOWER(SUBSTR(TRIM(first_name),2)) || ' ' ||
       UPPER(SUBSTR(TRIM(last_name),1,1))  ||
LOWER(SUBSTR(TRIM(last_name),2)),
       COALESCE(LOWER(TRIM(email)), 'not provided'),
       zip_code
FROM (SELECT s.*, ROW_NUMBER() OVER (PARTITION BY customer_id ORDER
BY customer_id) AS rn
      FROM stg_customers s)
WHERE rn = 1;
SELECT * FROM dim_customer;
-- STEP 2: reject bad orders with a reason
INSERT INTO reject_orders
SELECT o.order_id,
       CASE WHEN o.total_amount = '' OR o.total_amount GLOB '*[^0-
9.]*'
            THEN 'total_amount is not a number'
            ELSE 'customer ' || o.customer_id || ' not found' END
FROM stg_orders o
WHERE o.total_amount = '' OR o.total_amount GLOB '*[^0-9.]*'
   OR NOT EXISTS (SELECT 1 FROM dim_customer d WHERE d.customer_id =
CAST(o.customer_id AS INTEGER));
SELECT * FROM reject_orders;
-- STEP 3: build the fact table (convert, look up, derive)
INSERT INTO fact_order_line
SELECT CAST(i.item_id AS INTEGER), CAST(o.order_id AS INTEGER),
       SUBSTR(o.order_date,7,4) || '-' || SUBSTR(o.order_date,4,2) ||
'-' || SUBSTR(o.order_date,1,2),
       CAST(o.customer_id AS INTEGER), i.product_code,
p.product_name,
       CAST(i.quantity AS INTEGER), CAST(i.quantity AS INTEGER) *
CAST(i.unit_price AS REAL)
FROM stg_order_items i
JOIN stg_orders o ON o.order_id = i.order_id
JOIN products p   ON p.product_code = i.product_code
WHERE o.order_id NOT IN (SELECT order_id FROM reject_orders);
SELECT * FROM fact_order_line;
-- STEP 4: aggregate
INSERT INTO agg_daily_sales
SELECT order_date, COUNT(DISTINCT order_id), SUM(quantity),
SUM(line_amount)
FROM fact_order_line GROUP BY order_date;
```

```
SELECT * FROM agg_daily_sales ORDER BY order_date;
-- STEP 5: reconcile
SELECT (SELECT COUNT(*) FROM stg_orders) AS read_rows,
       (SELECT COUNT(DISTINCT order_id) FROM fact_order_line) AS
loaded,
       (SELECT COUNT(*) FROM reject_orders) AS rejected;
```

▶ **Show a worked answer**

Lab 2: the next day, with a watermark and history

Goal: run day 2 of the job incrementally, keep Asha's old zip code as history, survive an unknown product and a code with no mapping, and prove that a re-run changes nothing. Use a fresh empty database. This script builds the state at the end of day 1 and day 2's incoming feeds.

1. **Create the end-of-day-1 state and the day 2 feeds.**
   You start with warehouse tables already holding day 1, a watermark of 5005, and the new feeds, where Asha's zip is 560002 and orders 5006 and 5007 are new.

```
-- State at the end of run 1 (day 1)
CREATE TABLE dim_customer (customer_key INTEGER PRIMARY KEY
AUTOINCREMENT, customer_id INTEGER,
  full_name TEXT, email TEXT, zip_code TEXT, valid_from
TEXT, valid_to TEXT, is_current INTEGER);
INSERT INTO dim_customer (customer_id, full_name, email,
zip_code, valid_from, valid_to, is_current) VALUES
 (1001, 'Asha Menon',  'asha.menon@example.com', '560001',
'2026-10-23', '9999-12-31', 1),
 (1002, 'Ravi Kumar',  'ravi.k@example.com',     '600017',
'2026-10-23', '9999-12-31', 1),
 (1003, 'Meera Nair',  'not provided',           '682001',
'2026-10-23', '9999-12-31', 1);

CREATE TABLE fact_order_line (item_id INTEGER PRIMARY KEY,
order_id INTEGER, order_date TEXT,
  customer_id INTEGER, product_code TEXT, product_name TEXT,
quantity INTEGER, line_amount REAL);
INSERT INTO fact_order_line VALUES
 (1, 5001, '2026-10-20', 1001, 'P-100', 'Laptop Stand', 1,
1499.0),
 (2, 5002, '2026-10-20', 1002, 'P-200', 'USB Cable',    2,
200.0),
 (3, 5002, '2026-10-20', 1002, 'P-300', 'Phone Case',   1,
50.5),
 (4, 5003, '2026-10-21', 1001, 'P-400', 'Desk Lamp',    1,
75.0);

CREATE TABLE etl_control (job_name TEXT PRIMARY KEY,
last_order_id INTEGER);
INSERT INTO etl_control VALUES ('orders_daily', 5005);

CREATE TABLE products (product_code TEXT PRIMARY KEY,
product_name TEXT, price REAL);
INSERT INTO products VALUES
 ('P-100', 'Laptop Stand', 1499.00), ('P-200', 'USB Cable',
100.00),
 ('P-300', 'Phone Case', 50.50),     ('P-400', 'Desk Lamp',
75.00);
CREATE TABLE status_map (status_code TEXT PRIMARY KEY,
status_name TEXT);
INSERT INTO status_map VALUES ('S','Shipped'),
('P','Pending'), ('C','Cancelled');

-- Day 2 feeds (text, as the source system writes them)
```

```
CREATE TABLE feed_customers (customer_id TEXT, first_name
TEXT, last_name TEXT, email TEXT, zip_code TEXT);
INSERT INTO feed_customers VALUES
 ('1001', 'Asha',  'Menon', 'asha.menon@example.com',
'560002'),
 ('1002', 'Ravi',  'Kumar', 'ravi.k@example.com',
'600017'),
 ('1003', 'Meera', 'Nair',  NULL,
'682001');
CREATE TABLE feed_orders (order_id TEXT, customer_id TEXT,
order_date TEXT, total_amount TEXT, status_code TEXT);
INSERT INTO feed_orders VALUES
 ('5001','1001','20/10/2026','1499.00','S'),
('5002','1002','20/10/2026','250.50','S'),
 ('5003','1001','21/10/2026','75.00','P'),
('5004','1003','21/10/2026','abc','S'),
 ('5005','9999','22/10/2026','303.00','S'),
 ('5006','1002','23/10/2026','100.00','C'),
('5007','1001','23/10/2026','120.00','Z');
CREATE TABLE feed_order_items (item_id TEXT, order_id TEXT,
product_code TEXT, quantity TEXT, unit_price TEXT);
INSERT INTO feed_order_items VALUES
 ('1','5001','P-100','1','1499.00'), ('2','5002','P-
200','2','100.00'), ('3','5002','P-300','1','50.50'),
 ('4','5003','P-400','1','75.00'),   ('5','5004','P-
200','1','100.00'), ('6','5005','P-300','6','50.50'),
 ('7','5006','P-200','1','100.00'),  ('8','5007','P-
999','2','60.00');

