To nest, or not to nest? Nested data types in Polars

Daniel Finnan

daniel.finnan.fr

PyCon DE & PyData 2026 Darmstadt, 15 April 2026

About me

- 2nd year PhD, Lirsa laboratory, Conservatoire national des arts et métiers (CNAM), Paris, France
- Thesis on decentralized finance, specifically DEXs
  - Supervised by Dr. Iryna Veryzhenko & Dr. Elisa Darriet
- Coding primarily in Python & R (occasionally Rust & JavaScript)

What are Polars nested types?

- Vector-like containers for storing multiple values per row
- Contain the **same** data type

Why?

- Structure the relations in your data
- Express cardinality explicitly
- More intuitive, avoid duplication (?)

Introducing...

Basic list

```
import polars as pl

df = pl.DataFrame({
    "foo": [[1, 2], [3, 4, 5]],
    "bar": [["apple", "pear", "orange"], ["mango"]]
})
```

```
+-----------+-------------------------------+
| foo       | bar                           |
| ---       | ---                           |
| list[i64] | list[str]                     |
+=======================================+
| [1, 2]    | ["apple", "pear", "orange"]   |
|-----------+-------------------------------|
| [3, 4, 5] | ["mango"]                     |
+-----------+-------------------------------+
```

Characteristics

- `pl.List()` can be **variable-length**
- `pl.Array()` must be **fixed-length**
- Same `[]` syntax as Python lists, but these are **not** Python lists!
- Polars assumes `list[]` by **default** unless specified
- Namespaces `.list` & `.arr` provide access to methods

Back to the future

Historical context of lists & arrays

- Long-standing feature in Polars
- Under active development, for example:
  - `list.filter()` added as feature in May 2025 (22749)
  - `list.agg` & `arr.agg()` added as feature in Oct 2025 (24790)
- Other expressions, e.g. `arr.filter()`, in feature list

Toy example 1 - Lists

Prost!

Dataframe of German beers[1]

name	brewery	alcohol	energy	containers	ingredients
str	str	f64	list[i64]	list[f64]	list[str]
Naturtrübes ...	Braustüb'l	4.8	[45, 188]	[0.33, 0.5]	["water", "barley...
Dunkel	Grohe	5.1	[]	[]	["water", "barley...
Römer Pils	Binding	4.9	[40, 168]	[]	["water", "barley...
German Pale Ale	Braufactum	5.0	[42]	[]	["water", "barley...
Rauchweizen	Schlenkerla	5.2	[]	[0.5]	["water", "barley...
Leichtes Weizen	Aktienbraue...	2.8	[28, 118]	[0.5]	["water", "barley...
Dunkel	Maisel & Fr...	5.1	[40, 167]	[0.5]	["water", "barley...

Columns `energy`, `containers`, and `ingredients` use lists.

[1] Source: bier-universum

Toy example 1 - Lists

Queries

1. Find all wheat beers
2. Calculate units of alcohol for each container
3. Determine % of recommended calorie intake per container

Live coding!

Toy example 2 - Arrays

Punctuality!

Dataframe of European train punctuality[2]

```
trains = pl.DataFrame(
    {
        "operator": [
            "Deutsche Bahn",
            "SNCF",
            "SNCB",
            "SBB"
        ],
        # 2025 Jan - Dec
        "punctuality": [
            [90.4, 90.2, 90.1, 89.0, 89.2, 88.0, 87.4, 88.4, 86.4, 84.4, 84.5, 87.8],
            [87.2, 88.6, 89.7, 90.1, 89.7, 84.0, 83.6, 85.1, 85.2, 85.8, 84.3, 85.4],
            [91.3, 89.6, 91.9, 91.7, 91.5, 90.9, 93.4, 93.8, 92.0, 89.8, 91.6, 93.2],
            [94.4, 94.7, 94.9, 94.9, 94.1, 93.8, 94.1, 94.2, 93.0, 93.1, 93.0, 95.1]
        ]
    },
    schema={
        "operator": pl.String,
        "punctuality": pl.Array(pl.Float64, 12),
    }
)
```

