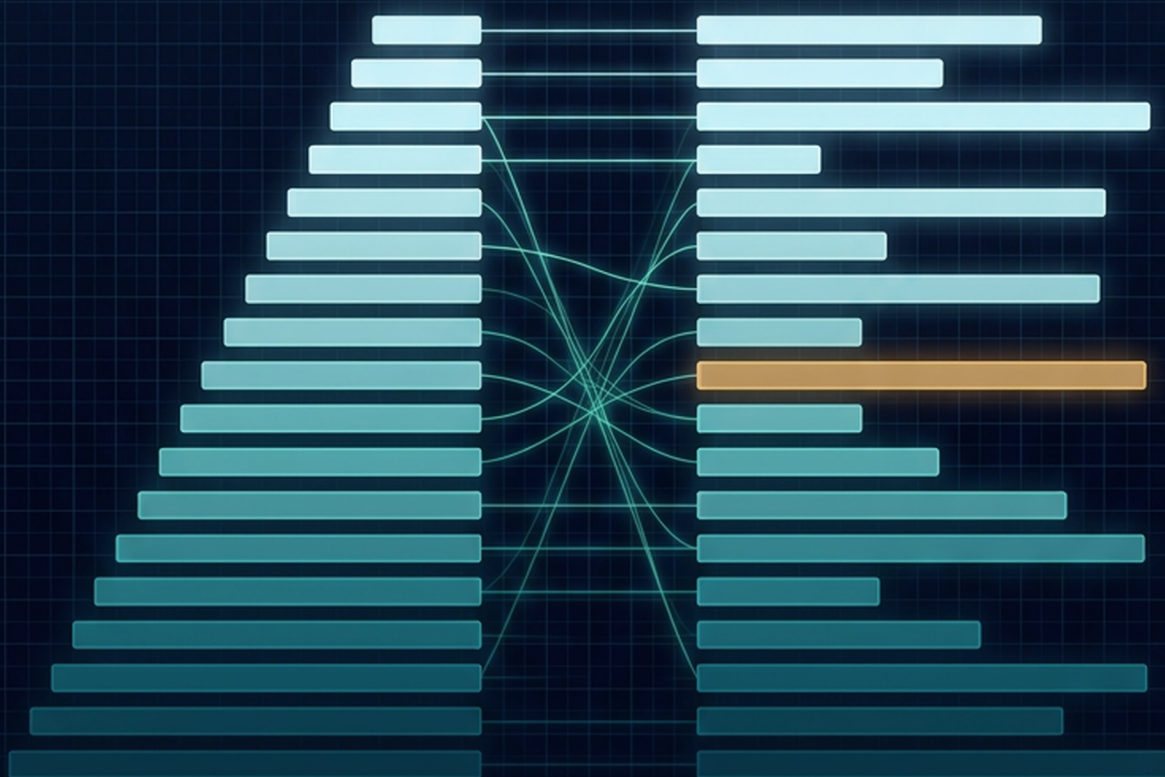


MIGRATING TO POLARS 2

What breaks silently between 1.x and 2.0,
shown with real runs under both versions



Migrating to Polars 2

What breaks silently between 1.x and 2.0, shown with real runs under both versions

Ground Truth Books

This book is available at <https://leanpub.com/migrating-to-polars-2>

This version was published on 2026-10-10



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Ground Truth Books

Contents

About This Book	i
Introduction	1
Who this book is for	1
What changed in 2.0	2
How the book is organized	5
How the examples were produced	6
 Part I: The Engine Change	 8
The Streaming Engine Is Now the Default	9
What engine='auto' resolves to	9
Eager operations are unaffected	9
sink_* already streamed	9
SQL eager=True now streams	9
Restoring in-memory execution	9
 Row Order Is No Longer Guaranteed	 10
Joins that lose their order	10
group_by and unpivot ordering	12
Requesting order explicitly	15
A left join under both versions	18
 Part II: What Breaks and How to Fix It	 19
Breaking Changes by Area	20
The guide's list, sorted by failure mode	20
concat and union strictness	20
explode empty_as_null	20
Int128 supertype and is_in coercion	20