CREATE TABLE stg_customers (customer_id TEXT, first_name
TEXT, last_name TEXT, email TEXT, zip_code TEXT);
CREATE TABLE stg_orders (order_id TEXT, customer_id TEXT,
order_date TEXT, total_amount TEXT, status_code TEXT);
CREATE TABLE stg_order_items (item_id TEXT, order_id TEXT,
product_code TEXT, quantity TEXT, unit_price TEXT);
```

Expected result: No output. `SELECT * FROM etl_control;` returns orders_daily with 5005.

2

Extract past the watermark.

Only orders with an id above 5005 should reach staging. Customers are small, so they are copied in full.

Expected result: Staging holds orders 5006 and 5007.

3 **Close the changed customer, then insert the new version.**

Two statements keep history instead of overwriting it.

Expected result: Customer 1001 has two rows. Key 1: zip 560001, valid until 2026-10-24, is_current 0. Key 4: zip 560002, valid from 2026-10-24 to 9999-12-31, is_current 1.

4 **Load the new fact rows.**

Left join keeps order 5007 even though product P-999 is unknown.

Expected result: Two new rows: USB Cable with 100.0, and UNKNOWN PRODUCT with 120.0.

5 **Move the watermark last, then re-run the extract.**

If the extract finds nothing, the job is safe to repeat.

Expected result: The watermark reads 5007, and the re-run extracts 0 rows.

Run steps 2 to 5 as one script:

```
-- STEP 1: extract. Customers: full copy. Orders and items: only
beyond the watermark.
INSERT INTO stg_customers SELECT * FROM feed_customers;
INSERT INTO stg_orders
SELECT * FROM feed_orders
WHERE CAST(order_id AS INTEGER) > (SELECT last_order_id FROM
etl_control WHERE job_name = 'orders_daily');
INSERT INTO stg_order_items
SELECT * FROM feed_order_items WHERE order_id IN (SELECT order_id
FROM stg_orders);
SELECT order_id FROM stg_orders;
-- STEP 2: close the old version of any customer whose data changed
UPDATE dim_customer
SET valid_to = '2026-10-24', is_current = 0
WHERE is_current = 1
  AND EXISTS (SELECT 1 FROM stg_customers s
              WHERE CAST(s.customer_id AS INTEGER) =
dim_customer.customer_id
                AND (s.zip_code <> dim_customer.zip_code
                  OR COALESCE(LOWER(TRIM(s.email)),'not provided') <>
dim_customer.email));
-- STEP 3: insert a current version for every customer who has none
INSERT INTO dim_customer (customer_id, full_name, email, zip_code,
valid_from, valid_to, is_current)
SELECT CAST(s.customer_id AS INTEGER),
       UPPER(SUBSTR(TRIM(s.first_name),1,1)) ||
LOWER(SUBSTR(TRIM(s.first_name),2)) || ' ' ||
       UPPER(SUBSTR(TRIM(s.last_name),1,1))  ||
LOWER(SUBSTR(TRIM(s.last_name),2)),
       COALESCE(LOWER(TRIM(s.email)), 'not provided'), s.zip_code,
       '2026-10-24', '9999-12-31', 1
FROM stg_customers s
WHERE NOT EXISTS (SELECT 1 FROM dim_customer d
                  WHERE d.customer_id = CAST(s.customer_id AS
INTEGER) AND d.is_current = 1);
SELECT customer_key, customer_id, zip_code, valid_from, valid_to,
is_current
FROM dim_customer ORDER BY customer_id, customer_key;
-- STEP 4: load new fact rows, with lookups that do not lose rows
INSERT INTO fact_order_line
SELECT CAST(i.item_id AS INTEGER), CAST(o.order_id AS INTEGER),
       SUBSTR(o.order_date,7,4) || '-' || SUBSTR(o.order_date,4,2) ||
'-' || SUBSTR(o.order_date,1,2),
       CAST(o.customer_id AS INTEGER), i.product_code,
       COALESCE(p.product_name, 'UNKNOWN PRODUCT'),
```