[2]Source: Deutsche Bahn, SNCF, SNCB, SBB

Toy example 2 - Arrays

Descriptive statistics

```
desc_stats = trains.select(
    pl.col('operator'),
    mean = pl.col("punctuality").arr.mean(),
    std_dev = pl.col("punctuality").arr.std()
)
```

```
+---------------+-----------+----------+
| operator      | mean      | std_dev  |
| ---           | ---       | ---      |
| str           | f64       | f64      |
+=====================================+
| Deutsche Bahn | 87.983333 | 2.039979 |
| SNCF          | 86.558333 | 2.396004 |
| SNCB          | 91.725    | 1.298338 |
| SBB           | 94.108333 | 0.753728 |
+---------------+-----------+----------+
```

Toy example 2 - Arrays

Number of months below EU average of 87%[3]

```
eu_avg = trains.select(
    pl.col("operator"),
    less_87 = pl.col("punctuality").arr.eval(pl.element() < 87)
                                   .arr.count_matches(True)
)
# NOTE Not got filter for arr yet
```

```
+---------------+---------+
| operator      | less_87 |
| ---           | ---     |
| str           | u32     |
+=========================+
| Deutsche Bahn | 3       |
| SNCF          | 7       |
| SNCB          | 0       |
| SBB           | 0       |
+---------------+---------+
```

[3] Source: Euronews

Toy example 2 - Arrays

Mean across seasons

```
seasons = trains.select(
    pl.col("operator"),
    spring = (pl.col("punctuality").arr.get(2)
            + pl.col("punctuality").arr.get(3)
            + pl.col("punctuality").arr.get(4)) / 3,
    summer = (pl.col("punctuality").arr.get(5)
            + pl.col("punctuality").arr.get(6)
            + pl.col("punctuality").arr.get(7)) / 3,
    autumn = (pl.col("punctuality").arr.get(8)
            + pl.col("punctuality").arr.get(9)
            + pl.col("punctuality").arr.get(10)) / 3,
    winter = (pl.col("punctuality").arr.get(0)
            + pl.col("punctuality").arr.get(1)
            + pl.col("punctuality").arr.get(11)) / 3
)
# NOTE Array doesn't have gather yet
```

Toy example 2 - Arrays

Mean across seasons again! - *with lists can specify seasons in column, and use* `gather()`

```
trains_list = trains.select(
    pl.col("operator"),
    pl.col("punctuality").arr.to_list(), # Equivalent .cast(pl.List)
    season = [[2, 3, 4], [5, 6, 7], [8, 9, 10], [0, 1, 11]]
).select(
    pl.col("operator"),
    spring = pl.col("punctuality").list.gather(
                pl.col("season").list.gather(0).explode()
        ).list.eval(pl.element().mean()).explode(),
    summer = pl.col("punctuality").list.gather(
                pl.col("season").list.gather(1).explode()
        ).list.eval(pl.element().mean()).explode(),
    autumn = pl.col("punctuality").list.gather(
                pl.col("season").list.gather(2).explode()
        ).list.eval(pl.element().mean()).explode(),
    winter = pl.col("punctuality").list.gather(
                pl.col("season").list.gather(3).explode()
        ).list.eval(pl.element().mean()).explode()
)
```

Takeaways

- Punctuality data may be better represented in panel format

```
+---------------------+---------------+------+------+------+
| month               | Deutsche Bahn | SNCF | SNCB | SBB  |
| ---                 | ---           | ---  | ---  | ---  |
| datetime            | f64           | f64  | f64  | f64  |
+=============================================================+
| 2025-01-01 00:00:00 | 90.4          | 87.2 | 91.3 | 94.4 |
| 2025-02-01 00:00:00 | 90.2          | 88.6 | 89.6 | 94.7 |
| ...                 | ...           | ...  | ...  | ...  |
+---------------------+---------------+------+------+------+
```

- Depending on data format, list & array syntax can be expressive or inscrutable series of methods on namespaces
- Ability to work across element-wise across columns crucial to usefulness of nested types
- With current state of play, what do nested types mean for Polars storage, performance and query structure? *Let's find out...*

