CASSION · COURSE

Routine Data and DHIS2

Work with an aggregate health information system — its data elements, org unit hierarchy and period logic — and answer the question that always follows: what exactly was counted?

Instructor	Pierre Robentz Cassion
Level	Intermediate
Learner hours	14 h
Taught in	Python · R
Updated	July 28, 2026
Sectors	Public Health · Nutrition
Standards and methodologies	UNICEF indicator definitions · Sustainable Development Goals (SDG) · Results-Based Management (RBM)

Read and run the course online at data-analysis.cassion.dev/courses/routine-data-and-dhis2

What you will be able to do

- Read a DHIS2 extract as the database it came from — data elements, category combinations, org units and periods — rather than as an unexplained spreadsheet
- Aggregate up an org unit hierarchy without double-counting a facility that moved between districts
- Treat reporting completeness and timeliness as denominators, and say what a coverage figure means when a quarter of units are silent
- Pull an extract through the Web API with a script that runs every month unchanged and records what it asked for
- Reconcile a routine aggregate against a survey estimate of the same thing, and decompose the gap into measure and selection
- Answer what was counted, by whom, over what period, for any figure the system produces

Prerequisites

None. This course assumes no prior programming experience.

Contents

I How the database is shaped **1**

1 What a value in DHIS2 actually is **2**
1.1 A value has five coordinates . . . 2
1.2 Data element: the thing that is counted . . . 2
1.3 Category combinations: the breakdown built into the element . . . 3
1.4 Data element versus indicator . . . 3
1.5 Datasets: the form, and what "expected" means . . . 4
1.6 Ask for the metadata, not just the data . . . 5
1.7 What comes next . . . 5

2 The hierarchy that changes underneath you **6**
2.1 One tree, and everything hangs off it . . . 6
2.2 Groups are not levels . . . 6
2.3 The property nobody warns you about . . . 7
2.4 What to do about it . . . 7
2.5 Reassignment, reconstructed . . . 8
2.6 Aggregating up without double-counting . . . 8
2.7 Coverage is a district-level indicator . . . 9
2.8 What comes next . . . 9

II Periods and completeness **10**

3 Periods, and the aggregation operator nobody sets **11**
3.1 Period types, and the identifier that carries them . . . 11
3.2 Which period does a value belong to? . . . 11
3.3 The three annual figures . . . 12
3.4 Why they differ . . . 12
3.5 The aggregation operator . . . 13
3.6 Deadlines, expiry and locking . . . 13
3.7 Completeness is a separate record . . . 14
3.8 What comes next . . . 14

4 Reporting rate is a denominator, not a footnote **15**
4.1 Four measures, one word . . . 15
4.2 Where the system's answer differs from yours . . . 15
4.3 Reporting rate as the denominator of everything else . . . 16
4.4 The two ways to handle a silent facility . . . 17
4.5 Report the pair . . . 17
4.6 What comes next . . . 18

III Getting the data out **19**

5 Asking the system instead of exporting by hand **20**
5.1 The monthly export is the problem . . . 20
5.2 Two endpoints, two questions . . . 20
5.3 Addressing the cube . . . 20
5.4 The response shape . . . 21
5.5 The four things to get right first . . . 22
5.6 Pull the metadata too . . . 23
5.7 What comes next . . . 23

6 An extract you can point at **24**
6.1 The question this lesson answers . . . 24
6.2 The shape . . . 24
6.3 The manifest . . . 24
6.4 The script . . . 25
6.5 Validate at the boundary . . . 26
6.6 Diff against the last extract . . . 27
6.7 What this replaces . . . 27
6.8 What comes next . . . 27

IV Reading it against something else **28**

7 Three answers to one question **29**
7.1 The meeting . . . 29
7.2 The three sources . . . 29
7.3 Hold the measure, vary the selection: 8.7% to 11.4% . . . 29
7.4 Hold the selection, vary the measure: 11.4% to 14.9% . . . 30
7.5 What each source is actually good for . . . 30
7.6 Triangulation, not reconciliation . . . 31
7.7 The sentence in the report . . . 31
7.8 When you only have routine data . . . 32
7.9 What comes next . . . 32

8 What exactly was counted? **33**
8.1 The question . . . 33
8.2 Four parts . . . 33
8.3 The provenance block . . . 33
8.4 Interrogating a figure you did not produce . . . 34
8.5 What the system cannot tell you . . . 34
8.6 Where this course leaves you . . . 35
8.7 Where module 3 leaves you . . . 35

About this course

Most of the numbers this sector reports come out of an aggregate health information system, and in most countries that system is DHIS2. Every earlier course in this programme has borrowed from one — the vaccination extract has appeared in five of them — without ever explaining where it came from.

This course explains it. Not as software training: you will not be clicking through an interface. As **the data model underneath the export**, because knowing that a value is a data element crossed with a category combination, attached to an org unit, in a period, from a dataset, is what turns an unexplained spreadsheet into something you can defend.

The question the course is built around is the one that follows every routine figure ever presented: **what exactly was counted?** Answering it needs four things the export usually does not carry — which data element, which org units, which period logic, and how many of the units that should have reported did.

The worked examples run on the DHIS2-shaped vaccination extract: 38 facilities, twelve months, six antigens, and 642 of its 2,736 facility-months carrying no report at all. And the course ends on a comparison that has three real answers to one question — acute malnutrition measured at 8.7% by the routine screening register, 11.4% by a household survey using the same measure, and 14.9% by a SMART survey using a different one.

Both Python and R throughout, with the API examples in Python because that is where most extraction scripts in this sector live.

You should have done *Data Quality Assessment*, which already teaches reporting completeness and timeliness as indicators. This course does not repeat that; it shows how the system computes them, and what it does not tell you.

Part I

How the database is shaped

CHAPTER 1

What a value in DHIS2 actually is

Lesson 1 of 8 · 90 min

Five coordinates, not one number. Data element, category option combination, org unit, period and dataset — and the difference between a data element and an indicator, which is where most confusion about routine figures starts.

1.1 A value has five coordinates

The vaccination extract looks like a spreadsheet: facility, period, antigen, doses. In the system it came from, a single stored value is addressed by five things, and knowing all five is what makes a figure defensible.

Coordinate	In the extract	What it answers
Data element	`antigen` plus the measure	What was counted
Category option combination	not present	Which breakdown of it
Org unit	`facility_id`	Where
Period	`period`	When
Dataset	not present	Which form it was collected on

Two of the five are missing from the export, and that is the normal case. Most of this lesson is about what those two carry and why their absence causes arguments.

1.2 Data element: the thing that is counted

A **data element** is the raw measure a form collects — "Penta3 doses administered", "BCG doses administered". It is a count of something, entered by a person, and it is stored exactly as entered.

```python
import pandas as pd

vax = pd.read_csv("vaccination-coverage-2024.v1.csv", parse_dates=["period"])
print(sorted(vax["antigen"].unique()))
print(vax.groupby("antigen")["doses_administered"].sum())
```

```r
library(dplyr)
library(readr)

vax <- read_csv("vaccination-coverage-2024.v1.csv")

vax |> summarise(doses = sum(doses_administered), .by = antigen)
```

Six antigens, six data elements. The `target_population` column is a seventh value of a different kind, and in a real instance it would usually not be a data element at all — it is a denominator, stored either as a data element captured once a year or as an org unit attribute.

Ask which it is. A denominator stored as an annual data element can be revised retrospectively; one stored as an attribute changes for every period at once. That difference is why a coverage figure computed in January and again in June can differ with no new doses.