```
       CAST(i.quantity AS INTEGER), CAST(i.quantity AS INTEGER) *
CAST(i.unit_price AS REAL)
FROM stg_order_items i
JOIN stg_orders o ON o.order_id = i.order_id
LEFT JOIN products p ON p.product_code = i.product_code;
SELECT item_id, order_id, order_date, product_name, quantity,
line_amount
FROM fact_order_line WHERE order_id ⩾ 5006;
-- STEP 5: move the watermark, last
UPDATE etl_control
SET last_order_id = (SELECT MAX(CAST(order_id AS INTEGER)) FROM
stg_orders)
WHERE job_name = 'orders_daily';
SELECT * FROM etl_control;
-- STEP 6: run it again by mistake: the watermark makes the second
run a no-op
DELETE FROM stg_orders;
INSERT INTO stg_orders
SELECT * FROM feed_orders
WHERE CAST(order_id AS INTEGER) > (SELECT last_order_id FROM
etl_control WHERE job_name = 'orders_daily');
SELECT COUNT(*) AS rows_extracted_on_rerun FROM stg_orders;
```

▶ **Show a worked answer**

3.33 Common mistakes and gotchas

MISTAKES BEGINNERS MAKE

- **Transforming while extracting.** Business logic buried in the source query is invisible and slows the source.
- **Inner joins on lookups.** A failed match silently deletes rows. Use left joins with a default, or an explicit reject.
- **Moving the watermark before the load finishes.** A failure then loses data forever.
- **Overwriting when history matters.** Upsert is right for corrections; a history-keeping load is right when reports must show "as it was".
- **Letting one bad row stop everything, or letting bad rows disappear.** Reject with a reason, and set a threshold for when to stop.
- **No reconciliation.** A job that "succeeds" with the wrong totals is worse than one that fails loudly.
- **Forgetting that deletes are invisible to a watermark.** Plan for them, or accept and document the gap.

3.34 Recap

RECAP

- ETL moves data from sources to targets in three steps: **extract** (read), **transform** (clean, convert, enrich, combine, summarise) and **load** (write), usually through a **staging area** that keeps a raw copy.
- Extraction can be full or incremental; a **watermark** or change data capture tells an incremental run what is new.
- Transformation covers cleaning, type conversion, value mapping, deduplication, lookups, filtering and rejecting, joining, derived columns and aggregation. Lookups and joins must decide what happens when nothing matches.
- Loads can be truncate and load, append, incremental, upsert, bulk, or history-keeping (slowly changing dimension). A safe load can be run twice without harm.
- An ETL architecture has sources, an integration engine, a metadata repository, a scheduler, monitoring and logs, and targets. Tools add layers: mappings, sessions, workflows, schedules.
- A **mapplet** is Informatica's reusable group of transformations. TDM can import mapplets as masking and generation rules.
- Good ETL rejects bad rows with reasons, tolerates a set number of errors, reconciles counts, and records every run.
- ELT loads raw data first and transforms inside the target; test data work usually transforms (masks) before loading.

ETL and TDM: how they relate

ETL is covered in ETL from scratch, and what TDM has to achieve in the chapters on subsetting, masking and generation. The two join up: **a TDM job is an ETL job whose output is a safe, small, consistent copy of production for testing**. Seeing the shared skeleton makes each tool easier to understand, because you can ask of every screen: which ETL stage is this?

The two are not the same. Treating a TDM job as an ordinary ETL job loads unmasked data, breaks foreign keys or leaves no record of what was done.

3.35 Words to know first

TDM pipeline

The chain of stages that turns production (or a generator) into a usable test data set: find sensitive data, cut it down, mask it, load it, check it.

Provisioning

Delivering a prepared data set to a test environment so testers can use it.

In-place masking

Masking that changes the values inside the database you point it at. Delphix describes in-place jobs as updating the source environment with the masked values.

On-the-fly masking

Masking while copying: unmasked data is read from the source and only masked data is written to the target. This is the ETL order.

Deterministic (consistent) masking

Masking that turns the same input into the same output every time, so the same customer is masked identically in every table and every run.

Driving table

The table whose rows decide which rows are copied. Informatica uses the term for the starting point of a subset.

Audit trail

A record of who ran what, when, and with which settings, kept so that compliance questions can be answered later.

Refresh

Re-running the pipeline to replace old test data with a new copy.

Receiver

GenRocket's component that receives generated data and writes it into a usable format at a destination. It plays the role of the load stage.

Workflow (Informatica)

The runnable job that Test Data Manager generates from a plan.

3.36 The same underlying idea

Concept. Both ETL and TDM *move data and change it on the way*. Both read from a source, apply rules, and write to a target. Both are driven by definitions that tell the engine what to do, both run as repeatable jobs, and both need logs, error handling and connections to databases.

Evidence from the products. The strongest proof is in the Informatica guide itself. The PowerCenter repository, it says, stores the workflows and mappings generated from Test Data Manager and "information required to extract, transform, and load data" (User Guide 10.5.10, p.26). The workflows that perform data subset, data masking and data generation are run by the PowerCenter Integration Service (p.27). In other words, Informatica TDM *is* an ETL engine with a test-data front end.

ETL extract, transform, load	=	**TDM job** extract a subset, mask, load to test

The analogy is useful but incomplete: a photocopier that blacks out names is a good picture of masking, and a photocopier is not an ETL engine. TDM adds *purpose-built rules* (find the sensitive columns, hide them consistently, keep keys valid) on top of the same moving parts.

3.37 How they differ

The seven differences below explain why you cannot take an everyday ETL job and call it a TDM job.

Dimension	Typical ETL	TDM
Purpose	Make data usable for reporting, integration or migration	Make data safe and suitable for testing
Direction and shape	Many sources converge into a reshaped target (warehouse tables)	One source copied to one or many test environments, usually with the *same* shape
Output consistency	Consistent with business meaning (totals are right)	Relationally intact: every foreign key still matches, even after values are hidden
Privacy	Protect access to the target; data stays real	Sensitive values must not reach the target at all
Size	As much data as the business needs, often all of it	As little as the tests need (a subset), or newly generated
Repeatability	Scheduled, usually nightly; results are the latest facts	On demand or per release; results should be reproducible and resettable
Who uses it	Data engineers, BI developers	TDM engineers, QA engineers, developers, with data stewards and compliance officers as approvers

3.37.1 The differences in more detail

Shape. ETL changes shape because the target has a different purpose (a star schema, as in the previous chapter). TDM keeps shape because the application under test expects its own tables. The documents show this: GenRocket's G-Subset can only migrate between databases of the same type, for example MySQL to MySQL (G-Subset overview); Delphix on-the-fly masking supports Oracle to Oracle or delimited file to delimited file, and database to delimited file (jobs page).

Consistency. A warehouse can lose the original keys and use surrogate ones. A test database cannot, because the application joins on the real keys. Masking must keep joins working: if `customer_id` in CUSTOMERS is masked differently from `customer_id` in ORDERS, the join returns no rows. Delphix's terminology page shows how its algorithms deal with this: a secure lookup algorithm picks replacements by a cryptographic hash of the original value, "so masking output is consistent for each input"; mapping algorithms guarantee no collisions; segment mapping permutes short values with no collisions and is suggested for unique columns such as Social Security Numbers and keys. Delphix masking terminology ↗

Size. Informatica describes a data subset as a "small, targeted, and referentially intact copy" of production for nonproduction use (User Guide 10.5.10, p.112). Small matters because test environments are cheap and many; but small must still be *whole*, meaning parents and children come together. Section 3.40 shows the integrity options.

Repeatability. Informatica lets you use criteria parameters so the same workflow produces different subsets by changing values at run time (p.113). Delphix jobs can be multi-tenant, so the same rule set masks identical schemas through different connectors chosen when the job runs. GenRocket can regenerate the same data with a seed (General FAQ).

Success measure. A broken ETL job is noticed when a report is wrong. A broken TDM job may produce a data set that *looks* fine and leaks a real phone number. Its success has to be proven by checks for leakage as well as for counts, which is why section 3.41 ends with a test step.

3.38 Where they overlap

Every TDM capability is an ETL operation with a TDM twist.

TDM capability	ETL equivalent	The TDM twist
Discovery and profiling	Source profiling before building a mapping	Looks for *sensitive* data (SSN, phone) and for the relationships between tables
Subsetting	A filtered, incremental extract	Must follow foreign keys so parents and children stay together
Masking	A transformation (value replacement)	Deterministic and format-preserving; the original must not be recoverable from the output unless reversible by design
Generation	A source (the data comes from rules, not a system)	Parents before children; designed boundary, null and volume data (see the generation concepts chapter)
Provisioning and refresh	A load, with truncate-and-load or reset	Resettable to a known state; often self-service for testers
Rules, policies, rule sets	Mapping definitions in the metadata repository	Reusable by data domain ("every SSN column gets this rule")
Plan, workflow, job	Workflow and schedule	Generated from a specification, not drawn by hand
Monitor, logs, audit	Monitoring and logging	Also a compliance record

3.38.1 Masking really is a transformation

A cleaning transformation such as `LOWER(TRIM(email))` (see ETL from scratch) changes a value row by row, according to a rule. A masking rule such as `'user' || customer_id || '@example.test'` does exactly the same. The only difference is the reason: cleaning makes a value *better*; masking makes it *safe*. In the Informatica guide, masking rules that come from mapplets are literally PowerCenter transformation logic imported into TDM (User Guide

10.5.10, p.137). The same is true of generation: a custom generation rule is built from mapplet logic (p.192).

3.38.2 Subsetting really is a filtered extract

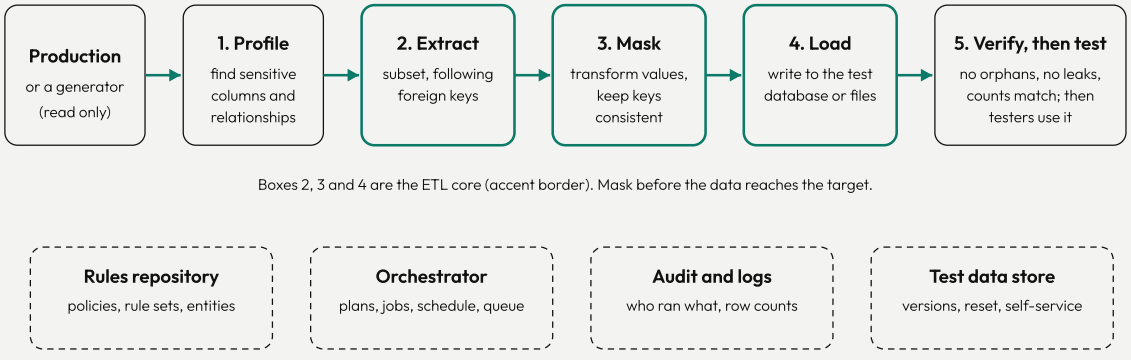
The Informatica guide defines an entity as a *driving table* plus the tables related to it through constraints. Define a filter, such as EmpID equal to 10, and the workflow loads the Employee row, its history rows and its orders into the target (pp.114 and 121). In SQL that is `WHERE` on the parent plus joins down to the children. Delphix rule sets offer the same at table level: each table can carry a *Filter*, "a WHERE clause to filter or subset the data", or a custom SQL SELECT (Managing rule sets). Managing rule sets ↗ GenRocket's G-Subset offers a "Where" clause, a percentage of rows, or a number of rows (G-Subset overview).

3.38.3 Provisioning really is a load

Loading rules from ETL apply directly: load parents before children, decide truncate versus append, use commit sizes, handle constraints. Delphix on-the-fly jobs expose a *Truncate* checkbox, *Commit Size*, *Batch Update* and *Disable Constraints*. Informatica's plan has an update strategy and a commit interval (p.284 to 285). Provisioning adds one thing ETL rarely has: a path *back*. Informatica's test data warehouse stores versions of data sets and lets you reset a target to any version (pp.236 to 237).

3.39 A reference architecture for a TDM pipeline

Every stage below is a stage of ETL; the two extra boxes at the ends (profile and test) are TDM additions.



Original diagram: a TDM pipeline built from ETL-style stages. Solid arrows are the data path; dashed boxes are the supporting services.

3.39.1 Stage by stage

1. **Profile.**
   Reason: you cannot mask what you have not found. Profiling scans columns for patterns (SSN-like, phone-like) and learns which tables relate.

   Expected result: Output: a list of sensitive columns and a relationship map, approved by a data steward. Covered in the Informatica profiling chapter and the Delphix profiling chapter.

2. **Extract the subset.**
   Reason: copying everything is slow and expands the exposure. Choose driver rows, then pull in all related rows so the subset is whole.

   Expected result: Output: a consistent slice in staging or in a stream, still containing real values.

3. **Mask (transform).**
   Reason: this is the stage that removes risk, so it must happen before data lands anywhere testers can read.

   Expected result: Output: the same rows with sensitive values replaced consistently.

4. **Load.**
   Reason: put the result where tests run, in the right order and within the target's constraints.

   Expected result: Output: a populated test database or set of files.

5. **Verify, then test.**
   Reason: nobody should trust a TDM job on its exit code. Check keys, counts and, crucially, that no original sensitive value survived.

   Expected result: Output: pass or fail evidence, stored with the run.