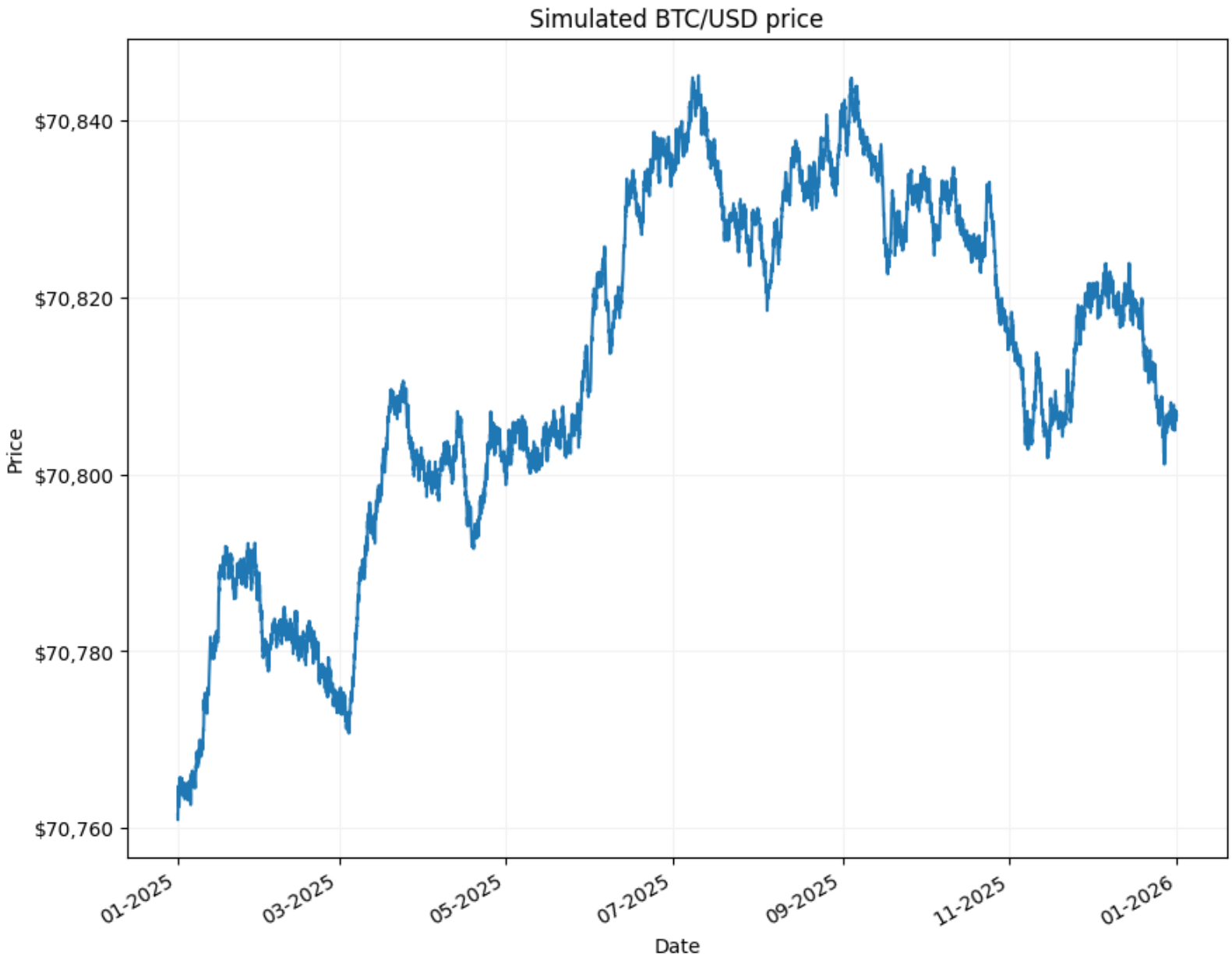
Simulated Limit Order Books

- Example taken from work creating features from real world order books from crypto exchanges, economic design: $[(P_i^{(b)}, V_i^{(b)}), (P_i^{(a)}, V_i^{(a)})]$