1.3 Category combinations: the breakdown built into the element

A **category combination** splits a data element into the cells the form actually collects. Penta3 might be broken down by age (under 1 / 1 and over) and by sex, giving four **category option combinations** per data element.

```
Data element:  Penta3 doses administered
Category combo: Age (<1, 1+) x Sex (F, M)
Stored values:  4 per org unit per period
```

This is where the most common misreading of a DHIS2 export lives. If your extract has one row per antigen per facility per month, one of two things is true, and they are not equivalent:

- **The data element has no disaggregation**, so there is one value and nothing was lost.
- **The export aggregated across category option combinations**, in which case the disaggregation exists in the system and your file cannot recover it.

```python
expected = (vax["facility_id"].nunique() * vax["period"].nunique()
            * vax["antigen"].nunique())
print(f"{expected} expected rows, {len(vax)} in the file")
```

```r
c(expected = n_distinct(vax$facility_id) * n_distinct(vax$period) *
             n_distinct(vax$antigen),
  actual = nrow(vax))
```

$38 \times 12 \times 6 = 2{,}736$, and the file has 2,736 rows. **The grid is exactly full**, which tells you the export is at data element level with no category disaggregation surviving. If a report then asks for coverage by sex, the answer is not in this file, and it may or may not be in the system.

1.4 Data element versus indicator

The distinction that causes the most trouble, and it is worth being precise about.

- A **data element** is stored. Somebody typed it.
- An **indicator** is calculated. It has a numerator, a denominator and a factor, and DHIS2 evaluates it at whatever level you ask for.

```
Indicator:   Penta3 coverage
Numerator:   #{Penta3 doses administered}
Denominator: #{Surviving infants}
Factor:      100
```

Three consequences that come up constantly.

An indicator is not stored, so it cannot be wrong in the database. If a coverage figure is wrong, either a data element is wrong or the indicator definition is. Those have different owners and different fixes.

The indicator is evaluated at the level you request, and the arithmetic is sum-then-divide, not divide-then-average. Requesting district-level coverage sums numerators

and denominators across facilities; it does not average facility coverages. That is the right behaviour and it is the opposite of what a spreadsheet user usually does.

```python
district = vax[vax["report_submitted"] == True]
correct = (district["doses_administered"].sum()
           / district["target_population"].sum())
wrong = (district["doses_administered"] / district["target_population"]).mean()
print(f"ratio of sums {correct:.3f}, mean of ratios {wrong:.3f}")
```

```r
vax |>
  filter(report_submitted) |>
  summarise(ratio_of_sums = sum(doses_administered) / sum(target_population),
            mean_of_ratios = mean(doses_administered / target_population))
```

An indicator can be requested for a period the data does not support. Ask for annual coverage and DHIS2 will sum twelve monthly numerators over a denominator that may be an annual figure or a summed monthly one, depending on how the denominator element is configured. Getting a plausible number out is not evidence that the aggregation was right.

1.5 Datasets: the form, and what "expected" means

A **dataset** is the form a group of data elements is collected on, assigned to a set of org units, with a period type. It is the thing that decides **which units are expected to report**, which is the foundation of every completeness figure the system produces.

```python
reported = vax["report_submitted"] == True
print(f"{reported.sum()} of {len(vax)} facility-month-antigen values reported")
print(f"reporting rate {reported.mean():.1%}")
```

```r
mean(vax$report_submitted)
```

76.5%. That number only means anything because a dataset assignment says all 38 facilities were expected to submit every month. **Without the dataset, there is no denominator and no completeness.**

Note also what the extract does not carry: reporting here is all-or-nothing per facility-month across all six antigens, which is what a single dataset containing all six looks like. A system with the antigens split across two datasets would show partial months, and that structure would be visible in the data.

```python
partial = (
    vax[reported].groupby(["facility_id", "period"])["antigen"].nunique()
)
print(f"{(partial < 6).sum()} facility-months with only some antigens reported")
```

```r
vax |>
  filter(report_submitted) |>
  summarise(antigens = n_distinct(antigen), .by = c(facility_id, period)) |>
  filter(antigens < 6) |>
  nrow()
```

Zero. One dataset, six elements, submitted together.

1.6 Ask for the metadata, not just the data

The practical conclusion of this lesson. When you request an extract, request the metadata with it:

- The **data element definitions**, including how each is aggregated over periods (summed, averaged, or last value — this matters enormously and nobody asks).
- The **category combination** for each element, so you know what was collapsed.
- The **dataset assignment**, so you know how many units were expected.
- The **indicator definitions** for anything calculated, with numerator and denominator expressions.

That is four short files and they turn an export into something a reference sheet can be written from. The *Indicator Design* course asked for a source down to the field; this is what that looks like when the source is DHIS2.

1.7 What comes next

You know what a value is and where it sits. The next lesson moves it — up an org unit hierarchy that is not fixed, where a facility reassigned between districts in March means a district total that cannot simply be summed.

CHAPTER 2

The hierarchy that changes underneath you

Lesson 2 of 8 · 90 min

Org units, levels and groups, and the property nobody warns you about — the tree is current, not historical, so a facility reassigned in March silently rewrites last year's district totals.

2.1 One tree, and everything hangs off it

Every value in the system is attached to an **org unit**, and org units form a single tree: country, region, district, facility. The level a unit sits at is a property of the tree, not a column on the unit.

```
Level 1  Country
Level 2  Region
Level 3  District
Level 4  Facility        <- our 38 units
```

Aggregation is the tree walked upward. Ask for a district figure and the system sums every facility below it; ask for a national one and it sums every district. This is the same "aggregate to the grain" discipline the joining course taught, with the grain supplied by a hierarchy somebody else maintains.

```python
import pandas as pd

vax = pd.read_csv("vaccination-coverage-2024.v1.csv", parse_dates=["period"])
print(f"{vax['facility_id'].nunique()} facilities in the extract")
```

```r
library(dplyr)
n_distinct(vax$facility_id)
```

Thirty-eight, all at level 4. **The extract carries no parent**, which is the usual case and the reason most district totals in this sector are computed by joining to a facility list rather than by the system.

2.2 Groups are not levels

The extract has a `facility_type` column, and it is worth being clear about what that is.

```python
by_type = (
    vax.groupby("facility_type")["facility_id"].nunique().sort_values(ascending=False)
)
print(by_type)
```

```r
vax |> summarise(facilities = n_distinct(facility_id), .by = facility_type)
```

Facility type	Facilities
Health post	20
Health centre	16
District hospital	2

That is an **org unit group**, not a level. The distinction matters in three practical ways:

- **A unit belongs to exactly one parent and any number of groups.** Health post is a group; it does not tell you where the facility is.
- **Groups can overlap.** A facility can be in "health post", "hard to reach" and "supported by partner X" at once, and aggregating by two overlapping groups double-counts.
- **Group sets are the safe version.** A group set is a set of mutually exclusive groups — facility type is usually one — and aggregating by a group set is safe in the way aggregating by an arbitrary group is not.

Ask whether the thing you are grouping by is a group set. If it is not, check that the groups partition before you sum.

2.3 The property nobody warns you about

Here is the one that costs people a quarter.

DHIS2 stores the current hierarchy, not a historical one. A data value is attached to a facility. The facility is attached to a parent *now*. There is no record, in the ordinary data model, of which parent it had in March.

So when a facility is reassigned from District A to District B — a boundary change, a reorganisation, a correction — **all of its historical data moves with it.** District A's total for last January, already reported and already in a donor's spreadsheet, changes the next time anybody asks for it.

Three symptoms, all of which look like data quality problems and none of which is:

- A district total that differs from the same query run six months ago, with no data entry in between.
- A published annual figure that cannot be reproduced.
- Two reports of the same period disagreeing, both correct on the day they were run.

2.4 What to do about it

You cannot stop the hierarchy moving. You can stop it moving your numbers silently.