The supporting services mirror ETL architecture: a rules repository (metadata), an orchestrator (scheduler), logs (monitoring), plus a place to keep versions of data sets. Informatica provides the last as the test data warehouse, which stores multiple versions of test data in a central location, shareable across teams (User Guide 10.5.10, p.236).

WHY THE ORDER EXTRACT, MASK, LOAD MATTERS

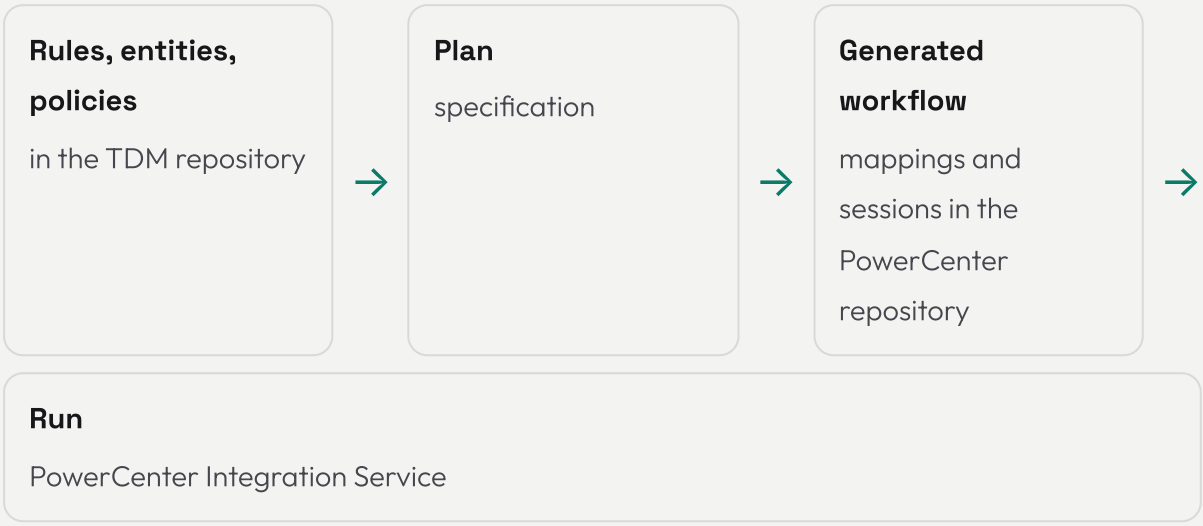
If you load first and mask afterwards (an ELT-style order), there is a time window in which real personal data sits in the test database, copies may be taken, and a failed masking job leaves the original data behind. Delphix says it plainly for in-place jobs: when a masking job fails, the target "may be in a partially masked or unmasked state" (Job monitoring), and stopping a running job results in semi-masked data (Running and Stopping Jobs). On-the-fly masking avoids the window by never writing unmasked data to the target.

3.40 How the three tools sit on this picture

Each tool packages the same stages differently. The goal here is not to learn the screens (the tool chapters do that) but to see the ETL skeleton inside each one.

3.40.1 Informatica TDM: a plan that becomes a PowerCenter workflow

Core idea. You do not draw an ETL mapping. You describe *what you want* in a plan, and TDM generates the ETL objects for you. A plan defines a data subset, masking or data generation operation and holds the components it needs (User Guide 10.5.10, p.274). When you generate a workflow, "the PowerCenter Repository Service generates mappings for the workflow"; when you start it, the PowerCenter Integration Service does the subset, masking and generation tasks (p.308). The guide lists what the Integration Service creates for masking: mappings, transformations, mapplets, sessions and workflows (p.150).



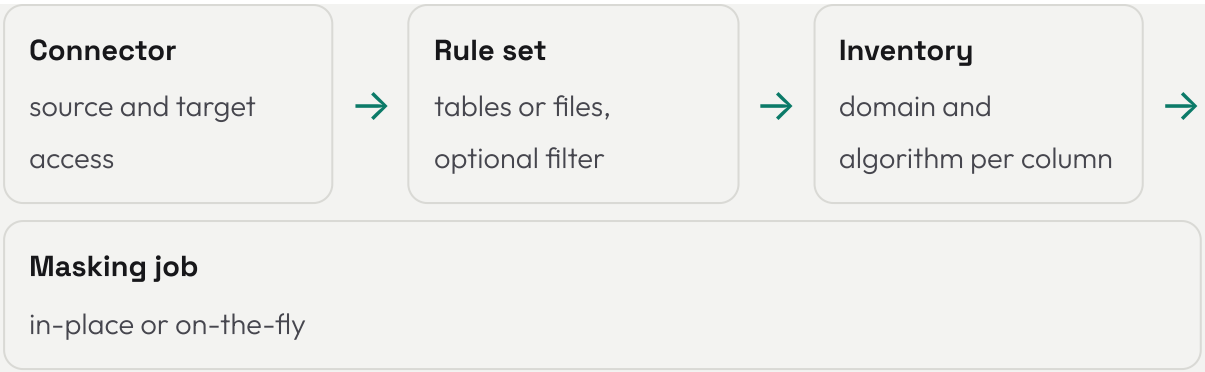
Where each pipeline stage lives:

- **Extract subset.** An *entity* has a driving table, and TDM adds the tables related to it through constraints; you attach filter parameters to columns. The workflow loads the matching subset into a target database (pp.112 to 114). Three data integrity options decide how whole the subset is: *Transactional Integrity for Major Relation Type* (a selected child pulls in its parents, the parents pull in all their children, and this repeats until no more related rows are found), *Referential Integrity Only* (the least rows needed to keep keys valid, so some children of a parent may be missing) and *Two-way* (criteria propagate down and then up until every table has some data). Recursive evaluation and Two-way need a staging connection (pp.121 to 122).
- **Staging.** A plan lets you choose source, target, staging, lookup, storage and dictionary connections (p.276). A staging connection is a database used for staging data, and cyclic tables in an entity require one (p.114).
- **Mask.** Masking rules (standard, advanced, or imported from a mapplet) are assigned to columns, data domains and policies. Subset and masking can share a plan: "the subset component gets priority", so TDM subsets first and then masks the subset, and does not mask data outside the subset (p.275). That is the extract-then-transform order of the reference architecture.
- **Load.** The target is a database, flat file, or the test data warehouse, which keeps versioned data sets that can be reset to a test system (pp.236 to 238, 275). Plan settings add update strategy (insert, update, or update else insert), commit type and interval, and error and recovery options (pp.284 to 285). A pre workflow and post workflow, and SQL before and after writing to the target, are available (pp.277 to 278).
- **Operate.** The Monitor view shows progress, and the Session Details tab shows affected, rejected and written rows (p.275). Up to eight workflows run at once (p.276).

Architecture reminder (pp.25 to 27): Test Data Manager is the web front end; the TDM repository stores rules, projects and entities; the PowerCenter Repository Service and Integration Service hold and run the generated workflows. The Informatica architecture chapter covers this in full, and the plans and workflows chapter covers the plan screens.

3.40.2 Delphix Continuous Compliance: connectors, rule sets and masking jobs

Core idea. Delphix's masking engine models the pipeline as four objects you configure once and reuse: a *connector* (how to reach a source or target), a *rule set* (which tables or files), an *inventory* (which columns get which algorithm) and a *job* (the thing that runs). The terminology page defines masking as iterating through a dataset to find sensitive fields and replace them with desensitised values, and a rule set as a group of tables or flat files on which profile, masking or tokenization jobs run. Delphix masking terminology ↗



- **Extract subset.** In a database rule set each table can have a Filter (a WHERE clause) or a custom SQL SELECT, and a logical key that uniquely identifies a row (Managing rule sets).
- **Mask.** The inventory lists table or file, column or field, classification and chosen algorithm; an optional setting lets a profiling job update the assignment or lets the user's choice override it (Managing inventories). Managing inventories ↗
- **Load: the two job methods.** This is the clearest ETL parallel in any of the tools. *In-place* jobs update the source environment with the masked values. *On-the-fly* jobs read unmasked data from the source and write masked data to the target, and require a source environment and source connector as well as the target. Supported pairs are same-type databases or files, and database to delimited file (Create, view, edit, delete, verify jobs). Create, view, edit, delete, verify jobs ↗
- **Load options.** Commit Size, Update Threads, Batch Update, Truncate (on-the-fly only), Drop Indexes, Disable Triggers, Disable Constraints, a Pre SQL and Post SQL script, Number of Streams for parallel tables, Row Limit, Feedback Size, memory settings and email notification. Multi-tenant jobs choose connectors at run time (same page).
- **Operate.** A submitted job is run or queued depending on memory and a limit on simultaneously running jobs (default 7); a stopped running job leaves semi-masked data, and indexes, constraints or triggers that the job dropped are not recreated (Running and Stopping Jobs). Running and Stopping Jobs ↗ Job executions show states such as Queued, Running, Succeeded, Failed and Cancelled (Job monitoring).
- **Provisioning.** A masking job can be marked eligible for masked provisioning of virtual databases under a feature called Selective Data Distribution (jobs page); the connectors and VDB chapter covers it.

3.40.3 GenRocket: generators feed receivers; G-Subset is an ETL job

Core idea. GenRocket's main product creates data, so in the pipeline it replaces the "source". Its five key components give each ETL stage a name: a *domain* (like a table) with *attributes* (like columns), a *generator* that creates values, a *receiver* that "receives the data from an

Attribute's Generator and morphs it into a usable format", and a *scenario* with the instructions for what to generate and how much. GenRocket Five Key Components ↗

- **Load = receiver.** The Receivers overview describes 120+ receiver types that format and deliver generated data: SQL and NoSQL databases, web services, fixed and delimited files, big data formats such as Parquet and Avro, message formats such as Kafka and MQ, packaged applications, EDI, documents and images, and cloud environments. More than one receiver can be attached to a domain, and data is generated for each in order. Receivers Overview ↗ The FAQ adds that a receiver does not know whether it delivers to a mainframe or a PC; it only knows its format and destination. For large volumes, native bulk load receivers are preferred over JDBC. Pre and post SQL runs through the SQLUpdateReceiver.
- **Where it runs.** The Runtime executes scenarios on premises, behind your firewall (General FAQ). General FAQ ↗ This is the ETL engine; the cloud part is where you model and design (Architecture page).
- **Synthetic replacement as a masking transform.** "Synthetic Data Replacement" replaces sensitive values with generated ones and is used in G-Subset and file masking. What is SDR? ↗

Generated data for testing ETL itself. GenRocket's 2020 webinar and 2022 overview video both list ETL testing among its uses. In the 2020 webinar GenRocket's CEO said customers use it to test data warehouses, data lakes and ETL, and mentioned an ETL-testing webinar given with another vendor. The 2022 overview lists data for ETL testing among its use cases. These are vendor statements, but the idea is easy to apply: generate source-system data that contains known edge cases (a null in a required field, a duplicate customer, a date on 29 February, the largest allowed amount), run the ETL job, and compare the target with the result you worked out by hand for each transformation. Because you chose the input, a difference points at one transformation.

G-Subset as an ETL job. G-Subset is GenRocket's subsetting and masking feature for databases. Its description reads like our reference architecture. G-Subset Overview ↗

1. **Import the schema.**
   You upload only the source database's metadata, not customer data. This is the "model" step and the source profile.
2. **Define subset conditions.**

A "Where" clause, a percentage of rows or a number of rows, and the sensitive columns to replace.

3 **Download the configuration.**
The files are the instruction set. The actual work runs in your environment.

4 **Extract into temporary dump files.**
Rows to be subset are read from the source tables into temporary files on the machine doing the migration. This is the staging area. In default mode, sensitive columns are never read: the dump file marks them "to be masked" and fills them with synthetic data from an automatically created domain and scenario.

5 **Load.**
The data is inserted into the destination tables, directly via JDBC with bulk files or using SQL insert files, and the temporary files are removed afterwards.

Expected result: Result: a smaller, masked copy in a destination database of the same type as the source.

Here the ETL parts are named: extract into staging, transform (replace), load, clean up. Note one design difference from most masking tools: by default the sensitive values are not read at all, which is a privacy advantage over reading and then hiding them. The overview also describes an optional conditional masking feature, disabled by default, that does read the actual values.

3.40.4 Side by side

Pipeline stage	Informatica TDM	Delphix masking engine	GenRocket
Profile	Data discovery profiles; data domains	Profiling jobs update the inventory	Metadata import; column-name profiling for PII in G-Subset (it checks column headers against defined names)
Define what to move	Entity with driving table; group; criteria	Rule set: tables or files, filter, logical key	G-Subset configuration; scenario with domains
Source of rows	Relational, flat file, XSD sources	Connector to a database or files	Source database via JDBC, or generators
Staging	Staging connection (database)	Not documented	Temporary dump files
Transform	Masking rules and policies; mapplets; generation rules	Inventory algorithms (secure lookup, mapping, segment mapping and others)	Generators; Synthetic Data Replacement
Engine	PowerCenter Integration Service	Masking engine job with streams and threads	GenRocket Runtime on premises
Load	Target connection; test data warehouse	In-place or on-the-fly job; optional truncate	Receivers; JDBC or bulk files
Specification	Plan, generated into a workflow	Job saved in the engine	Scenario (and G-Cases)

Pipeline stage	Informatica TDM	Delphix masking engine	GenRocket
Monitor	Monitor view; session details	Job executions, logs, states	Not documented

> **WHAT "NOT DESCRIBED" MEANS**
>
> An entry that says "Not documented" means the documentation does not state it. It does not mean the tool lacks the feature. Check your product version.

3.41 An end-to-end ShopSphere story: profile, subset, mask, load, test

The request. The ShopSphere QA team needs a test copy of "Customer Orders" for a release. They need customers who ordered on 20 October 2026, with all of their orders and items. The compliance officer adds one rule: no real SSN, phone, email, tax id, name, date of birth or exact zip code may reach the test database. You are the TDM engineer. Here is the plan, with every step labelled as ETL-style or TDM-specific.

Step	What happens	ETL-style?	TDM-specific part
1. Profile	Scan CUSTOMERS columns for SSN-like and phone-like patterns	Partly (source profiling)	Classifying columns as sensitive; steward approval
2. Subset	Choose the customers with an order on 2026-10-20	Yes (filtered extract)	Children and parents must travel together
3. Mask	Replace names, email, phone, SSN, tax id, birth date and zip	Yes (transformation)	Deterministic, format-preserving, nothing real left
4. Load	Insert customers, then orders, then items into the test tables	Yes (load, parents first)	Target must be a non-production system
5. Test	No orphans, no leaked values, counts match, rerun gives same output	Reconciliation is ETL-style	The leak check is TDM-specific

The SQL below is a hand-made miniature of the pipeline. A tool would do the same work from rules and plans, with logging; the goal here is to see each stage with your own eyes. Mark which statements are extract, transform and load as you go.

Lab: run the ShopSphere TDM pipeline by hand

Goal: profile, subset, mask, load and verify, and confirm that the test tables contain no original SSN or email. Use an empty SQLite database (or adapt the function names for another database).

1 **Create the fake production tables, the staging helper, a name dictionary and empty test tables.**

Production stays read only for the whole lab; we only ever select from it.