CONTENTS

- Output names and Map dtype 20
- Decimal widening, SQL, and Parquet ENUM 21
- Sequencing the audit 21
- Casts and Operations That Now Fail 22**
 - Casts that now fail 22
 - is_in and list.contains comparisons 22
 - Operations that now raise 22
 - Five more items from the section 22
 - Casting examples under both versions 22
 - Finding them in your code 22
- API Reshapes, Removals, and Deprecations 24**
 - API reshapes 24
 - Removed APIs and their new errors 24
 - New deprecations in 2.0 24
- CSV Reading and Schema Inference 25**
 - Your CSV imports now run on the lazy reader 25
 - read_csv now dispatches to scan_csv 25
 - Arguments read_csv lost 25
 - schema_overrides and column order 25
 - infer_schema_files and counting from zero 25
 - raise_if_empty and file column order 25
- Part III: The Migration 27**
 - A Migration Checklist 28**
 - Pinning 1.44.2 and 2.0.0 28
 - A dual-version test harness 28
 - Diffing 1.x and 2.0 outputs 28
 - A CI step that runs the diff 28
 - Conclusion 29**
 - What you can now do 29
 - Where to go next 29
 - Appendix A: Commands used to produce and verify example outputs 30**
 - The two pinned environments 30
 - The lab data 30

The probe commands	30
The migrate_check.py harness	30
The audit commands	30
Appendix B: Polars 1.44.2 versus 2.0.0 API change table	31
Scope	31
Engine and row order	31
The same line, two versions	31
Removed APIs and replaced arguments	31
Casts no longer supported	31
Loud changes	31
Silent changes	32
Deprecations	32
New exception types	32
Glossary	33
References	34

About This Book

Migrating to Polars 2 is published by Ground Truth Books. Every output printed in this book was captured by running the code under Polars 1.44.2 and Polars 2.0.0 on Python 3.14.

First edition, October 2026. Copyright © 2026 Ground Truth Books. All rights reserved.

“Polars” is used descriptively. This book is not affiliated with or endorsed by Polars Inc.

Introduction

Who this book is for

Run `pip install -U polars` on a pipeline that has worked for a year, and the tests still pass. The report then comes back with rows in a different order, a join pairs the wrong records, or a nightly CSV no longer matches last month's file. No exception is raised, because from Polars' point of view the query is fine. Polars 2.0.0, released on 6 October 2026 [1, 2], switched the default execution engine behind `LazyFrame.collect()`, and the new engine does not guarantee row order for `unpivot`, `group_by`, and joins [3]. Joins and `unpivot` used to come back in a stable order on the in-memory engine; `group_by` key order was never promised, even in 1.x [3, 4]. The upgrade guide's own danger box states that the change “may silently impact the results of your pipelines” [3].

This book gives you a deterministic path from Polars 1.44.2 to 2.0.0. The path pins both versions and runs the same code under each, then diffs the outputs and fails a CI step when they differ [5, 6]. Every change that can be reproduced on the lab data appears twice, once as it behaves in 1.44.2 and once in 2.0.0, from real runs; the rest are described from the upgrade guide and say so by citing it.

You are the reader if you run Polars 1.44.2 in production: scheduled batch jobs, notebooks that feed reports, service code that transforms frames on request. You know Polars 1.x, the lazy frame against the eager one, expressions, `collect()`. You do not yet know what 2.0 changed, and you cannot afford to learn it from a production incident.

The book targets code that breaks silently. If you rely on incidental row order, taking the first row of a group, zipping two frames by position, or writing output a colleague diffs by eye, the engine switch can change your results without an error [6, 3]. If your pipeline is eager-only, much of the engine discussion is context, because eager `DataFrame` operations are unaffected by the 2.0 change [3]. The exception is `DataFrame.sql()`: it looks eager, but it runs its query through `SQLContext(eager=True)`, which calls `collect()`

and so now runs on the streaming engine [7]. You still need the API chapters, because removals and stricter dtype behaviour apply to eager code too [3].

What you end up with is a migration you can defend: a fixed 2.0.0, a checklist that turns each breaking change into one explicit edit, and a test that fails when the new engine changes your output [5, 6].

What changed in 2.0

The maintainers intended 2.0 to be unexciting. It was not meant as a big feature release, though the release post adds that it “still packs a lot” [8, 1]. At its core is the removal of design decisions that blocked further work and the tightening of defaults [8, 9]. Two headline decisions sit behind the breaking changes that affect output: the default engine changed, and Polars got stricter about implicit behaviour [8, 1]. The native Map dtype and the new deprecations sit alongside those two decisions. Chapters 4 and 6 cover them.