Pin the hierarchy with the extract. Pull the org unit list at the same time as the data and store it beside the values.

```
org_units = pd.read_csv("outputs/extract/2026-07/org-units.csv")
    # ea/facility id, name, parent_id, level, extracted_on

pinned = vax.merge(org_units[["facility_id", "district_id"]],
                   on="facility_id", how="left", validate="many_to_one")
assert pinned["district_id"].notna().all(), "facility with no district in the pinned tree"
```

```r
org_units <- readr::read_csv(here::here("outputs", "extract", "2026-07", "org-units.csv"))

pinned <- vax |>
  left_join(select(org_units, facility_id, district_id), by = "facility_id",
            relationship = "many-to-one")

stopifnot(!any(is.na(pinned$district_id)))
```

The join is `many_to_one` and the assertion is not decorative. A facility present in the data and absent from the pinned tree means the tree was pulled at a different time from the data, which is exactly the situation this is meant to prevent.

Aggregate from the pinned tree, not from the live one. Then a figure published in March is reproducible in September, because the tree it used is on disk.

Record the extraction date on both. The next lesson but one makes that a property of the extract script rather than a discipline.

2.5 Reassignment, reconstructed

Where the reassignment date is known — and somebody always knows it, even if the system does not — the honest handling is to carry it explicitly.

```python
reassignments = pd.DataFrame([
    {"facility_id": "FAC017", "from_district": "D01", "to_district": "D02",
     "effective": "2024-04-01", "reason": "boundary revision"},
])

def district_at(facility, period, log=reassignments):
    moves = log[(log["facility_id"] == facility)
                & (pd.to_datetime(log["effective"]) <= period)]
    return moves["to_district"].iloc[-1] if len(moves) else None
```

```r
reassignments <- tibble::tribble(
  ~facility_id, ~from_district, ~to_district, ~effective,     ~reason,
  "FAC017",     "D01",          "D02",        as.Date("2024-04-01"), "boundary revision"
)
```

That is a slowly changing dimension, and treating it as one is the correct answer. It is also more work than most teams will do, so the fallback is honest labelling:

> District totals are computed on the org unit hierarchy as at 2026-07-28. One facility was reassigned from D01 to D02 in April 2024; its 2024 data appears under D02 throughout, including for periods before the reassignment.

That sentence is the deliverable. It costs nothing and it turns an irreproducible number into a reproducible one.

2.6 Aggregating up without double-counting

Two failure modes when you do the aggregation yourself.

A facility appearing twice in the tree. Impossible in DHIS2 — a unit has one parent — but entirely possible in the CSV of facilities somebody emailed you, where a facility that moved appears under both districts.

```python
duplicated = org_units["facility_id"].duplicated(keep=False)
assert not duplicated.any(), org_units.loc[duplicated, ["facility_id", "district_id"]]
```

```r
stopifnot(!any(duplicated(org_units$facility_id)))
```

Summing across levels. A district total plus its facilities is the district counted twice. This happens when an extract mixes levels — ask for level 4 only, and check.

```python
print(org_units["level"].value_counts())
```

```r
org_units |> count(level)
```

Every extract should be at exactly one level. If it is not, the first thing your script does is filter to one, and say which.

2.7 Coverage is a district-level indicator

A point from the survey course arriving here with a mechanism. Facility catchment populations overlap, and people cross boundaries to be vaccinated, so a facility-level coverage figure is a numerator from one population over a denominator from another.

```python
by_facility = (
    vax[vax["report_submitted"] == True]
    .query("antigen == 'penta3'")
    .groupby("facility_id")
    .agg(doses=("doses_administered", "sum"), target=("target_population", "sum"))
)
by_facility["coverage"] = by_facility["doses"] / by_facility["target"]
print(by_facility["coverage"].describe()[["min", "max"]].round(3))
```

```r
vax |>
  filter(report_submitted, antigen == "penta3") |>
  summarise(coverage = sum(doses_administered) / sum(target_population),
            .by = facility_id) |>
  summarise(min = min(coverage), max = max(coverage))
```

The spread across facilities is far wider than any real difference in service delivery, because the boundary crossing is noise at facility level and cancels at district level. **Aggregate to the level where the crossing happens inside the unit**, and report facility figures as workload rather than coverage.

2.8 What comes next

You can now place a value in space. The next unit places it in time — period types, which period a late entry belongs to, and the completeness registration that decides whether the value was ever expected at all.

Part II

Periods and completeness

CHAPTER 3

Periods, and the aggregation operator nobody sets

Lesson 3 of 8 · 90 min

One file, one indicator, three annual coverage figures — 6.5%, 60.8% and 77.5% — all from defensible-sounding sentences. The difference is how the denominator aggregates over time.

3.1 Period types, and the identifier that carries them

Every value belongs to a period, and the period type is a property of the dataset the value was collected on. DHIS2's period identifiers encode the type in the string, which is worth adopting even outside DHIS2:

```
202408      monthly
2024W32     weekly
2024Q3      quarterly
2024        yearly
2024April   financial year starting April
```

They sort correctly, they cannot be ambiguous between conventions, and the type is readable without a schema. The joining course made the case for ISO period keys; this is the same argument with the system's own vocabulary.

```python
import pandas as pd

vax = pd.read_csv("vaccination-coverage-2024.v1.csv", parse_dates=["period"])
vax["period_id"] = vax["period"].dt.strftime("%Y%m")
print(sorted(vax["period_id"].unique())[:4])
```

```r
library(dplyr)
vax <- vax |> mutate(period_id = format(period, "%Y%m"))
```

3.2 Which period does a value belong to?

The question has one right answer and two wrong ones in daily use.

- **Right: the period the activity happened in.** A dose given on 28 August is August, however late the form arrives.
- **Wrong: the period the form was submitted in.** Common where data entry is batched, and it shifts activity forward by a month wherever reporting is late.
- **Wrong: the period the value was keyed in.** Same failure, one step further.

DHIS2 stores the value against the period the data entry form was opened for, so the system gets this right *if the person opened the right form*. The failure is human and it shows up as a characteristic pattern: a low month followed by a high one, which is the spike-and-dip the data quality course taught you to check for.

3.3 The three annual figures

Now the lesson. Take penta3 for the year and ask for annual coverage.

```python
penta3 = vax[vax["antigen"] == "penta3"]
reported = penta3[penta3["report_submitted"] == True]

doses = reported["doses_administered"].sum()

summed_denominator = reported["target_population"].sum()
annual_target = penta3.groupby("facility_id")["target_population"].first().sum()

print(f"doses: {doses:,}")
print(f"a) doses / summed monthly target : {doses / summed_denominator:.1%}")
print(f"b) doses / annual target, all 38 : {doses / annual_target:.1%}")
```

```r
penta3 <- vax |> filter(antigen == "penta3")
reported <- penta3 |> filter(report_submitted)

doses <- sum(reported$doses_administered)
summed <- sum(reported$target_population)
annual <- penta3 |> summarise(t = first(target_population), .by = facility_id) |>
  summarise(sum(t)) |> pull()

c(a = doses / summed, b = doses / annual)
```

Sentence	Figure
"Doses over target population, summed across the year"	**6.5%**
"Annual doses over the annual target population"	**60.8%**
"Annual doses over the annual target, among reporting facility-months"	**77.5%**

Three numbers, one file, one indicator, and each comes out of a sentence somebody would say in a meeting without blinking.

3.4 Why they differ

6.5% is a monthly figure wearing an annual label. `target_population` is an *annual* cohort — the surviving infants in the catchment for the year — and the extract repeats it in every month. Summing it across twelve months produces a denominator twelve times too large. The result is the average monthly coverage, and multiplying it by twelve gets you back to 78%.

60.8% counts silent facilities as having vaccinated nobody. The denominator is every facility's full annual cohort, and the numerator is only the doses from facility-months that reported. 107 of 456 penta3 facility-months are missing, and this figure attributes zero doses to all of them.

