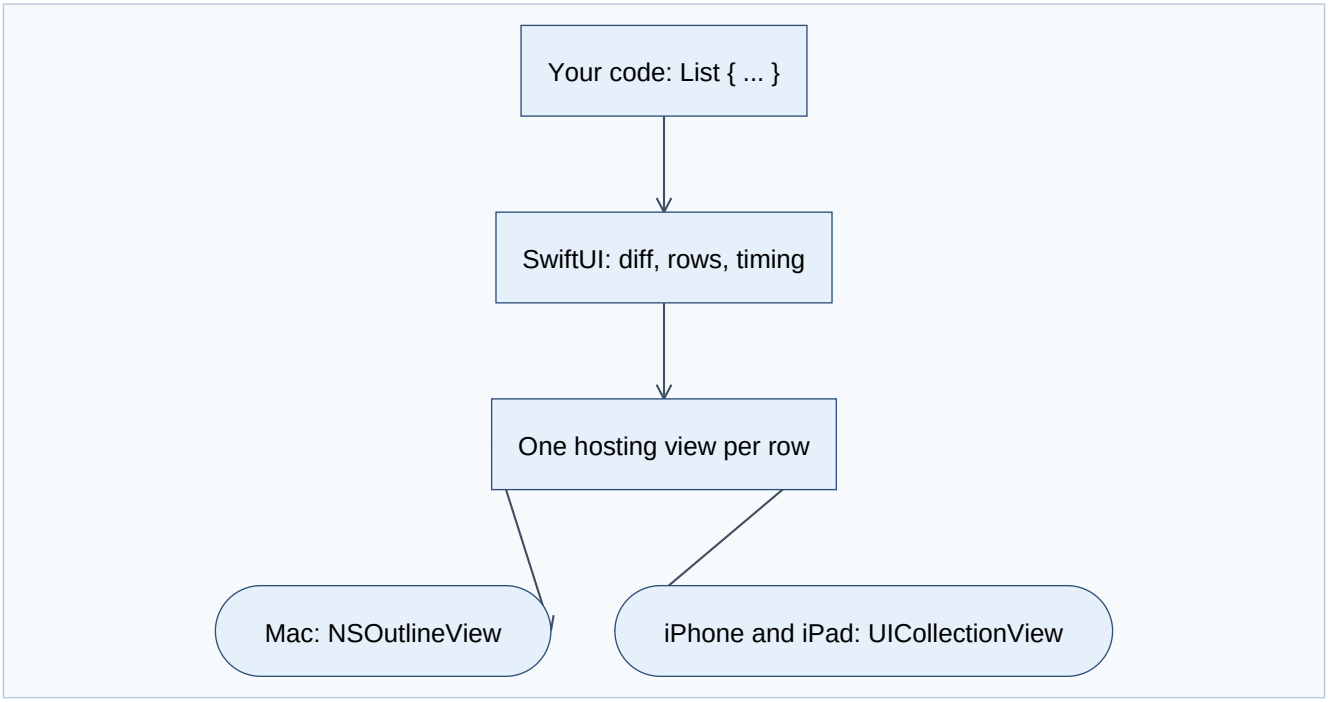


SwiftUI Doesn't Have a List. It Wraps Someone Else's.

I wrote `List`, and a list appeared. It scrolled, it recycled its rows, it animated an insert. So I assumed SwiftUI had built a list.

It has not. What is on the screen is a container that was there long before SwiftUI: a table view on the Mac, a collection view on the iPhone and the iPad. SwiftUI puts its own cells and its own change detection on top, and hands the platform container the rest of the work. This is not a secret. It is just not what the word `List` makes you picture. Here is the whole stack, from your code down to the platform:



What a List is made of: three layers are SwiftUI's, the container at the bottom is not.

I found it out because I was building a forum of my own, with long lists in it, and I had been told that a collection view is much faster than a SwiftUI List. I had no number behind that. So I measured, and to measure honestly I first had to know what I was measuring. This post is the first half of that story: what is under a `List`, how I know, what SwiftUI adds, and what that does to the cost of a change. The full measurements are in a longer article I am publishing next, and the code and raw data are public now: the link is at the end.