The engine change is the headline. `LazyFrame.collect()` and `collect_async()` still default to `engine="auto"`, but "auto" now resolves to the streaming engine for lazy queries, where it resolved to the in-memory engine in 1.x [3]. Eager `DataFrame` operations keep resolving to in-memory, and `sink_*` writes are unaffected because they already ran on the streaming engine [3]. `explain()` is unaffected and still prints the optimized plan [3]. `show_graph()` is not: its `plan_stage` default moved to "physical", so a bare `show_graph()` now draws the streaming physical graph. Pass `plan_stage="ir"` to get the old picture back [3]. The guide's introduction lumps the two together as unaffected, but its own `show_graph()` section says otherwise [3]. The streaming engine splits the data into chunks, runs them in parallel, and merges results as they finish, the general pattern behind unordered output [2]. Polars does not spell the mechanism out in the release post, so treat it as the pattern for streaming engines rather than a 2.0 promise [2]. The guide remains the contract. It promises no particular new order, says only that `unpivot`, `group_by`, and `joins` no longer keep the pre-2.0 order, and recommends sorting explicitly when you rely on it [3, 2].

Here is the same query run under both versions. A six-row frame, unpivoted and collected, in each environment:

```
1 # environment: polars==1.44.2
2 import polars as pl
3
4 lf = pl.LazyFrame({"a": ["x", "y", "z"], "b": [1, 2, 3], "c": [4, 5, 6]})
5 print(lf.unpivot(pl.selectors.numeric(), index="a").collect())
```

1 shape: (6, 3)

2

a	variable	value
---	---	---
str	str	i64
x	b	1
y	b	2
z	b	3
x	c	4
y	c	5
z	c	6

13

```
1 # environment: polars==2.0.0, same code
2 print(lf.unpivot(pl.selectors.numeric(), index="a").collect())
```

1 shape: (6, 3)

2

a	variable	value
---	---	---
str	str	i64
x	b	1
x	c	4
y	b	2
y	c	5
z	b	3
z	c	6

13

1.44.2 emits column by column, all of b, then all of c. 2.0.0 emits row by row: x's two values, then y's, then z's. Both outputs hold six rows, both look like tidy tables, and neither raises [10].

Hold two cautions before you treat the second table as the 2.0 contract. This is one machine and one run. An independent run joined a 2,000,000-row

table against a 1,000-key lookup table on 2.0.0 and got the rows out of input order by default, which is all a single run can show [2]. Another ran a 360-row join five times on 2.0.0rc2 with the default four threads, got a different result each time, and happened to return input order once [10]. The guide's wording is that the order is not guaranteed at all [3]. Treat the 2.0 table as evidence that the order can change, not as the order you will get.

You can ask for the old engine, per call or globally, and the old behaviour comes back:

```
1 # environment: polars==2.0.0, request the old engine
2 print(lf.unpivot(pl.selectors.numeric(),
   → index="a").collect(engine="in-memory"))
```

```
1 shape: (6, 3)
```

a	variable	value
---	---	---
str	str	i64
x	b	1
y	b	2
z	b	3
x	c	4
y	c	5
z	c	6

The output now matches 1.44.2 exactly on this machine. Three switches select the in-memory engine: `collect(engine="in-memory")` per query, `pl.Config.set_engine_affinity("in-memory")` process-wide, and the `POLARS_ENGINE_AFFINITY=in-memory` environment variable [3, 6]. Treat the match as luck rather than a contract. The engine argument is a request rather than a guarantee [10], and the 2.0 in-memory engine is not the 1.44.2 one. A left join of `orders.parquet` against `customers.parquet` came back in input order on three runs out of three under 1.44.2 [11]. Under 2.0.0 with `engine="in-memory"`, the same join came back unsorted, starting at a different row on each run [12]. When a result must be deterministic, sort explicitly. `join` and `group_by` also accept `maintain_order`, but `unpivot` does not, so a sort is its only remedy [3, 10]. Chapter 2 covers the switches in detail.



Set `POLARS_ENGINE_AFFINITY=in-memory` and run your existing suite against 2.0.0 before changing any code, so the loud removals and stricter errors surface before most of the row-order changes do [3, 6]. Affinity does not hold row order fixed: the 2.0 in-memory engine can reorder a join over file scans too [12]. Find your order dependencies and give each one a `sort` or `maintain_order`, whichever engine runs [3].