77.5% is the defensible one, and it needs an explicit denominator adjustment:

```python
months_reported = reported.groupby("facility_id").size()
targets = penta3.groupby("facility_id")["target_population"].first()
prorated = (targets * months_reported.reindex(targets.index).fillna(0) / 12).sum()

print(f"c) pro-rated denominator: {doses / prorated:.1%}")
```

```
months <- reported |> summarise(m = n(), .by = facility_id)
targets <- penta3 |> summarise(t = first(target_population), .by = facility_id)

prorated <- targets |>
  left_join(months, by = "facility_id") |>
  mutate(m = coalesce(m, 0)) |>
  summarise(sum(t * m / 12)) |> pull()

doses / prorated
```

Each facility contributes the share of its annual cohort matching the months it actually reported. It is coverage among reporting facility-months, and the label must say so.

3.5 The aggregation operator

Underneath all three figures is one configuration setting most analysts never see: **how a data element aggregates over periods.**

Operator	Meaning	Right for
Sum	Add across periods	Counts of events: doses, admissions, consultations
Average	Mean across periods	Stock levels, staffing, anything that is a state not an event
Last value	Take the most recent	Populations, targets, register sizes

`doses_administered` sums. `target_population` **must not** — it is a state, and it sums to twelve times itself. In a real instance the element would be configured as average or last value, and requesting annual coverage would then work. Here it is a repeated column and the correction is yours to make.

Ask for the aggregation operator with the metadata. It is one field per data element, it is invisible in the export, and it is the difference between 6.5% and 78%.

The same setting exists for aggregation across org units, where "sum" is almost always right and "average" is almost always wrong — a district's doses are the sum of its facilities', not their mean.

3.6 Deadlines, expiry and locking

Three settings that determine whether a value can still change.

- **The deadline** is when the dataset is due. It drives the timeliness figure the DQA course computes.
- **Expiry days** lock the form a fixed number of days after the period ends. After that, entry needs an unlock.
- **A lock exception** unlocks one dataset, one org unit, one period. Every one is a decision, and a system with hundreds of them has effectively no locking.

The consequence for an analyst: **an extract of a recent period is provisional.** Pull August in September and again in November and the numbers will differ, legitimately, because late entry is still arriving. That is not a data quality problem and it must not be reported as one — it is the reason the next lesson but one records an extraction date with every pull.

3.7 Completeness is a separate record

The last structural point, and it explains a number that otherwise looks inconsistent.

Completeness in DHIS2 is a registration, not a computation. When a user clicks "complete" on a data entry form, the system stores a completeness record for that dataset, org unit and period. The reporting rate is built from those records.

Which means a dataset can be **complete with no values** — somebody clicked complete on an empty form — and **full of values but not complete** — data entered, nobody clicked. Both happen constantly.

```
grid = (vax["facility_id"].nunique() * vax["period"].nunique()
        * vax["antigen"].nunique())
print(f"{grid} rows expected, {len(vax)} present, "
      f"{(vax['report_submitted'] == True).sum()} flagged reported")
```

```
c(expected = n_distinct(vax$facility_id) * n_distinct(vax$period) * n_distinct(vax$antigen
    ),
  present = nrow(vax),
  reported = sum(vax$report_submitted))
```

2,736 rows present and 2,094 flagged as reported. **The row exists either way**, which is exactly the shape the cleaning course warned about — a missing report arriving as a zero with a flag beside it.

3.8 What comes next

The reporting flag is the raw material of the completeness figure, and the next lesson turns it into a denominator. The DQA course already showed why that matters; this one shows how the system computes it, and the two ways its answer differs from yours.

CHAPTER 4

Reporting rate is a denominator, not a footnote

Lesson 4 of 8 · 75 min

The system computes four completeness measures and they disagree. Which one your report quotes decides whether a coverage figure is an estimate or a lower bound, and August is 29% either way.

4.1 Four measures, one word

The *Data Quality Assessment* course established that reporting completeness is an indicator in its own right. This lesson is about what the system means by it, because DHIS2 exposes four related measures under names that get used interchangeably.

Measure	Numerator	Denominator
Reporting rate	Datasets marked complete	Datasets expected
Actual reports	Datasets marked complete	— (a count)
Expected reports	—	Org units assigned x periods
Reporting rate on time	Completed by the deadline	Datasets expected

The first and last differ only by timeliness, and a report that quotes "reporting rate" without saying which has left a reader unable to tell whether 82% means "submitted" or "submitted on time".

```python
import pandas as pd

vax = pd.read_csv("vaccination-coverage-2024.v1.csv", parse_dates=["period"])
vax["reported"] = vax["report_submitted"] == True

facility_months = vax.drop_duplicates(["facility_id", "period"])
print(f"expected: {len(facility_months)}")
print(f"actual:   {facility_months['reported'].sum()}")
print(f"rate:     {facility_months['reported'].mean():.1%}")
```

```r
library(dplyr)

vax |>
  distinct(facility_id, period, .keep_all = TRUE) |>
  summarise(expected = n(), actual = sum(report_submitted),
            rate = mean(report_submitted))
```

456 facility-months expected, 349 submitted, **76.5%**. Note the deduplication: completeness is registered per dataset per org unit per period, not per data element, so counting the 2,736 element-level rows would answer a different question and give the same percentage by coincidence.

4.2 Where the system's answer differs from yours

Two differences, and both have caught people out.

Expected reports counts assignment, not existence. A facility that closed in March remains in the denominator for the rest of the year unless somebody un-assigns the dataset or closes the org unit. Reporting rates therefore drift downward as facilities close and nobody updates the assignment, and the drift looks like deteriorating performance.

Completeness is a click, not a calculation. A facility can enter every value and never mark the form complete; the system counts it as not reporting while the data sits there. The opposite also happens. So the reporting rate and the presence of data are two different questions:

```python
has_data = (
    vax.groupby(["facility_id", "period"])["doses_administered"]
    .sum().gt(0).rename("has_data")
)
flags = facility_months.set_index(["facility_id", "period"])["reported"]

crosstab = pd.crosstab(flags, has_data.reindex(flags.index))
print(crosstab)
```

```r
vax |>
  summarise(reported = first(report_submitted),
            has_data = sum(doses_administered) > 0,
            .by = c(facility_id, period)) |>
  count(reported, has_data)
```

On this extract the two agree exactly — every reported month has doses and every unreported month has none — which is the tidy case. **Run the crosstab anyway.** The off-diagonal cells are where a real instance keeps its most interesting problems, and a cell of "marked complete, no data" is a facility submitting empty forms to meet a target.

4.3 Reporting rate as the denominator of everything else

The operational point, and it is the reason this lesson sits in a DHIS2 course rather than only in the DQA one.

```python
monthly = (
    vax[vax["antigen"] == "penta3"]
    .groupby(vax["period"].dt.strftime("%Y-%m"))
    .apply(lambda g: pd.Series({
        "reporting_rate": g["reported"].mean(),
        "coverage_reporting": (g.loc[g["reported"], "doses_administered"].sum()
                               / g.loc[g["reported"], "target_population"].sum()),
    }))
)
print((monthly * 100).round(1))
```

```r
vax |>
  filter(antigen == "penta3") |>
  mutate(month = format(period, "%Y-%m")) |>
  summarise(
    reporting_rate = mean(report_submitted),
    coverage = sum(doses_administered[report_submitted]) /
               sum(target_population[report_submitted]),
    .by = month
  )
```

Month	Reporting rate	Coverage among reporters
July	82%	…
August	**29%**	…
September	**45%**	…
October	95%	…

Two columns, always adjacent. August's coverage figure is computed on eleven of thirty-eight facilities, and whatever it says, it says it about those eleven.

Three rules that follow, and they are the deliverable of this lesson.