```
-- "Production" (fake data, as in the shared ShopSphere
example)
CREATE TABLE customers (customer_id INTEGER PRIMARY
KEY, first_name TEXT, last_name TEXT, email TEXT,
  phone TEXT, ssn TEXT, date_of_birth TEXT, zip_code
TEXT, tax_id TEXT);
INSERT INTO customers VALUES
 (1001, 'Asha',  'Menon', 'asha.menon@example.com',
'9876543210', '123-45-6789', '1990-04-12', '560001',
'AAAPM1234C'),
 (1002, 'Ravi',  'Kumar', 'ravi.k@example.com',
'9123456780', '234-56-7890', '1985-11-02', '600017',
'BBBPK5678D'),
 (1003, 'Meera', 'Nair',  'meera.n@example.com',
'9988776655', '345-67-8901', '1999-07-23', '682001',
'CCCPN9012E');
CREATE TABLE orders (order_id INTEGER PRIMARY KEY,
customer_id INTEGER REFERENCES customers,
  order_date TEXT, total_amount REAL);
INSERT INTO orders VALUES
 (5001, 1001, '2026-10-20', 1499.00), (5002, 1002,
'2026-10-20', 250.50),
 (5003, 1001, '2026-10-21', 75.00),   (5004, 1003,
'2026-10-21', 100.00);
CREATE TABLE order_items (item_id INTEGER PRIMARY KEY,
order_id INTEGER REFERENCES orders,
  product_code TEXT, quantity INTEGER, unit_price
REAL);
INSERT INTO order_items VALUES
 (1, 5001, 'P-100', 1, 1499.00), (2, 5002, 'P-200', 2,
100.00), (3, 5002, 'P-300', 1, 50.50),
 (4, 5003, 'P-400', 1, 75.00),   (5, 5004, 'P-200', 1,
100.00);
-- Staging area, test target and a dictionary of fake
first names
CREATE TABLE subset_keys (customer_id INTEGER PRIMARY
KEY);
CREATE TABLE mask_names (slot INTEGER PRIMARY KEY,
first_name TEXT);
INSERT INTO mask_names VALUES (0,'Anil'), (1,'Bina'),
(2,'Chitra'), (3,'Dev');
CREATE TABLE t_customers AS SELECT * FROM customers
WHERE 0;
CREATE TABLE t_orders AS SELECT * FROM orders WHERE 0;
```