How I looked

I did not read this anywhere. I read it from the running app, two ways that share no code: with the Objective-C runtime, whose public functions can name the class of any object in the view tree, and with the layers of the live app through my own XRay Viewer, which reads the delegate of every layer. A layer's own class says little (`NSViewBackingLayer` on the Mac, `CALayer` on iOS), but its delegate usually says what the object is. When the two readings give the same class names, I believe them.

Before I trusted either reading on a `List`, I pointed it at a plain `NSTableView` and a plain `UICollectionView` whose behaviour I drive myself. If the probe cannot get a known answer right, its answer on the unknown is worth nothing.

What a List is

```
Mac
  SwiftUIOutlineListView
  subclass of NSOutlineView
  subclass of NSTableView

iPhone and iPad
  UpdateCoalescingCollectionView
  subclass of UICollectionView
```

Not an opinion: the runtime's own answer, one line per platform.

What is underneath

On the Mac, a `List` is a `SwiftUIOutlineListView`. It is a subclass of `NSOutlineView`, which is itself a subclass of `NSTableView`, so the Mac `List` is a table view. It sits in a scroll view with its own clip view, it has one column, and automatic row heights are on. Each live row is a `ListTableRowView` holding a `ListTableCellView`, which holds one hosting view with your SwiftUI row inside. Apple's own page for `NSOutlineView` says that subclassing it is not recommended, which is a nice detail next to a subclass that SwiftUI makes.

On the iPhone, a `List` is an `UpdateCoalescingCollectionView`. That is a `UICollectionView`, with a compositional layout, and its cells are `ListCollectionViewCell`, a `UICollectionViewListCell`. The cell's own content configuration is empty. The content is a hosting view that holds your SwiftUI row. The simulator, a real iPhone and a real iPad agree on every class.

A table view	A collection view	A collection view
Mac	**iPhone**	**iPad**
An outline view, which is a table view. A hosting view in every row.	SwiftUI's own subclass of it. A hosting view in every cell.	The same one as the iPhone, reached by a slightly different route.

Three platforms, two containers, no `UITableView` *anywhere.*

The data source and the delegate are one object of SwiftUI's own in both places: `OutlineListCoordinator` on the Mac, `UICollectionViewListCoordinator` on iOS. The iOS one is not a `UICollectionViewDiffableDataSource`. SwiftUI does its own diffing and talks to the container directly.

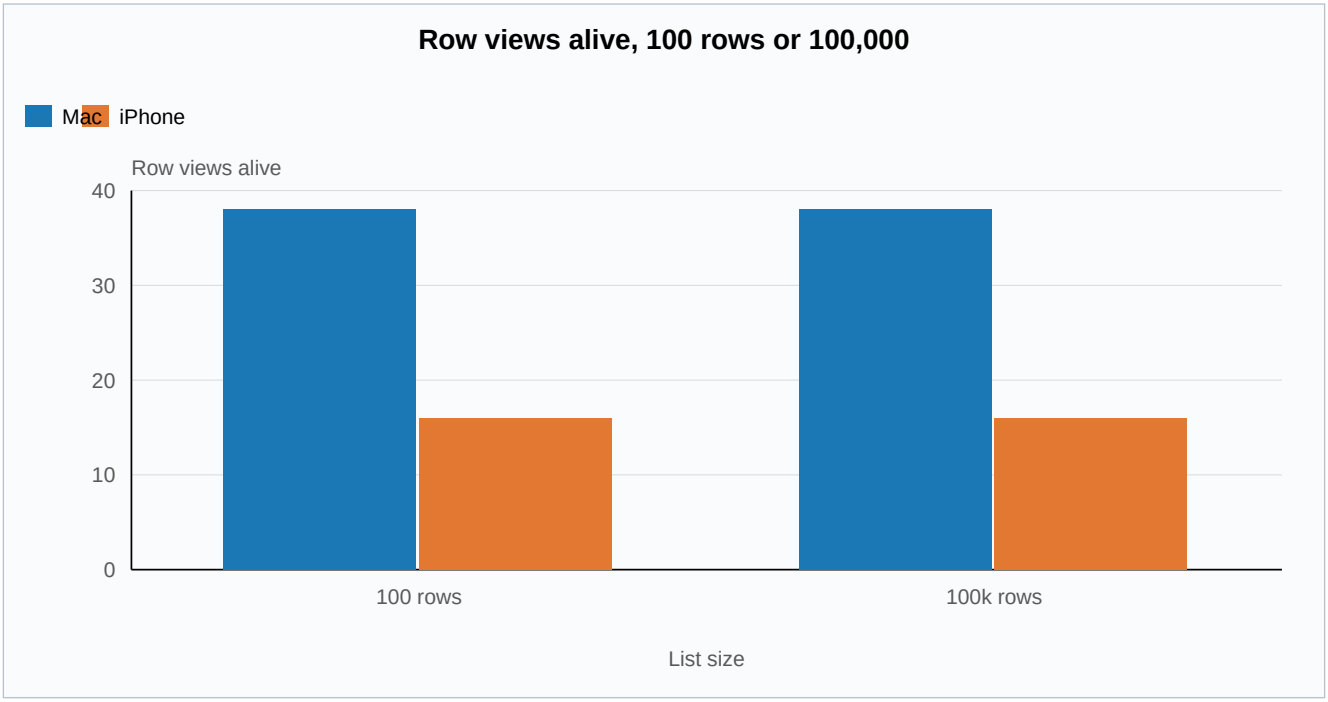
A `Table` is the same story, only more so. On the Mac it is an outline view too, `SwiftUIOutlineTableView`, with a header view and two hosting views per row in my two-column probe, one per column. On iOS it is the same collection view class as `List`, with its own cell type, one hosting view per row for all the columns. In a compact width, Apple's documentation says a `Table` hides its headers and every column after the first, which is why one cell per row is enough on a phone.