- **Below 90% reporting, a coverage figure is a lower bound**, not an estimate. Label it "among reporting facilities" every time.
- **Do not compare two periods with materially different reporting rates** without saying so. August against July is a comparison of eleven facilities with thirty-one.
- **A trend line needs the reporting rate on the same chart.** Plotted alone, a coverage series through August looks like a programme collapse; plotted with completeness, the explanation is immediate.

4.4 The two ways to handle a silent facility

Neither is wrong. They answer different questions and they must be labelled.

Exclude it from both numerator and denominator. Coverage among facilities that reported. Honest, and it is what the code above does.

Impute its doses. Fill from the facility's own recent months, and say so. This is what national estimates do — WUENIC and similar exercises impute rather than leave gaps, because the question is about children rather than about paperwork.

```
by_facility = (
    vax[(vax["antigen"] == "penta3") & vax["reported"]]
    .groupby("facility_id")["doses_administered"].median()
)
missing = (vax["antigen"] == "penta3") & ~vax["reported"]
imputed_total = (vax.loc[missing, "facility_id"].map(by_facility).sum()
                 + vax.loc[(vax["antigen"] == "penta3") & vax["reported"],
                           "doses_administered"].sum())

annual_target = (vax[vax["antigen"] == "penta3"]
                 .groupby("facility_id")["target_population"].first().sum())
print(f"imputed annual coverage: {imputed_total / annual_target:.1%}")
```

```
median_by_facility <- vax |>
  filter(antigen == "penta3", report_submitted) |>
  summarise(m = median(doses_administered), .by = facility_id)
```

Never impute silently. An imputed figure and a reported one in the same column with no flag is the defect the cleaning course named: an estimate that has stopped being distinguishable from a measurement.

4.5 Report the pair

```python
summary = pd.DataFrame({
    "period": ["2024-08", "2024-09", "2024-10"],
    "facilities_reporting": [11, 17, 36],
    "facilities_expected": [38, 38, 38],
    "reporting_rate": ["29%", "45%", "95%"],
    "coverage_basis": ["among reporting facilities"] * 3,
})
```

```r
tibble::tribble(
  ~period,   ~reporting, ~expected, ~basis,
  "2024-08",         11,        38, "among reporting facilities",
  "2024-09",         17,        38, "among reporting facilities",
  "2024-10",         36,        38, "among reporting facilities"
)
```

Five columns and no coverage figure travels without them. That is the whole argument, and it is cheaper than the meeting where somebody asks why August collapsed.

4.6 What comes next

Everything so far has run on a CSV somebody exported by hand. The next unit replaces that: the Web API, what to ask it for, and a script that makes the same request every month and records what it asked.

Part III

Getting the data out

CHAPTER 5

Asking the system instead of exporting by hand

Lesson 5 of 8 · 90 min

Two endpoints that answer different questions, the dimension syntax that addresses a cube, and the four things to get right before any of it — authentication, paging, rate limits and the fact that analytics tables are stale until someone runs them.

5.1 The monthly export is the problem

Somebody opens the interface, picks the org units, picks the periods, picks the data elements, clicks export, and emails a spreadsheet. Next month somebody does it again, slightly differently, and nobody can say how.

That is the process this lesson replaces. Not because the interface is bad — it is how most people should use DHIS2 — but because **an analysis that begins with a hand-built file cannot be reproduced**, and reproducing it is the whole of what module 2 was about.

5.2 Two endpoints, two questions

The single most useful thing to know about the API is that there are two ways to get data out and they answer different questions.

Endpoint	Returns	Use it for
`/api/dataValueSets`	Raw stored values, exactly as entered	Reproducing the register; anything where you must s
`/api/analytics`	Aggregated, calculated values	Indicators, totals up the hierarchy, anything the syst

The difference matters more than it sounds.

`dataValueSets` **gives you the truth as stored** — one row per data element, org unit, period and category option combination. No aggregation, no indicator arithmetic, nothing filled in. This is what you want when the analysis has to be defensible down to the data entry.

`analytics` **gives you the answer the system would show on a dashboard** — aggregated up the hierarchy, indicators evaluated, aggregation operators applied. This is what you want when you need the same number the ministry is looking at.

Pull both when the two are supposed to agree. Where they do not, the gap is an aggregation rule you have not accounted for, and finding it is worth an afternoon.

5.3 Addressing the cube

`analytics` requests are dimensional. You are asking for a slice of a cube, and each dimension is named.

```
GET /api/analytics.json
  ?dimension=dx:PENTA3_UID;BCG_UID
  &dimension=ou:LEVEL-4;DISTRICT_UID
  &dimension=pe:202401;202402;202403
  &displayProperty=NAME
```

- `dx` — data dimension: data elements, indicators, data element operands.
- `ou` — org units. `LEVEL-4` means every unit at level 4; putting a district UID beside it scopes to that subtree.
- `pe` — periods, either explicit (`202401`) or relative (`LAST_12_MONTHS`).

Prefer explicit periods over relative ones in a script. `LAST_12_MONTHS` returns a different window every month, which is convenient for a dashboard and fatal for reproducibility. Compute the window in your code, then ask for it by name, so the request itself records what was asked.

```python
import requests

BASE = "https://play.dhis2.org/40/api"

def analytics(dx, ou, pe, session):
    response = session.get(
        f"{BASE}/analytics.json",
        params={
            "dimension": [f"dx:{';'.join(dx)}", f"ou:{ou}", f"pe:{';'.join(pe)}"],
            "displayProperty": "NAME",
            "skipMeta": "false",
        },
        timeout=120,
    )
    response.raise_for_status()
    return response.json()
```

```r
library(httr2)

analytics <- function(dx, ou, pe) {
  request("https://play.dhis2.org/40/api/analytics.json") |>
    req_url_query(
      dimension = paste0("dx:", paste(dx, collapse = ";")),
      dimension = paste0("ou:", ou),
      dimension = paste0("pe:", paste(pe, collapse = ";")),
      displayProperty = "NAME",
      .multi = "explode"
    ) |>
    req_perform() |>
    resp_body_json()
}
```

`skipMeta=false` is deliberate. The metadata block carries the names behind every UID in the response, and without it you have a table of identifiers nobody can read.

5.4 The response shape

`analytics` returns a headers block, a rows array of arrays, and a metadata dictionary. Turn it into a frame immediately and join the names on:

```python
def to_frame(payload):
    import pandas as pd

    columns = [h["name"] for h in payload["headers"]]
    frame = pd.DataFrame(payload["rows"], columns=columns)
```

```
    names = {uid: item["name"]
             for uid, item in payload["metaData"]["items"].items()}
    for column in ("dx", "ou", "pe"):
        if column in frame:
            frame[f"{column}_name"] = frame[column].map(names)
    return frame
```

```
to_frame <- function(payload) {
  cols <- vapply(payload$headers, \(h) h$name, character(1))
  rows <- do.call(rbind, lapply(payload$rows, \(r) unlist(r)))
  as.data.frame(rows, stringsAsFactors = FALSE) |> setNames(cols)
}
```

Keep the UIDs as well as the names. Names change; UIDs do not. A script that joins on a name breaks the first time somebody corrects a spelling, and it breaks silently by returning fewer rows.

5.5 The four things to get right first

Authentication. Use a personal access token where the instance supports it, basic auth otherwise, and **never put either in the script.** An environment variable or a credentials file outside the repository, always.

```
import os

session = requests.Session()
session.headers["Authorization"] = f"ApiToken {os.environ['DHIS2_TOKEN']}"
```

```
req_headers(request(url), Authorization = paste("ApiToken", Sys.getenv("DHIS2_TOKEN")))
```

An analysis repository holding a working credential for a national health information system is a serious incident, and it has happened often enough that this is the first thing a reviewer should look for.