```
CREATE TABLE t_order_items AS SELECT * FROM order_items
WHERE 0;
```

Expected result: No output. `SELECT COUNT(*) FROM customers;` returns 3.

2 **Profile.**

Count values that match an SSN pattern and a 10-digit phone pattern. Real profilers do this across all columns and also check column names.

Expected result: Both counts are 3, which tells you these columns are fully populated with sensitive-looking data.

3 **Extract the subset keys.**

The driver is customers with an order on 2026-10-20. The keys go into a small table that later steps join to; that is our staging area.

Expected result: Customers 1001 and 1002. Customer 1003 is not selected, since their only order is on the 21st.

4 **Transform and load customers.**

Each sensitive column gets a rule built from the customer id, so the same input always produces the same output. The name comes from a four-name dictionary chosen by `customer_id % 4`.

Expected result: Two masked rows, for example 1001 becomes Bina Tester1001, user1001@example.test, SSN 900-00-1001, zip 560000.

5 **Extract and load the children.**

Orders of the subset customers, then items of those orders. Parents were loaded first.

Expected result: Orders 5001, 5002 and 5003. Customer 1001 brings order 5003 although it is dated the 21st: the subset follows the customer, not the date.

6 **Test.**

Check for orphans, leaks and counts.

Expected result: Orphans 0, leaked SSNs 0, leaked emails 0, and counts 2 customers, 3 orders, 4 items.

7 **Repeatability.**
A deterministic rule gives identical output on every run, which lets a refresh produce the same masked values.

Expected result: The query returns 1 (true).

Run steps 2 to 7 as one script:

```
-- STEP 1 (profile): which columns look sensitive?
SELECT COUNT(*) AS ssn_like_values FROM customers
WHERE ssn GLOB '[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-
9]';
SELECT COUNT(*) AS phone_like_values FROM customers
WHERE phone GLOB '[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]
[0-9]';
-- STEP 2 (extract a subset): choose the driver rows
INSERT INTO subset_keys SELECT DISTINCT customer_id FROM orders
WHERE order_date = '2026-10-20';
SELECT * FROM subset_keys;
-- STEP 3 (transform = mask, and load customers)
INSERT INTO t_customers
SELECT c.customer_id,
       m.first_name,
       'Tester' || c.customer_id,
       'user' || c.customer_id || '@example.test',
       '90000' || SUBSTR('00000' || c.customer_id, -5),
       '900-00-' || SUBSTR('0000' || c.customer_id, -4),
       SUBSTR(c.date_of_birth, 1, 4) || '-01-01',
       SUBSTR(c.zip_code, 1, 3) || '000',
       'MASK' || SUBSTR('000000' || c.customer_id, -6)
FROM customers c
JOIN subset_keys k ON k.customer_id = c.customer_id
JOIN mask_names m ON m.slot = c.customer_id % 4;
SELECT * FROM t_customers;
-- STEP 4 (extract and load the children that belong to the
subset)
INSERT INTO t_orders SELECT o.* FROM orders o JOIN subset_keys k
ON k.customer_id = o.customer_id;
INSERT INTO t_order_items SELECT i.* FROM order_items i JOIN
t_orders o ON o.order_id = i.order_id;
SELECT * FROM t_orders;
-- STEP 5 (test the result): no orphans, no leaked values, no
lost rows
SELECT COUNT(*) AS orphan_orders FROM t_orders o
LEFT JOIN t_customers c ON c.customer_id = o.customer_id WHERE
c.customer_id IS NULL;
SELECT COUNT(*) AS leaked_ssn FROM t_customers t JOIN customers
c ON c.ssn = t.ssn;
SELECT COUNT(*) AS leaked_email FROM t_customers t JOIN
customers c ON c.email = t.email;
SELECT (SELECT COUNT(*) FROM t_customers) AS customers,
       (SELECT COUNT(*) FROM t_orders) AS orders,
       (SELECT COUNT(*) FROM t_order_items) AS items;
```