There is no `UITableView` anywhere in an iOS tree. And SwiftUI has no collection view of its own that you can reach: the one under a `List` is created and kept by SwiftUI, and what you get is the `List`.

What SwiftUI does add

It would be unfair to say it adds nothing. Three things are clearly SwiftUI's.

The first is the rows. Only the rows on screen have views, at 100 rows or 100,000: 38 row views on the Mac in a 900 point window and 16 cells on an iPhone screen, in my runs. Making only what is visible is something a table view and a collection view have always done, so recycling is the platform's. What lives in each cell is yours, hosted by SwiftUI. Apple's words for it: an outline view "retrieves only the data that needs to be displayed", and a `List` made from a collection "computes its rows on demand".

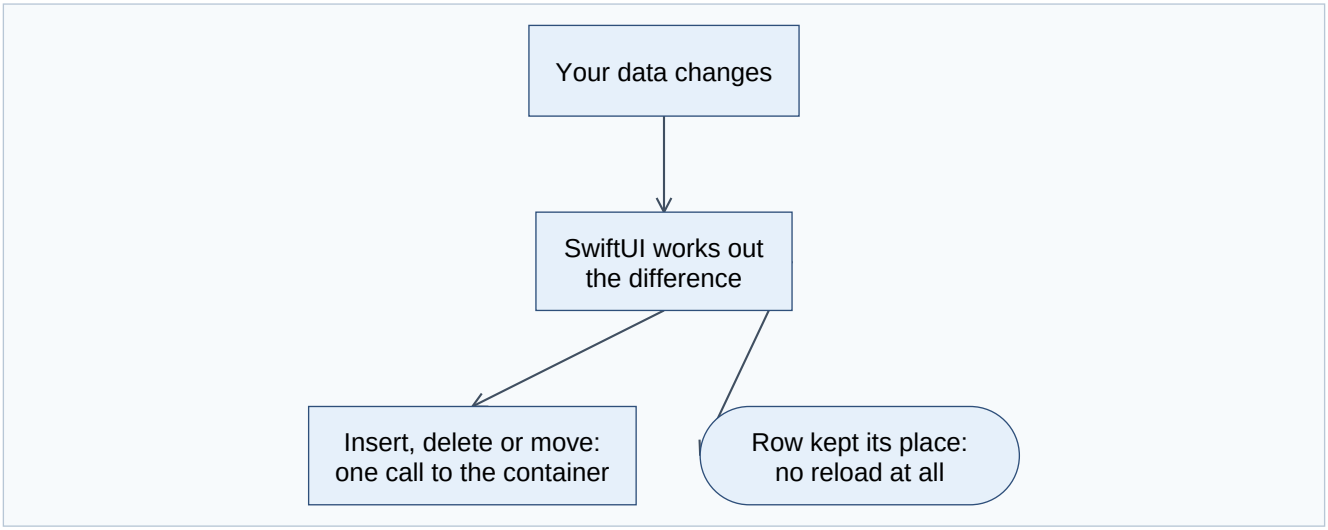


100 rows or 100,000: the same number of views.