Paging. Metadata endpoints page by default at 50. Data endpoints will happily return millions of rows and time out first.

```
def paged(url, session, page_size=1000):
    page = 1
    while True:
        payload = session.get(url, params={"page": page, "pageSize": page_size},
                              timeout=120).json()
        yield payload
        pager = payload.get("pager", {})
        if page >= pager.get("pageCount", 1):
            return
        page += 1
```

```
# Same idea: loop until page == pageCount, and never assume one request is all of it.
```

Rate and size. Ask for one district and one year at a time, not the country and five years. A request that returns in eight seconds beats one that times out at ninety, and the loop is three lines. Be a good citizen of a server that also has data clerks on it.

Analytics tables are stale. This is the one that surprises people. `analytics` reads from pre-aggregated tables that are rebuilt on a schedule, typically nightly. **Data entered today is not in `analytics` until the tables run.** `dataValueSets` sees it immediately.

So the two endpoints can legitimately disagree, and the correct response is not to file a bug. Check when analytics last ran:

```python
info = session.get(f"{BASE}/system/info", timeout=60).json()
print(info.get("lastAnalyticsTableSuccess"))
```

```r
resp <- request(paste0(base, "/system/info")) |> req_perform() |> resp_body_json()
resp$lastAnalyticsTableSuccess
```

Record that timestamp with every extract. It is the difference between "the numbers changed" and "the numbers changed because the analytics tables had not run when I pulled".

5.6 Pull the metadata too

Everything lesson 1 asked for is an endpoint.

```
/api/dataElements.json?fields=id,name,aggregationType,categoryCombo[id,name]
/api/organisationUnits.json?fields=id,name,level,parent[id]&paging=false
/api/dataSets.json?fields=id,name,periodType,organisationUnits~size
/api/indicators.json?fields=id,name,numerator,denominator,indicatorType[name]
```

Four requests. They give you the aggregation operator, the pinned hierarchy, the expected-reports denominator and the indicator definitions — which is every question the earlier lessons said you would need to ask a colleague.

Save them beside the data, and the next lesson makes that automatic.

5.7 What comes next

You can ask the system a question. The next lesson makes the asking reproducible: one script, a stated window, the metadata pulled alongside, and a manifest that records exactly what was requested and when.

CHAPTER 6

An extract you can point at

Lesson 6 of 8 · 75 min

One script, one dated folder, a manifest that records what was asked and when, and the raw response kept beside the tidy table — so a figure published in March can still be defended in September.

6.1 The question this lesson answers

Six months after you publish a district coverage figure, somebody reruns the query and gets something else. That will happen, and there are four legitimate reasons for it: late data entry, a revised denominator, a reassigned facility, and analytics tables that had not run.

You cannot prevent any of them. **You can make it possible to say which one it was**, and that is entirely a matter of what you kept.

6.2 The shape

```
extracts/
  2026-07-28/
    manifest.json          what was requested, by whom, when
    raw/
      analytics.json       the response, untouched
      dataValueSets.json
      dataElements.json
      organisationUnits.json
      dataSets.json
      indicators.json
    tidy/
      coverage_by_facility_period.csv
```

Three rules give that folder its value.

One dated folder per extraction, never overwritten. Disk is cheaper than the meeting.

The raw response is kept untouched. If your parsing turns out to be wrong, the data is still there. A tidy CSV is a derived artefact and a derived artefact cannot be un-derived.

The manifest is the point. Everything else is recoverable from it.

6.3 The manifest

```
import json
import datetime as dt
from pathlib import Path

def write_manifest(folder, request, system_info):
    manifest = {
        "extracted_at": dt.datetime.now(dt.timezone.utc).isoformat(),
        "extracted_by": "me-officer@example.org",
        "instance": request["base_url"],
        "dhis2_version": system_info.get("version"),
```

```
        "analytics_last_run": system_info.get("lastAnalyticsTableSuccess"),
        "request": {
            "endpoint": request["endpoint"],
            "dx": request["dx"],
            "ou": request["ou"],
            "pe": request["pe"],
        },
        "rows_returned": request["rows"],
        "script_version": "extract.py 1.3",
    }
    Path(folder, "manifest.json").write_text(json.dumps(manifest, indent=2))
    return manifest
```

```
manifest <- list(
  extracted_at        = format(Sys.time(), "%Y-%m-%dT%H:%M:%SZ", tz = "UTC"),
  instance            = base_url,
  analytics_last_run  = system_info$lastAnalyticsTableSuccess,
  request             = list(endpoint = "analytics", dx = dx, ou = ou, pe = pe),
  rows_returned       = nrow(tidy),
  script_version      = "extract.R 1.3"
)

jsonlite::write_json(manifest, file.path(folder, "manifest.json"), auto_unbox = TRUE)
```

Two fields are the ones that earn their place. **analytics_last_run** tells you whether late entry could have been included. **pe** records the exact periods requested, which is what makes a relative window like "last twelve months" reconstructible rather than a description of the day the script ran.

The rest is provenance the *Joining and Reshaping* course already asked for, arriving here with a system to pull it from.

6.4 The script

```
def extract(base_url, dx, ou, pe, token, out_root="extracts"):
    session = requests.Session()
    session.headers["Authorization"] = f"ApiToken {token}"

    folder = Path(out_root, dt.date.today().isoformat())
    (folder / "raw").mkdir(parents=True, exist_ok=True)
    (folder / "tidy").mkdir(exist_ok=True)

    info = session.get(f"{base_url}/system/info", timeout=60).json()

    payload = analytics(dx, ou, pe, session)
    (folder / "raw" / "analytics.json").write_text(json.dumps(payload))

    for name, path in METADATA_ENDPOINTS.items():
        meta = session.get(f"{base_url}/{path}", timeout=120).json()
        (folder / "raw" / f"{name}.json").write_text(json.dumps(meta))

    tidy = to_frame(payload)
    tidy.to_csv(folder / "tidy" / "coverage_by_facility_period.csv", index=False)

    write_manifest(folder, {"base_url": base_url, "endpoint": "analytics",
                            "dx": dx, "ou": ou, "pe": pe, "rows": len(tidy)}, info)
    return folder
```

```r
extract <- function(base_url, dx, ou, pe, token) {
  folder <- file.path("extracts", Sys.Date())
  dir.create(file.path(folder, "raw"), recursive = TRUE, showWarnings = FALSE)
  dir.create(file.path(folder, "tidy"), showWarnings = FALSE)
  # ... same shape: system info, analytics, metadata, tidy, manifest
  folder
}
```

The window is computed, then passed in. The caller decides which twelve months; the script records them. That one choice is what separates an extract you can point at from one you can only rerun.

```python
def last_full_months(n, today=None):
    today = today or dt.date.today()
    first_of_this = today.replace(day=1)
    months = []
    for i in range(n, 0, -1):
        m = first_of_this - dt.timedelta(days=1)
        for _ in range(i - 1):
            m = m.replace(day=1) - dt.timedelta(days=1)
        months.append(m.strftime("%Y%m"))
    return months
```

```r
last_full_months <- function(n) {
  ends <- seq(Sys.Date(), by = "-1 month", length.out = n + 1)[-1]
  rev(format(ends, "%Y%m"))
}
```

Note `last_full_months`, not `last_months`. **The current month is always incomplete**, and including it makes every trend appear to fall — the partial period trap from the joining course, arriving with a system that will happily serve it to you.