The second theme is strictness. Polars 2.0 removed implicit behaviour that could hide bugs. `is_in()` no longer lossily coerces operands, so checking an `Int64` value against a `Float64` list now raises `InvalidOperationError` where 1.x silently rounded values above 2^{53} to the same float [3, 8]. A signed integer plus `UInt64` moves from a lossy `Float64` result to an exact `Int128` result, silently changing both the output dtype and the computed values [3]. Horizontal concat now requires equal frame heights instead of padding with nulls [3]. Most removed APIs, melt among them, raise `AttributeRemovedError` or `ArgumentRemovedError` messages that name the replacement [3, 13]. A few raise a plain built-in exception instead, such as `AttributeError`, `TypeError`, `ImportError` or `ValueError`, and some of those give no pointer to the replacement [3]. The release also adds the native `Map` dtype and switches on out-of-core execution that spills to disk from about 80% of RAM, within a 64 GB default disk budget [1, 2]. Spilling covers sort, window functions and many expressions for now; the release post says joins and `group_by` will follow “in the coming time” [1]. The maintainers wrote that in aggregate they “expect the streaming engine to be easily 5x faster,” an expectation rather than a benchmark [8]. The figure appears in neither the release notes nor the engine-switch PR [10], and the benchmark in the release post is the vendor’s own run, whose footnote disclaims comparability with official TPC results [2].

Most breaks raise on the first run; the dangerous ones only change the answer [6]. Your migration starts at Chapter 2, which walks the engine switch and the three ways back to in-memory [3].

How the book is organized

The official upgrade guide, released alongside 2.0, is the reference. It lists every breaking change that was not previously deprecated, plus the significant new deprecations [14]. This book does not replace it. It reorders the guide’s material

by how likely each change is to corrupt a result unnoticed, so the engine switch and the row-order chapters come before the loud API and CSV breaks [6].

Chapter 2, *The Streaming Engine Is Now the Default*, covers which operations moved to streaming and which did not, plus the three switches back to in-memory [3]. Chapter 3, *Row Order Is No Longer Guaranteed*, shows where `join`, `group_by`, and `unpivot` order stops being stable and how an explicit sort or `maintain_order` restores determinism [3, 6]. Chapter 4, *Breaking Changes by Area*, walks the guide's list with a fix for each, from `concat` and `union` strictness to the `Int128` supertype, `strict is_in` coercion, output names, and `Decimal` widening [3]. It also covers `explode`'s new `empty_as_null=False` default, the `Map` dtype [1], and the SQL and Parquet ENUM handling [15]. Chapter 5, *Casts and Operations That Now Fail*, works the guide's 'Casts and operations no longer supported' section, where lenient conversions must become explicit [3]. Chapter 6, *API Reshapes, Removals, and Deprecations*, covers `profile`, the interchange protocol, `melt`, `with_row_count`, the `cut/qcut` deprecation, and the `avro` readers now marked unstable [3, 15]. Chapter 7, *CSV Reading and Schema Inference*, works through `read_csv` dispatching to `scan_csv().collect()` and what that does to arguments, `schema_overrides`, column order, and schema inference [3, 6]. Chapter 8, *A Migration Checklist*, collects each step into one runnable procedure that pins both versions and diffs their outputs, failing CI on any difference, and walks one pipeline through the full migration under both environments [5, 6]. Chapter 9 concludes, recapping which changes break silently, how to ask for the old behaviour explicitly, and how to gate the upgrade on real output diffs.

You need not read the chapters in order. To unblock an upgrade today, start at chapter 2. To gate a release, start at chapter 8 and use it as the checklist while chapters 3 to 7 supply the fixes it invokes.

How the examples were produced

Each output block in this book came from a real run against one of two fixed environments, and nowhere else. The first environment is a virtualenv with `polars==1.44.2`, and the second has `polars==2.0.0`. Both run the same Python 3.14.7 interpreter [16, 17] and read the same data files: `customers.parquet`, `orders.parquet`, and `orders_sample.csv`. We verified that both versions read the CSV with the same schema [18, 19]. We ran each query once in each

environment and captured the printed output, so the three unpivot tables above are exactly what the three runs printed [20, 21, 22].

The two-environment procedure is the one you can reproduce at home. Create two `virtualenvs`, one with `polars==1.44.2` and one with `polars==2.0.0`, run the same script in each, and compare the output [10]. An independent verification of the 2.0 release candidate used the same two-venv method, with 1.44.2 against 2.0.0rc2, to measure the row-order change [10]. Writings published before the release could only test the candidates, and they warn that candidate behaviour may differ from the final release [10], so the runs here use the final 2.0.0, confirmed at run time [23].

Streaming results can vary between runs, so a single run proves only that the order can change [2, 10]. Where a claim depends on determinism, this book shows the deterministic alternative, an explicit `sort` or `maintain_order`, under both versions [3, 6]. The first environment runs 1.44.2 [24], and the second runs 2.0.0 [23].

Chapter 2 starts where the silent damage begins, the engine switch behind `collect()`. Set up the two environments first if you want to run its examples as you read [10].