There is a second number that says the same thing from another side. When the Mac list first appears, the table asks SwiftUI for the height of 100 rows at 100 rows, and then for 301 at 1,000, at 10,000 and at 100,000. Every answer SwiftUI gives is the same: no height, which leaves it to the table's automatic row heights. I read the count. I did not read why it is 301, and it looks like a constant of the table, not a number SwiftUI chooses.

The second thing is the diff. When your data changes, SwiftUI works out what changed with its own code, and sends the container the smallest set of structural calls it can. I hooked the container and wrote down every call, on a list of 1,000 rows:

What changed	**What the container was told**
Append 50 rows	One insert of exactly those 50 indexes (1000 to 1049 on the Mac)
Delete one row	One remove of one index
Move row 500 to the front	One move
Replace every row with new identities	One remove and one insert
A row changes and keeps its place	No reload. On the Mac an empty insert and an empty remove inside the update, on iPhone no call at all



The last arrow is the surprising one: a changed row that keeps its place is never reloaded.

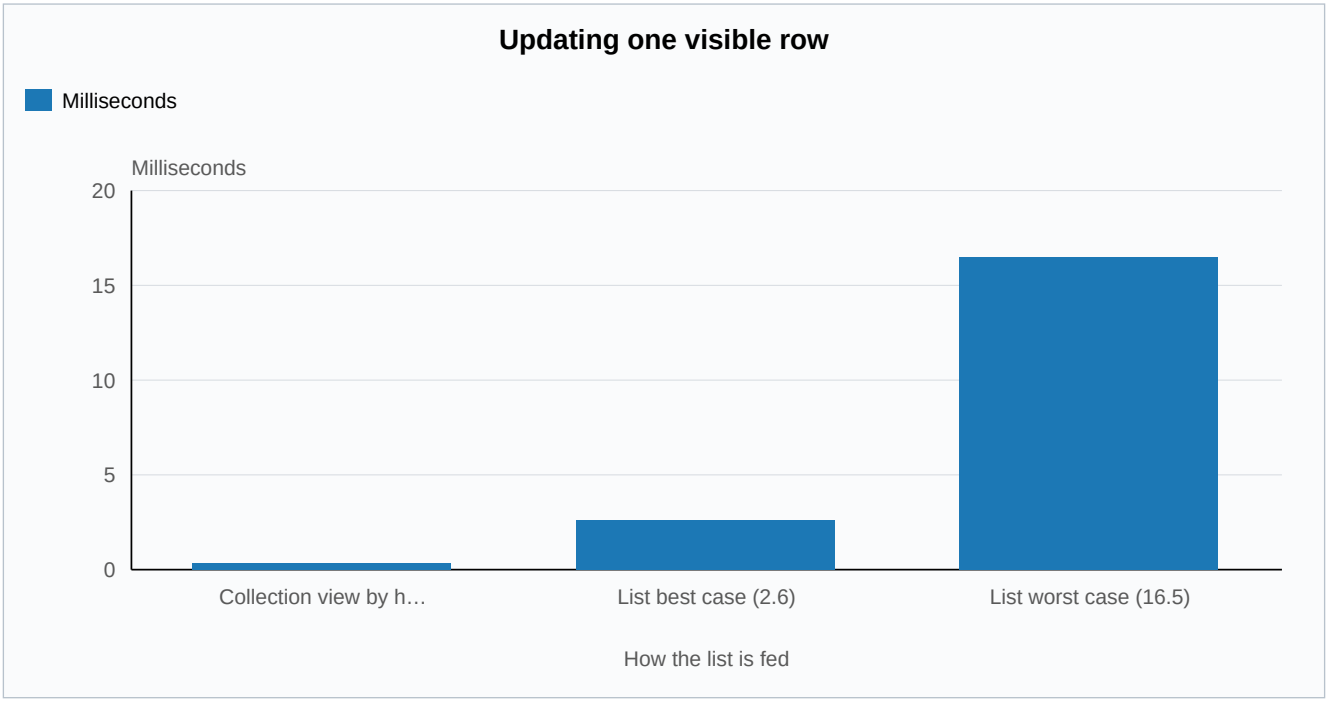
That last row of the table is my favourite finding. I put hooks on `reloadData`, `reloadItem` and the other reload calls of the container, changed one row, and then a tenth of the rows, and the hooks recorded nothing. The row still refreshed, so the refresh travels another way, through SwiftUI's own graph updating the hosting view. I can show that the container is not told. That the graph does it is my reading, not something I hooked. And there is a contrast with UIKit worth a sentence: its own advice for a hand-built list is to reconfigure existing cells instead of reloading them, for performance. SwiftUI's `List` goes one step further on the container side and tells the collection view nothing.