6.5 Validate at the boundary

The extract is a read from someone else's system, which makes it exactly the seam the cleaning course put a contract on.

```python
def check(tidy, expected_org_units, expected_periods):
    problems = []
    if tidy["ou"].nunique() != expected_org_units:
        problems.append(f"{tidy['ou'].nunique()} org units, expected {expected_org_units}"
    )
    if tidy["pe"].nunique() != expected_periods:
        problems.append(f"{tidy['pe'].nunique()} periods, expected {expected_periods}")
    if tidy["value"].isna().any():
        problems.append("null values in the response")
    return problems
```

```r
check <- function(tidy, expected_ou, expected_pe) {
  c(if (n_distinct(tidy$ou) != expected_ou) "unexpected org unit count",
    if (n_distinct(tidy$pe) != expected_pe) "unexpected period count")
}
```

The most valuable of those is the org unit count. **A silently smaller extract is the commonest API failure in practice** — a permissions change, a reorganisation, a unit

closed — and it produces a district total that is quietly lower with nothing in the file to say why.

6.6 Diff against the last extract

Two dated folders make a comparison possible, and the comparison is where the four explanations get separated.

```python
def compare(old_folder, new_folder, keys=("ou", "pe", "dx")):
    old = pd.read_csv(Path(old_folder, "tidy", "coverage_by_facility_period.csv"))
    new = pd.read_csv(Path(new_folder, "tidy", "coverage_by_facility_period.csv"))

    merged = old.merge(new, on=list(keys), how="outer",
                       suffixes=("_old", "_new"), indicator=True)
    changed = merged[merged["value_old"] != merged["value_new"]]
    return {
        "in_both": int((merged["_merge"] == "both").sum()),
        "new_only": int((merged["_merge"] == "right_only").sum()),
        "gone": int((merged["_merge"] == "left_only").sum()),
        "values_changed": len(changed),
    }
```

```r
compare <- function(old, new) {
  full_join(old, new, by = c("ou", "pe", "dx"), suffix = c("_old", "_new")) |>
    summarise(changed = sum(value_old != value_new, na.rm = TRUE),
              gone = sum(is.na(value_new)), added = sum(is.na(value_old)))
}
```

Four numbers, and each points at a different cause. **Values changed** with the same keys is late entry or a revision. **Rows gone** is a reassignment or a permissions change. **Rows added** is late reporting arriving. And a large change with `analytics_last_run` moving in between is the analytics tables, not the data.

Run this every month and keep the output. It is the DQA course's finding series, built from a source you now control.

6.7 What this replaces

```
Before:  a spreadsheet in an email, monthly, method unknown
After:   extracts/2026-07-28/, one command, manifest, raw responses,
         a diff against last month, and a figure you can defend in September
```

The script is about eighty lines. It is the single highest-return piece of code in this course, and it is the one most teams never write because the manual export works fine on the day.

6.8 What comes next

You have a routine figure you can point at. The last unit puts it beside a survey estimate of the same thing — three numbers for acute malnutrition, from 8.7% to 14.9% — and decomposes the gap into the two things that actually cause it.

Part IV

Reading it against something else

CHAPTER 7

Three answers to one question

Lesson 7 of 8 · 90 min

Acute malnutrition is 8.7% in the routine screening register, 11.4% in a household survey using the same measure, and 14.9% in a SMART survey using a different one. Decompose the gap into selection and measure, and neither source is wrong.

7.1 The meeting

The nutrition cluster has three figures for acute malnutrition in the same population and the same year. The routine screening register says 8.7%. A household survey says 11.4%. A SMART survey says 14.9%.

Somebody will ask which is right. The answer is that all three are, and the useful work is decomposing the gap into its two causes — **who was measured** and **what was measured** — because each cause has a different implication for what the programme should do.

7.2 The three sources

```python
import pandas as pd

muac = pd.read_csv("muac-screening-artibonite-2024.v1.csv")
measured = muac[muac["muac_mm"].notna()]
routine = ((measured["muac_mm"] < 125) | (measured["oedema"] == True)).mean()

print(f"routine screening register: {routine:.1%} on n={len(measured):,}")
```

```r
library(dplyr)

muac |>
  filter(!is.na(muac_mm)) |>
  summarise(rate = mean(muac_mm < 125 | oedema), n = n())
```

Source	Measure	Population	Estimate	n
Routine screening register	MUAC	Children brought to screening	**8.7%**	4,146
Household survey	MUAC	Population sample, weighted	**11.4%**	648
SMART survey	Weight-for-height z	Population sample	**14.9%**	874

Two things vary across those rows, and they vary one at a time, which is what makes the decomposition possible.

7.3 Hold the measure, vary the selection: 8.7% to 11.4%

Rows one and two both use MUAC. The only difference is who got measured.

The routine register measures **children someone brought to a screening**. The household survey measures **children sampled from the population**, with the weights the survey course built.

That is a 2.7-point gap, and it goes in the direction it always goes. Screening reaches children whose carers can reach a screening point — closer to the site, less constrained by other work, in a household with someone able to travel. The children who are hardest to reach are systematically more likely to be malnourished, and the routine figure does not contain them.

Routine data measures the served population, not the population. That is not a defect of the system; it is what a service register is. The mistake is reading it as a prevalence.

7.4 Hold the selection, vary the measure: 11.4% to 14.9%

Rows two and three are both population samples. The difference is the measurement.

MUAC and weight-for-height do not identify the same children. They correlate, but each finds cases the other misses — MUAC is more sensitive to younger and shorter children, weight-for-height to older and taller ones. Applying the standard cut-offs to the same population gives different prevalences, and in most settings weight-for-height gives the higher figure.

A 3.5-point gap here, and it means the two figures **cannot be read against the same threshold**. The 15% emergency threshold for global acute malnutrition is defined against a specific measure, and a MUAC-based prevalence compared to it is comparing two different quantities. The indicator course made this point about naming; here is what it costs.

```python
comparison = pd.DataFrame({
    "source": ["Routine register", "Household survey", "SMART survey"],
    "measure": ["MUAC", "MUAC", "weight-for-height z"],
    "population": ["screened", "population sample", "population sample"],
    "estimate": [0.087, 0.114, 0.149],
})
comparison["vs_previous"] = comparison["estimate"].diff()
print(comparison)
```

```r
tibble::tribble(
  ~source,             ~measure, ~population,         ~estimate,
  "Routine register",  "MUAC",   "screened",             0.087,
  "Household survey",  "MUAC",   "population sample",    0.114,
  "SMART survey",      "WHZ",    "population sample",    0.149
) |> mutate(vs_previous = estimate - lag(estimate))
```

Selection accounts for 2.7 points. Measure accounts for 3.5. Between them they account for the whole of the 6.2-point spread, and neither is an error.

7.5 What each source is actually good for

Source	Good for	Not f
Routine register	Caseload, workload, supply planning, trend within the served population	Preva
Household survey	Population prevalence with an interval, equity across strata	Anyth
SMART survey	Prevalence against the IPC and emergency thresholds, comparability with other surveys	Anyth

Read that table and the meeting resolves itself. The programme manager asking "how

many children will we admit next quarter" wants the routine register. The cluster asking "has the situation crossed the emergency threshold" wants the SMART survey. Asking either question of the other source produces a defensible-sounding wrong answer.

7.6 Triangulation, not reconciliation

The instinct when two sources disagree is to reconcile them — to decide which is right and adjust the other. Resist it. **The sources are measuring different things and the difference is information.**

What the difference tells you:

- **A widening gap between routine and survey** means screening coverage is falling, or the hardest-to-reach are becoming harder to reach. That is a programmatic finding and neither source alone shows it.
- **A routine figure that moves while the survey does not** is usually about case finding — a new outreach round, a new screening protocol — not about nutrition status.
- **A survey figure that moves while routine does not** means the programme is not seeing the change. That is the most actionable finding of the three.

```python
def gap_over_time(routine_series, survey_series):
    return pd.DataFrame({
        "routine": routine_series,
        "survey": survey_series,
        "ratio": survey_series / routine_series,
    })
```

```r
gap_over_time <- function(routine, survey) {
  tibble::tibble(routine, survey, ratio = survey / routine)
}
```

Track the ratio, not the difference. A ratio of survey to routine is roughly the inverse of screening coverage, and it is stable enough to be a monitoring indicator in its own right.