Part I: The Engine Change

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

The Streaming Engine Is Now the Default

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

What engine='auto' resolves to

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Eager operations are unaffected

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

sink_* already streamed

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

SQL eager=True now streams

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Restoring in-memory execution

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Row Order Is No Longer Guaranteed

You just collected the same left join in the book's 2.0.0 environment and the totals match. Then you compare the joined frame's first rows, and the `order_id`s no longer line up with yesterday's output. On the streaming engine, `join` and `unpivot` lose the row order they used to return, and `group_by` key order, which was not promised, becomes visibly unstable. Code that consumes rows by position produces a different answer, and the tool raises no error [10, 3]. The durable fixes are the ones chapters 1 and 2 pointed at, `maintain_order` on the operations that accept it and an explicit sort where they do not [3].

Both fixes name an ordering rule the engine can reproduce, so the row order becomes part of the query contract instead of an accident of scheduling [41, 3]. The sections below map which operations reorder in real runs under 1.44.2 and 2.0.0, then take one left join through both versions end to end.

Joins that lose their order

Start with a probe you can run in the two environments (`polars1` and `polars2`, both Python 3.14.7, same files). `orders.parquet` has 2,000,000 rows. `make_polars_lab.py` wrote `order_id` as `np.arange(n_orders)`, so it ascends from 0, and drew `customer_id` at random, so the join key itself is unordered [42]. The probe left-joins `orders` to `customers` on `customer_id`, keeps `order_id` and `region`, and collects.

```
1 out = (  
2     pl.scan_parquet("orders.parquet")  
3     .join(pl.scan_parquet("customers.parquet"), on="customer_id", how="left")  
4     .select("order_id", "region")  
5     .collect()  
6 )
```

A left join emits one output row per left row, so “did the order survive” reduces to whether `order_id` still ascends. The probe ran the join five times per version and printed the first eight `order_id`s plus a monotonic flag. Under 1.44.2, `engine="auto"` resolved to the in-memory engine, and each run kept the left order [32]:

```

1 join 1: first8=[0, 1, 2, 3, 4, 5, 6, 7] monotonic=True
2 join 2: first8=[0, 1, 2, 3, 4, 5, 6, 7] monotonic=True
3 join 3: first8=[0, 1, 2, 3, 4, 5, 6, 7] monotonic=True
4 join 4: first8=[0, 1, 2, 3, 4, 5, 6, 7] monotonic=True
5 join 5: first8=[0, 1, 2, 3, 4, 5, 6, 7] monotonic=True

```

Under 2.0.0, the same code and the same five iterations produce five different openings [43].

```

1 join 1: first8=[41, 47, 53, 54, 70, 78, 96, 98] monotonic=False
2 join 2: first8=[1648446, 1648462, 1648463, 1648465, 1648498, 1648547, 1648573,
  ↪ 1648595] monotonic=False
3 join 3: first8=[1125039, 1125051, 1125068, 1125071, 1125116, 1125117, 1125118,
  ↪ 1125122] monotonic=False
4 join 4: first8=[1046883, 1046899, 1046900, 1046903, 1046913, 1046928, 1046930,
  ↪ 1046945] monotonic=False
5 join 5: first8=[1375011, 1375018, 1375056, 1375087, 1375095, 1375107, 1375134,
  ↪ 1375157] monotonic=False

```

None of the five openings starts at 0, none of the five frames is monotonic, and the openings start at different chunk offsets: 41, 1,648,446, 1,125,039, 1,046,883, 1,375,011 [43]. That is the streaming engine's shape. It splits the query into chunks, runs the chunks in parallel, and merges each chunk's output as that chunk finishes [2]. The visible order is completion order, which the scheduler decides, so the order varies from run to run [10]. The mechanism predates 2.0. Passing `engine="streaming"` explicitly under 1.44.2 already reordered joins. 2.0 changed only the default [10].

Do not count on the in-memory engine to bring the 1.x order back, either. The same scan-based left join, collected with `engine="in-memory"` three times in one 2.0.0 process, printed this [12].

```

1 run 0 first3 [750000, 750001, 750002] sorted False
2 run 1 first3 [1000000, 1000001, 1000002] sorted False
3 run 2 first3 [875000, 875001, 875002] sorted False

```

The 1.44.2 run of the identical command printed `first3 [0, 1, 2] sorted True` on all three runs [11]. Pinning the engine removes many of the 2.0 order changes, but not this one, so a join whose consumer needs order needs `maintain_order` or a sort whichever engine runs it.

The note.com writer's summary runs through what does not change first. No exceptions. No warnings. The values remain correct. Only the order of rows changes [10].