The third thing is timing. The diff does not run when you assign the new data. It runs at the end of the run loop turn, from an observer of SwiftUI's own on the Mac and in the simulator, and through UIKit's update cycle on a real device. If you ever time a `List`, stop the clock after the run loop has finished its turn, not when your assignment returns.

So the honest picture is neither "SwiftUI has no list" nor "SwiftUI is just UIKit". It is a layer that drives the platform's list and decides, very carefully, what to tell it.

Why it matters: the container decides the costs

I said I had heard a collection view is much faster. Here is the first place that is true. Updating a row you can see, a collection view that touches the visible cell and tells the layout nothing needs about a third of a millisecond, at any list size. The `List` needs 2.6 to 16.5.

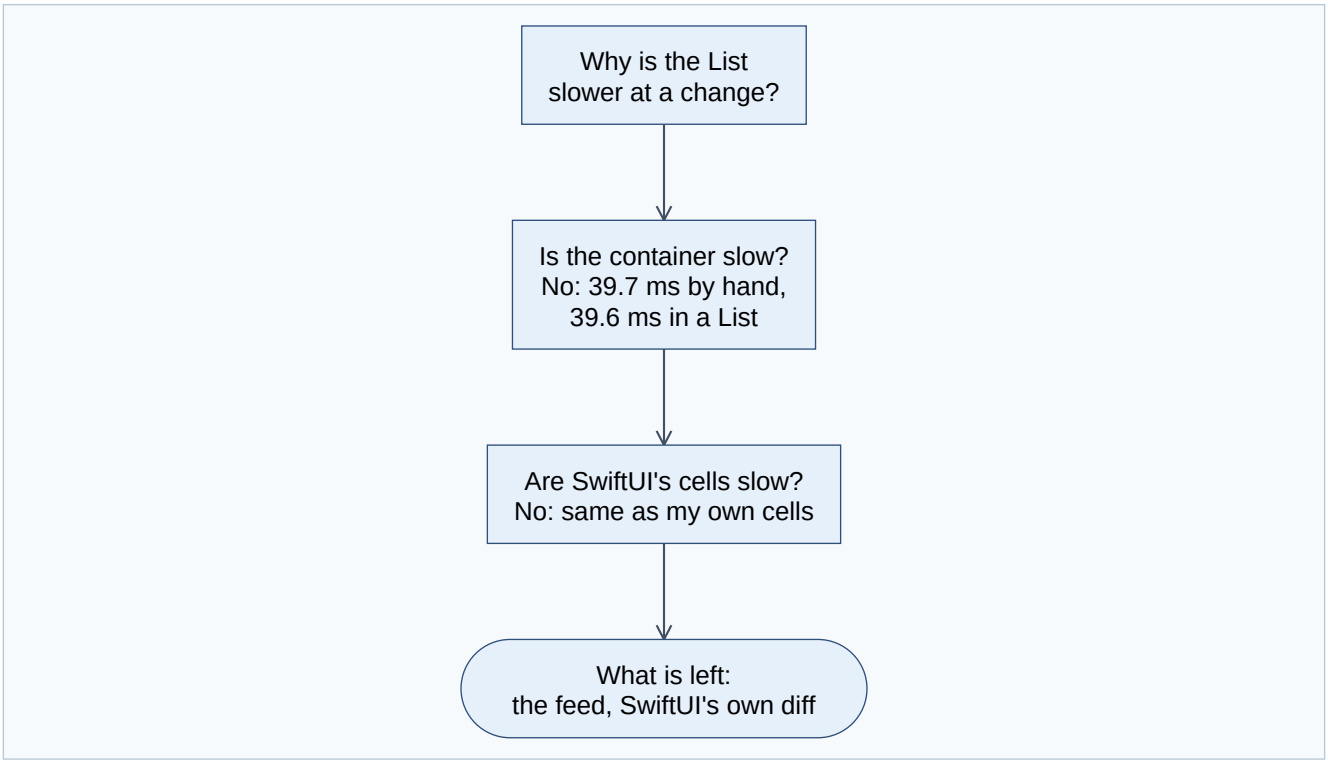


Same job, different feed. The 0.33 is "about a third of a millisecond", as measured; the two List figures are the ends of the measured range.

But I did not trust that number until I knew where the time went, and that needed two controls, the experiments that take one suspect away at a time.

The first control is a collection view whose cells are SwiftUI rows, the very row the `List` shows. If SwiftUI's hosting were the cost, that would behave like the `List`. It behaves like my own collection view, in every operation, within the same ranges.

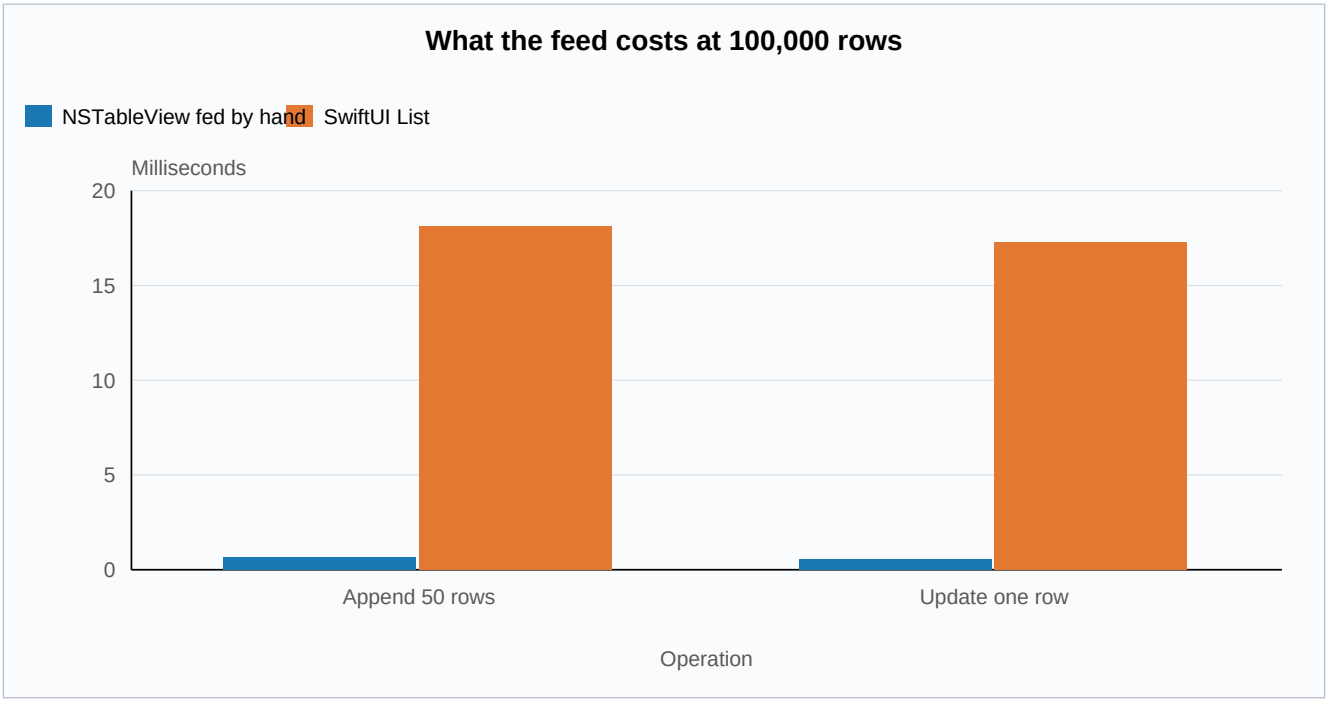
The second control is a plain `NSTableView`, the container the Mac `List` sits on, fed by hand with no SwiftUI at all. It shows 100,000 rows in 39.7 milliseconds. The `List` shows them in 39.6. Same time. So SwiftUI's container is not what makes a long list slow to show.