```
-- STEP 6 (repeatability): the same input gives the same masked
output
SELECT 'Tester' || 1001 = last_name AS same_as_first_run FROM
t_customers WHERE customer_id = 1001;
```

▶ **Show a worked answer**

3.42 Pitfalls: where ETL habits hurt TDM

3.42.1 Masking after load instead of before

The ETL reflex "land the raw data, then clean it" is wrong for sensitive data. Between the load and the end of masking, real values sit in an environment with wide access, and a failure leaves them there. Prefer on-the-fly or pre-load masking, or in-place masking inside a protected staging environment that testers cannot reach, and copy onward only after verification. Informatica's plan masks data inside the same workflow that loads the target, with subset first (p.275); Delphix on-the-fly jobs never write unmasked rows to the target; GenRocket G-Subset replaces sensitive values in the dump file before insertion.

3.42.2 Consistency across ETL and masking

When keys are masked, they must be masked identically wherever they appear. Example: adding 70000 to the key of the three ShopSphere customers and four orders.

```
-- Customer keys masked, order keys left alone: the join breaks
SELECT COUNT(*) FROM orders o JOIN (SELECT customer_id + 70000 AS
customer_id FROM customers) c
  ON c.customer_id = o.customer_id;                          -- returns
0

-- The same rule applied to both tables: the join works
SELECT COUNT(*) FROM (SELECT order_id, customer_id + 70000 AS
customer_id FROM orders) o
  JOIN (SELECT customer_id + 70000 AS customer_id FROM customers) c
  ON c.customer_id = o.customer_id;                          -- returns
4
```

Two further consistency risks come from ETL-style staging. First, if the same person appears in two systems (CRM and billing) and each is masked by a separate job with a random rule,

the person gets two different fake identities and cross-system tests fail. Use deterministic rules and share the algorithm and its key across jobs. Delphix algorithms differ in this respect: some are non-deterministic and "cannot guarantee the same output for the same input across masking jobs", while others are fixed (Algorithm syncability ↗); the terminology page also notes that a secure lookup can collide, which matters when a column must stay unique. Second, a *reversible* method such as tokenization (the only algorithm type that allows reversing, per the terminology page) is a deliberate exception, not a default.

3.42.3 Performance

Masking adds work to every row, and subsetting adds joins. Plan for it as you would for ETL: choose commit sizes, parallelism and index handling, and expect trade-offs. Delphix's guidance includes number of streams, update threads (reduce them if you see database deadlocks), and the option to drop indexes during a job; Informatica allows up to eight workflows at once. A subset reduces volume, which is usually the biggest performance win, so extract less before you optimise anything else.

3.42.4 Audit trails

An ordinary ETL log says whether the job ran. A TDM log must also help answer: who masked this, with which rules, and when? What the tools provide:

- **Informatica:** an Audit Trail Report listing recent activities in Test Data Manager (type of activity, object, user, date and time, with filters and CSV download), a Data Masking Report listing columns with their assigned rules and policies, a Plan Audit Report, and a Row Count Report (User Guide 10.5.10, pp.321 to 322, 324). The Monitor view shows job types such as Generate Workflow, Execute Workflow and Reset Data Set from Test Data Warehouse (p.316).
- **Delphix:** audit logs record user actions in the UI or REST API: the action, user and time, with action categories such as Create, Edit, Run, Login; the UI page is for admin users and can export a PDF. Audit logs ↗
- **GenRocket:** the documentation does not describe an audit log for the G-Subset feature, so check your version.

OTHER GOTCHAS

- **Treating the exit code as proof.** A job can finish and still have left a column unmasked. Always run the leak check.
- **Masking only some copies of a value.** The same SSN in a notes column or a file export remains. Profile every column and every file.
- **Letting a rerun double-load.** Use truncate, upsert or a reset path so a rerun does not duplicate rows.
- **Overwriting masked data with a refresh by accident.** Name and version data sets, as the test data warehouse does.

3.43 Recap

RECAP

- A TDM job is an ETL job with a different purpose: safe, small, consistent test data. Informatica TDM is built on a PowerCenter ETL engine.
- The seven differences: purpose, direction and shape, output consistency, privacy, size, repeatability, and who uses it.
- Overlaps: masking is a transformation, subsetting is a filtered extract, provisioning is a load, generation is a source.
- Reference pipeline: profile, extract subset, mask, load, verify, supported by a rules repository, an orchestrator, logs and a test data store. Mask before data lands where testers can read it.
- Informatica: plan to generated PowerCenter workflow. Delphix: connector, rule set, inventory, then an in-place or on-the-fly job. GenRocket: generators feed receivers; G-Subset extracts into dump files, replaces sensitive values and loads.
- Keys must be masked identically everywhere; use deterministic algorithms.
- Verify leaks as well as counts, and keep audit evidence.