7.7 The sentence in the report

```
Acute malnutrition in the district, 2024:

  14.9% (95% CI 12.3-17.9) by weight-for-height z-score, from a 30-cluster
  SMART survey of 874 children, design effect 1.5.

  11.4% by MUAC in the same population, from a household survey of 648
  measured children, weighted to the sampling frame.

  8.7% by MUAC among 4,146 children brought to routine screening. This is a
  measure of the screened population, not a prevalence estimate, and is
  reported here for comparison with previous rounds of screening only.

The gap between the first two figures is a measurement difference; between the
second and third, a difference in who was measured. Neither indicates an error.
```

Ten lines, and they end an argument that would otherwise recur every quarter. The last sentence is the one that does the work.

7.8 When you only have routine data

Which is most of the time. Three honest positions:

- **Report caseload, not prevalence.** "3,700 children screened, 362 acutely malnourished by MUAC" is a true sentence that supports supply planning.
- **Report the trend, not the level.** A routine series is far more reliable about direction than about level, provided screening coverage is stable — and say that it is, with a number.
- **Anchor to the last survey.** Where a survey exists, the routine-to-survey ratio from that year can be carried forward as a stated assumption, with its date. It is an assumption and it must be labelled as one.

What you must not do is present a routine screening rate as a population prevalence. It is the most common misuse of routine data in this sector and it always understates.

7.9 What comes next

Every lesson in this course has been a version of one question. The last lesson asks it directly — what exactly was counted, by whom, over what period — and turns it into a block you attach to any figure the system produces.

CHAPTER 8

What exactly was counted?

Lesson 8 of 8 · 75 min

Four questions, a provenance block that answers them, and a worked interrogation of one figure from raw value to published percentage. The question this course exists to make answerable.

8.1 The question

Somebody presents a slide: penta3 coverage, 78%. Somebody else asks what exactly was counted.

It is not a hostile question and it is not a technical one. It is the question the whole of module 2 and module 3 have been building toward, and a figure that cannot answer it is a figure that cannot be defended, however carefully it was computed.

8.2 Four parts

What was counted? The data element, its aggregation operator, and whether any category disaggregation was collapsed on the way out.

Where? Which org units, at which level, on which version of the hierarchy.

When? Which periods, by activity date or entry date, and how complete they were at extraction.

Out of what? The denominator, its source, its year, and how many of the expected reporting units contributed.

Each of those has been a lesson. Together they are the block below.

8.3 The provenance block

```
Figure          Penta3 coverage, district, 2024
Value           77.5%

What            Data element PENTA3_DOSES ("Penta 3rd dose administered"),
                aggregation operator SUM over periods and org units.
                No category disaggregation in the extract; the element has
                none configured.
Where           38 facilities at org unit level 4, hierarchy as pinned in
                extracts/2026-07-28/raw/organisationUnits.json. No facility
                was reassigned during 2024.
When            202401 to 202412, by activity period. Analytics tables last
                run 2026-07-27 23:14 UTC, so late entry up to that point is
                included.
Out of what     Surviving infants, MoH catchment estimates 2024, projected
                from the 2015 census at 2.4% annual growth. Denominator
                pro-rated to the months each facility reported: 9,238 of a
                full-year 11,774.
Completeness    349 of 456 facility-months reported (76.5%). August 29%,
                September 45%. The figure is coverage among reporting
                facility-months and is a lower bound on district coverage.
```

```
Extract          extracts/2026-07-28/, manifest.json, script extract.py 1.3
```

Seven lines and a heading. It takes ten minutes the first time and two thereafter, because six of the seven come straight out of the manifest and the metadata the extract already pulled.

Attach it to the figure, not to the report. A slide with the number and a footnote pointing at the block is what survives being forwarded.

8.4 Interrogating a figure you did not produce

More often you are handed a number and asked whether it can be used. Six questions, in the order that eliminates the most possibilities fastest.

1. Is it a data element or an indicator? If an indicator, ask for the numerator and denominator expressions. Most disagreements end here.

2. What is the denominator's year? A 2024 numerator over a 2019 projection is common and understates coverage by whatever nine years of growth amounts to.

3. What was the reporting rate? Below 90%, the figure is a lower bound. If nobody knows, that is the answer to whether it can be used.

4. Which org unit level? A figure that mixes levels double-counts; a facility-level coverage figure is unreliable for the catchment reasons lesson 2 gave.

5. Was the period complete when it was pulled? A recent month is provisional, and a figure quoted from a pull three days after month end will not reproduce.

6. What is it being compared to? Most misuse is comparison — against a different measure, a different denominator, or a period with a different reporting rate.

```python
def interrogate(figure):
    return {
        "type": figure.get("element_or_indicator"),
        "denominator_year": figure.get("denominator_year"),
        "reporting_rate": figure.get("reporting_rate"),
        "org_unit_level": figure.get("level"),
        "period_complete_at_extraction": figure.get("period_closed"),
        "comparable_to": figure.get("comparable_to"),
    }
```

```javascript
interrogate <- function(figure) {
  figure[c("element_or_indicator", "denominator_year", "reporting_rate",
           "level", "period_closed", "comparable_to")]
}
```

A blank in any of those is the finding. You are not asking somebody to justify themselves; you are establishing whether the number can carry the decision it is about to be used for.

8.5 What the system cannot tell you

Three things no amount of API work recovers, and each has to come from a person.

Why a facility stopped reporting. The system records that it did. The reason — a staff departure, a broken tablet, a security incident, a supervisor who left — is in somebody's head and it decides whether the gap is a data problem or a service problem.

What the data entry clerk understood the field to mean. A column labelled "cases" collects whatever a person believes a case is. Two facilities can be internally consistent

and mean different things, and that is the definition problem the indicator course exists for.

What happened outside the catchment. Routine data sees what came to the service. The previous lesson decomposed exactly what that excludes.

> The system is a record of what was reported, not of what happened. Every figure it produces inherits that, and stating it once in a report is worth more than any additional decimal place.

8.6 Where this course leaves you

You can read a DHIS2 extract as the database it came from, aggregate up a hierarchy that moves, treat completeness as a denominator, pull the same extract every month with a manifest that says what you asked for, place a routine figure beside a survey estimate and explain the gap, and answer what was counted for any number the system produces.

8.7 Where module 3 leaves you

Three courses. *Indicator Design and the LogFrame* defined the number. *Survey Analysis, Sampling and Weighting* put an interval on it. This one traced it back to the system that produced it.

Together they are the answer to the platform's own standing instruction — always state how an indicator is calculated, what its denominator is, and what decision it informs — for the two kinds of data this sector actually has: a survey and a routine system.

Module 4, *Sector Analysis*, is next and is the largest in the spine at six courses. It applies all of this to the sector you report on, starting with *Nutrition Analysis: CMAM and SMART* — the WHO growth standards, the SMART plausibility report, the Sphere performance thresholds, and coverage estimation, which is where the difference between admissions over expected caseload and actual coverage finally gets the twenty-four hours it needs.

About this edition

This PDF is generated from the same source as the web version of the course, so the two cannot drift. The web edition carries the runnable code, the downloadable datasets and any corrections issued since this file was produced.

> Every dataset referenced in this course is synthetic or fully de-identified. No record traces to a real person.

This course is published in English and in French. The French edition is a professional adaptation using the terminology UNICEF, WHO, WFP, OCHA and national ministries publish in — not a literal translation.

Read and run the course online at data-analysis.cassion.dev/courses/routine-data-and-dhis2

Generated July 28, 2026.