Remove the suspects one at a time and the feed is what remains.

What is left is the feed. At 100,000 one-line rows on the Mac, in milliseconds:

	Collection view, my cells	**Collection view, SwiftUI cells**	**`NSTableView`, fed by hand**	**SwiftUI List**
show the list	1,468	1,462	39.7	39.6
append 50 rows	1,457	1,446	0.67	18.1
update one row	1,466	1,461	0.53	17.3
one screen of scrolling	754	752	5.7	1.3



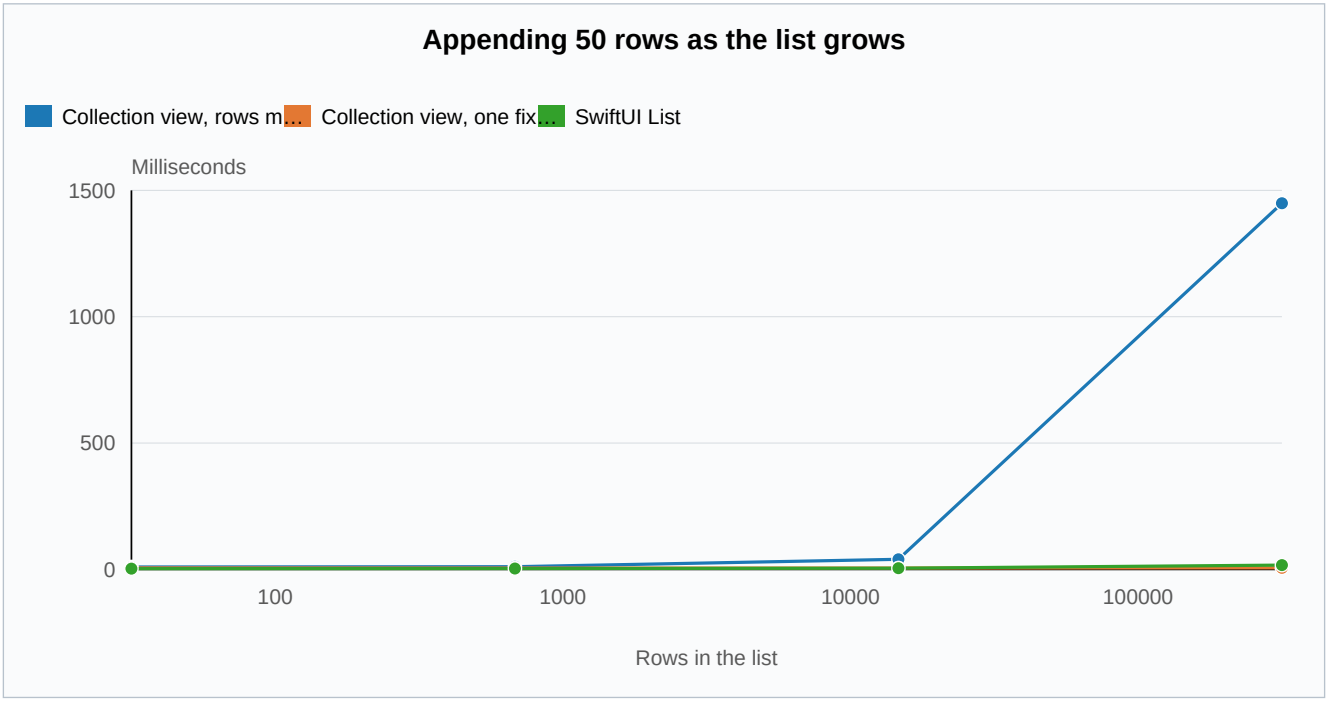
The same table view, fed by hand or by SwiftUI: the container does not change, the cost per change does.

Read the table in pairs. The two collection view columns are the same, so the cells are not the cost. The two columns on the right show the same container fed two ways, and it takes SwiftUI about 18 milliseconds at 100,000 rows to do what a hand-fed table does in well under one. At 10,000 rows the same update costs the `List` 4.7. So what SwiftUI adds per change grows with the list. Why, I did not measure. A cost proportional to the number of rows in its diff would look like this, and that is a guess.