The upgrade guide puts the warning in a danger block. "The streaming engine does not guarantee row order for operations that don't require it", it says, and advises sorting explicitly or passing `maintain_order` where the operation supports it [3]. The guide's summary sentence is "This change may silently impact the results of your pipelines", and "silently" is the word to keep [3]. After the join, the row labeled `order_id 41` still carries order 41's region. The values are correct. Only the row positions changed [10]. A downstream step that numbers rows by position, zips the frame with a fixed sequence, or writes a file that the next job reads positionally now pairs each output row with a different order. The step runs to completion, and the output still looks tidy. The exposure sits in whatever consumes the result [40].

The guide notes that joins are easy to miss, because the query does not look order-sensitive, and code review misses them for the same reason [3, 10].

An inner join behaves the same way. The dev.to run joined a 2,000,000-row orders table with a 1,000-key lookup table. With the defaults, the inner join returned rows out of input order. With `maintain_order="left_right"`, they came back in order [2]. A left join exposes the effect most cleanly, because every left row survives, unlike in an inner join, so the order check is unambiguous.

Why do tests miss it? Small data and few threads often keep the order. On a three-row left join, the streaming engine returns perfect left order, so a small fixture passes [44]. Thread count acts as a dial. In one measuring setup, a 360-row join reproduced correctly in all five runs with `POLARS_MAX_THREADS=1`, broke all five with two or eight threads, and with four threads it kept order once in five runs [10]. The 1.x behavior had been stable enough that order-dependent tests stayed green, which is exactly why the code that leans on order was written [10]. A single green run proves only that the order can stay, so it predicts little about the next run [10]. If you suspect an order-dependent test, rerun it with more threads. The measured join kept its order on a single thread and broke at two and eight [10].

group_by and unpivot ordering

The release lists `group_by` among the non-guaranteed operations [3], but group order needs its own treatment, because even 1.44.2 did not promise it. The probe aggregated orders by status and printed the group keys:

The 1.44.2 run printed this [32].

```
1 group_by default: ['refunded', 'paid', 'pending']
```

The 2.0.0 run printed this [43].

```
1 group_by default: ['paid', 'refunded', 'pending']
```

Neither version returns the keys sorted, and neither returns first-appearance order. With `maintain_order=True` both versions return `['paid', 'pending', 'refunded']`, the input's first-appearance sequence [32, 43]. The instability predates 2.0, and it reaches the in-memory engine too. In chapter 2, the same 1.44.2 default engine returned `paid`, `pending`, `refunded` for a `pl.len()` aggregation over the same file [25], and here it returned `refunded` first for a `sum` [32]. Run the same aggregation a few times and the group order can flip [4]. What changed in 2.0 is the default engine, and with it the practical stability, because the streaming engine does not bind the key order [3]. The contract that survives is narrower. Within each group, the order of rows is preserved, regardless of `maintain_order` [45]. A `group_by` promises structure, and a key sequence was not part of that promise. Code that treats “the first group is `paid`” as a fact still runs, in both versions, and some runs read a different group.

Row 0 shows the cost. The probe printed the aggregation's first row under each version [46, 47].

```
1 version 1.44.2
2 first group: ('refunded', 5991070.589999992)
```

```

1 version 2.0.0
2 first group: ('pending', 5988087.570000002)

```

Row 0 carries the refunded total under 1.44.2 and the pending total under 2.0.0. Under another 2.0.0 run, paid led the keys [43]. A report that treats row 0 as the refunded line prints a different number per version, and under the streaming engine per run. Each group's sum is correct for its own label, but do not expect a float sum to match across versions to the last digit. Floating-point addition depends on summation order, which the engine and the chunking decide. Chapter 8 compares integer cents for that reason. The label moved [46, 47].

The same five-row example from the ordering post returned ['green' , ' red'] once and [' red' , ' green'] on a later run, same code and same data [4].

A deterministic contract needs a sort. The probe appended `.sort("status")` to the same aggregation, and both versions returned ['paid' , 'pending' , 'refunded'] [46, 47]. Neither fix logs a fallback off the default engine in 2.0.0. With verbose logging, the `maintain_order=True` run and the sorted run both show their group-by and sort nodes running in the streaming graph and log no fallback (see the group_by section below) [48, 49].