And the table is not better at everything. One screen of scrolling takes the hand-fed table 5.7 milliseconds and the `List` 1.3. The `List` is not slow. It is a particular trade.

Where the self-sizing collection view loses

The collection view columns of that table, 1,468 milliseconds to show the list, are the biggest numbers in this post, and they are not about SwiftUI at all. They belong to a collection view that measures its own rows, using estimated heights. Appending 50 rows to it, in milliseconds:



The self-sizing layout pays for the size of the list. A fixed height does not, and a List only a little.

On an iPad I counted how many cells the layout asked to size themselves at first display. It was 36, whether the list had 100 rows or 100,000, while the time went from 4.7 to 1,063 milliseconds. So the growing cost is not in asking cells. It is in the layout. Give the collection view one fixed row height and the same append takes 5 or 6 milliseconds from 1,000 rows up to 100,000. The container is faster than the `List` only when you make the decision SwiftUI makes for you.

That is the shape of the whole thing. A hand-built container wins when you remove work, with one row height and updates in place. The `List` wins when your rows must measure themselves, because its table or collection view only measures what is on screen. Neither is "faster". They are two different feeds to the same kind of container, and the cost lives in the feed.

The iPad did not agree with the Mac

> **Read the iPad numbers as measurements, not as a curve**
> On a real iPad Pro, appending 50 rows to a list of 10,000 cost the `List` 74 milliseconds and the self-sizing collection view 14, which is the other way round from the Mac. Device runs wait for display frames, so some runs take a hundred milliseconds even for an operation that takes 0.2 when left alone. My first iPad numbers also did not reproduce after I fixed a leak in my own runner, and I do not know why. The mechanism, the container underneath, is the same. The timings are not the Mac's.

How I got one of these wrong

I should tell you about the table I got wrong, because it is the best reason to trust the rest. For a row that looks like a real forum post, with an avatar, an author line, one to twelve lines of text and a picture on one in four, the first version of my results said a `List` takes 3,031 milliseconds to show 100,000 of them, and I drew the opposite conclusion from the one I now hold.

The numbers were wrong because of my own harness. To draw a post I had put an `if` inside each row of the `List`, which made every row a conditional view, and the `List` several times slower. I only found it because a control I had promised in the protocol, rerunning older cells against the code from before the change, did not match for the `List`. I fixed the harness, measured every `List` cell again, and replaced them. With the corrected numbers, the `List` shows those 100,000 posts in 67 milliseconds.

The lesson is the one that sits under this whole post: when a number surprises you, suspect the instrument before the platform. It is the reason I read the classes two ways, and the reason I ran the probe on a plain table view first.

What I have not shown

> **Only List and Table were probed**
> What is under LazyVStack and LazyVGrid, I do not know.

- **Containers:** I did not point the probe at `LazyVStack` or `LazyVGrid`.
- **Machines:** one Mac, an iPhone simulator, a real iPhone and a real iPad. The real devices took a slightly different route to the same container.
- **Versions:** macOS 27.2.1, iOS 27.0 on the simulator and iOS 27.2.1 on the devices. The class names are private and can change in any release, so I say nothing about other versions.
- **Not looked at:** other list styles, sections, hierarchical lists, selection, edit mode, and what the render server does.

If the question is whether SwiftUI wraps the platform for everything it draws, I do not know yet. For lists and tables, I can show you the class names.

So, what changes

The things that go wrong with a `List` are the things that go wrong with a table view or a collection view, because that is what it is. If something is slow in a `List`, the answer is rarely "SwiftUI is slow". It is closer to "what am I asking a table view or a collection view to do, and how is SwiftUI translating it?" And the choice on an iPhone is not between UIKit and SwiftUI as containers. Both sides are a `UICollectionView`. The choice is who feeds it, and how its rows get their height.

Where the numbers come from

Every number here comes from my measured runs, which I describe in a longer article, "Is a Collection View Faster Than a SwiftUI List? I Measured.", coming next. The code that took them, the answer key and the raw data of every run are already in a public repository, ListBench. You do not have to take my word for any of it:

Run it yourself

```
$ git clone https://codeberg.org/Mihaela/ListBench.git
$ cd ListBench
$ swift run -c release ListBenchRun --out data.json --isolate --max-load 10
```

Same code, same answer key, your machine.