`unpivot` is the sharper break. Chapter 1 showed the six-row unpivot flipping from column-by-column to row-by-row, and two consequences follow here. First, the 2.0 example order is itself not a promise. The guide says only that the order is no longer the pre-2.0 order [3]. Second, unpivot has no `maintain_order`, so the guide's own remedy for unpivot is the sort [3]. Passing the parameter raises:

```

1 unpivot maintain_order TypeError: LazyFrame.unpivot() got an unexpected keyword
  ↳ argument 'maintain_order'

```

That error is from the 2.0.0 run [43]. The issue asking for the parameter, #27153, is still open with milestone 3.0, so it will not ship in 2.0 [10]. The both-versions fix is the explicit sort, and it yields the same table under each version: the block below is `lf.unpivot(pl.selectors.numeric(), index="a").collect().sort(pl.all())`, the exact call the guide recommends, as 1.44.2 and 2.0.0 both printed it [32, 43, 3]:

1 shape: (6, 3)

2

3

a	variable	value
---	---	---
str	str	i64

4

5

6

7

x	b	1
x	c	4
y	b	2
y	c	5
z	b	3
z	c	6

8

9

10

11

12

13

Notice what the sort saved. The two versions already disagree at row 1 of the default unpivot. Under 1.44.2, that row is (y, b, 2). Under 2.0.0, it is (x, c, 4) [32, 43]. A report that reads row 1's value prints either 2 or 4, depending on the engine, and no error announces the change. The sorted table removes the dependence, because every row's contents settle into a fixed place under both versions [32, 43].

Requesting order explicitly

The probes above mapped where the order can move. The fix is to say which order you mean. A join's `maintain_order` names a side. A `group_by`'s takes a boolean. Where neither fits, sort explicitly. This section covers each.

Joins

`maintain_order` on a join takes a side. The signature accepts 'none', 'left', 'right', 'left_right', and 'right_left' [41]. The value names which frame's order you want. 'left' pins the left frame's sequence, and 'right' the right's [41]. The measured probe shows the difference. With 'left' or 'left_right', the join returned `order_ids` 0 through 7 in ascending order [32, 43]. With 'right' or 'right_left', one measured run returned the left rows in the right frame's order instead, starting at 0, 3, 6, 9 [10]. 'none' is the default. The docstring warns against trusting any observed ordering without setting the parameter explicitly, since an unspecified order can break in a future release, and leaving it unset can improve performance [41]. The probe confirms the parameter is the durable fix, because both versions honor it identically:

```
1 join maintain_order='left': first8=[0, 1, 2, 3, 4, 5, 6, 7] monotonic=True
2 join maintain_order='left_right': first8=[0, 1, 2, 3, 4, 5, 6, 7]
  ↪ monotonic=True
```

The two lines above are the 1.44.2 output [32]; the 2.0.0 run printed the same two lines [43].

Because 1.44.2 already shipped the parameter, you can add `maintain_order="left"` to a pipeline before touching the dependency, verify it in CI, and upgrade later. The intermediate commits produce identical output under both versions.

Order preservation has a measurable cost. The fix for bug #29362, which reordered unmatched rows (PR #29371), raised memory per build-side row from 4 to 8 bytes in an ordered join that outputs unmatched build-side rows, because the engine now records each row's position in the input [10]. Even Polars itself has tripped over this combination. In July 2026, it fixed multiple issues where `maintain_order` combined with `slice/head` returned wrong results (#28419, #28448, #28551, #28556), all caused by the optimizer [10].

group_by

`group_by` works differently. Its `maintain_order` is a boolean that makes the group order consistent with the input data, and it is slower than the default [45]. The `DataFrame.group_by` docstring adds that setting it to `True` blocks the possibility of running on the streaming engine [45]. The 2.0.0 `LazyFrame.group_by` docstring, the one that governs the lazy queries in this chapter, says instead that it “blocks the possibility to run partitioned” [50]. With verbose logging on, the 2.0.0 run of a `maintain_order=True` `group_by` shows the group-by node running in the streaming graph and logs no fallback to the in-memory engine [48]. Chapter 2 explains why `polars-stream` lines alone do not prove which engine ran. The point here is that nothing in the run reports a switch away from the default.

```

1 $ polars2 -c "import polars as pl
2 pl.Config.set_verbose(True)
3 print(pl.scan_parquet('orders.parquet').group_by('status',
  ↳ maintain_order=True).agg(pl.len()).collect())"
4 shape: (3, 2)
5 ...
6 | paid      | 1800165 |
7 | pending   | 100048  |
8 | refunded  | 99787   |
9 ...
10 polars-stream: running multi-scan[parquet] in subgraph
11 polars-stream: running group-by in subgraph
12 ...

```

The sorted variant logs the same way [49]:

```

1 $ polars2 -c "import polars as pl
2 pl.Config.set_verbose(True)
3 print(pl.scan_parquet('orders.parquet').group_by('status').agg(pl.len()).sort_
  ↳ ('status').collect())"
4 shape: (3, 2)
5 ...
6 | paid      | 1800165 |
7 | pending   | 100048  |
8 | refunded  | 99787   |
9 ...
10 polars-stream: running group-by in subgraph
11 ...
12 polars-stream: running sort in subgraph
13 ...

```

So both `group_by` fixes, `maintain_order=True` for first-appearance order and a key sort for a named order, run on the default engine with no fallback logged. Pick by the order you need; the engine is the same either way.

Sort as the fallback

Sorting is the universal fix, and for unpivot it is the only fix [10, 3]. The guide sorts every column after an unpivot with `.sort(pl.all())` [3]; for group keys, sort the key column with `.sort("status")` [5]. A sort also reads as a deliberate choice in review, because it names the ordering rule. The rule works for tests too: if a test asserts row order after a `join` or `group_by`, it must also assert an explicit ordering rule [5].

One caution about `maintain_order`: it pins an order, it does not make the order meaningful. `'left'` returns the left frame's sequence, `'right'` the right's; if neither sequence is the rule you actually wanted, sort explicitly [10, 41].

A left join under both versions

Put both versions on one screen. The query is one line and it never changed across the upgrade. What changed is what comes back. Under 1.44.2, in every run, row k carries `order_id k`: the first eight are `0..7`, and monotonicity holds across all 2,000,000 rows [32]. Under 2.0.0, run 1's row 0 carries `order_id 41`, run 2's row 0 carries 1,648,446, and no two runs agree [43].

The join is correct in both versions: every output row pairs an `order_id` with its own `region`. What broke is the association of rows with positions, which only matters if you consume the frame by position [10]. The probe measured that directly with `order_ids` that incremented by construction; a random-key join in production hides the same effect inside consumer code.

The step that fails silently is the one after the join. A report that pairs the joined frame row by row with a static list of 2,000,000 labels, a `.head(n)` that “means the first n orders”, or a `zip` of two frames that happened to share row order, all compute a different result under 2.0.0, differently on each run, with the values still individually correct. Each of these steps needs `maintain_order` or a sort before it.

The migration action is an audit of call sites, not a search for exceptions. For every `join`, `group_by`, and `unpivot` on a lazy frame, ask one question: does anything after this depend on row order [44]? Where the answer is yes, apply the fix from this chapter: `maintain_order="left"` on a join that needs left order, `maintain_order=True` or a key sort after `group_by`, and a sort after `unpivot` [3, 41]. Where the answer is no, leave the query alone and keep the streaming engine's speed. Only the first answer produces a deterministic order. For the second, make every comparison order-insensitive: sort before you compare, or use `check_row_order=False` in `assert_frame_equal`. Chapter 8 turns this audit into a script: it pins both versions, diffs their outputs, and fails CI on any difference, and it walks one pipeline through every fix this chapter names.

Part II: What Breaks and How to Fix It

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Breaking Changes by Area

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

The guide's list, sorted by failure mode

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

concat and union strictness

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

explode empty_as_null

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Int128 supertype and is_in coercion

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Output names and Map dtype

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Output names

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Map dtype

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Decimal widening, SQL, and Parquet ENUM

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Decimal widening

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

SQL

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Parquet ENUM

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Sequencing the audit

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Casts and Operations That Now Fail

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Casts that now fail

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

is_in and list.contains comparisons

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Operations that now raise

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Five more items from the section

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Casting examples under both versions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Finding them in your code

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

API Reshapes, Removals, and Deprecations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

API reshapes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Selectors combined with pl.col

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

The six API reshapes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Removed APIs and their new errors

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

New deprecations in 2.0

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

CSV Reading and Schema Inference

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Your CSV imports now run on the lazy reader

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

read_csv now dispatches to scan_csv

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Arguments read_csv lost

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

schema_overrides and column order

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

infer_schema_files and counting from zero

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

raise_if_empty and file column order

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Part III: The Migration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

A Migration Checklist

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Pinning 1.44.2 and 2.0.0

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

A dual-version test harness

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Diffing 1.x and 2.0 outputs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

A CI step that runs the diff

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

What you can now do

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Where to go next

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Appendix A: Commands used to produce and verify example outputs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

The two pinned environments

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

The lab data

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

The probe commands

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

The migrate_check.py harness

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

The audit commands

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Appendix B: Polars 1.44.2 versus 2.0.0

API change table

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Scope

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Engine and row order

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

The same line, two versions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Removed APIs and replaced arguments

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Casts no longer supported

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Loud changes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Silent changes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Deprecations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

New exception types

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

Glossary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.

References

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/migrating-to-polars-2>.