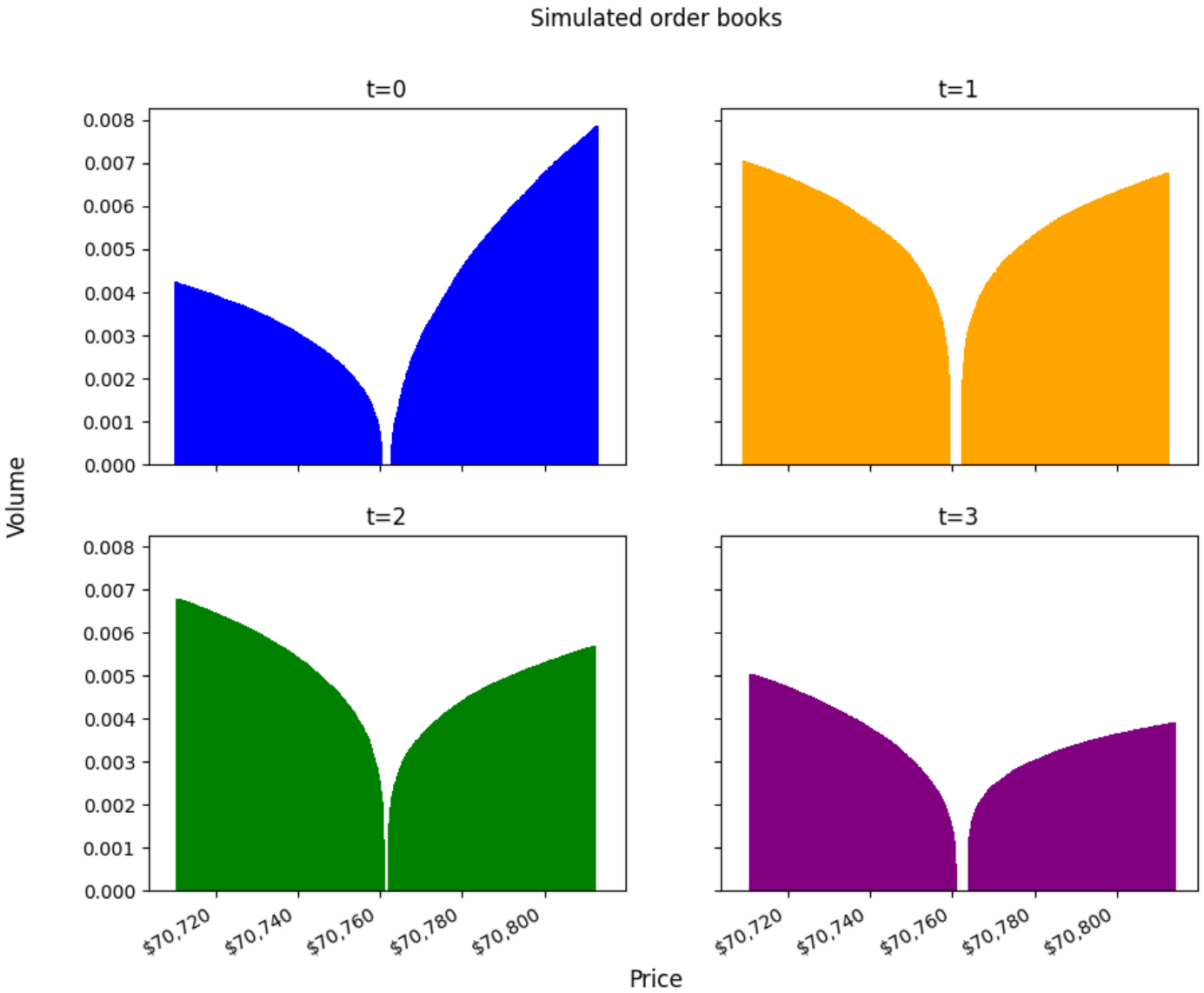
- Simulated with `numpy` (reproducible)
  1. Random walk, hourly observations, 1 year (8,760)
  2. Order book snapshots, 5000 levels on bid and asks
  3. Spread, volume and shape of book generated randomly

  Goal - Compare storage, performance and query structure
  1. No nesting
  2. Flat array
  3. Nested array
  4. Flat list
  5. Nested list

Dataset - Random walk



Dataset - Order book snapshots



Data structure

No nesting

last_update_id	timestamp	bids_p	bids_v	asks_p	asks_v
i64	i64	f64	f64	f64	f64
55069433	1735686000	70710.16	0.004232	70762.92	0.000101

Flat array

last_update_id	timestamp	bids	asks
i64	i64	array[f64, 2]	array[f64, 2]
55069433	1735686000	[70710.16, 0.004232]	[70762.92, 0.000101]

Nested array

last_update_id	timestamp	bids	asks
i64	i64	array[f64, (5000, 2)]	array[f64, (5000, 2)]
55069433	1735686000	[[70710.16, 0.004232], [70710....	[[70762.92, 0.000101], [70762....

Data structure

Flat list

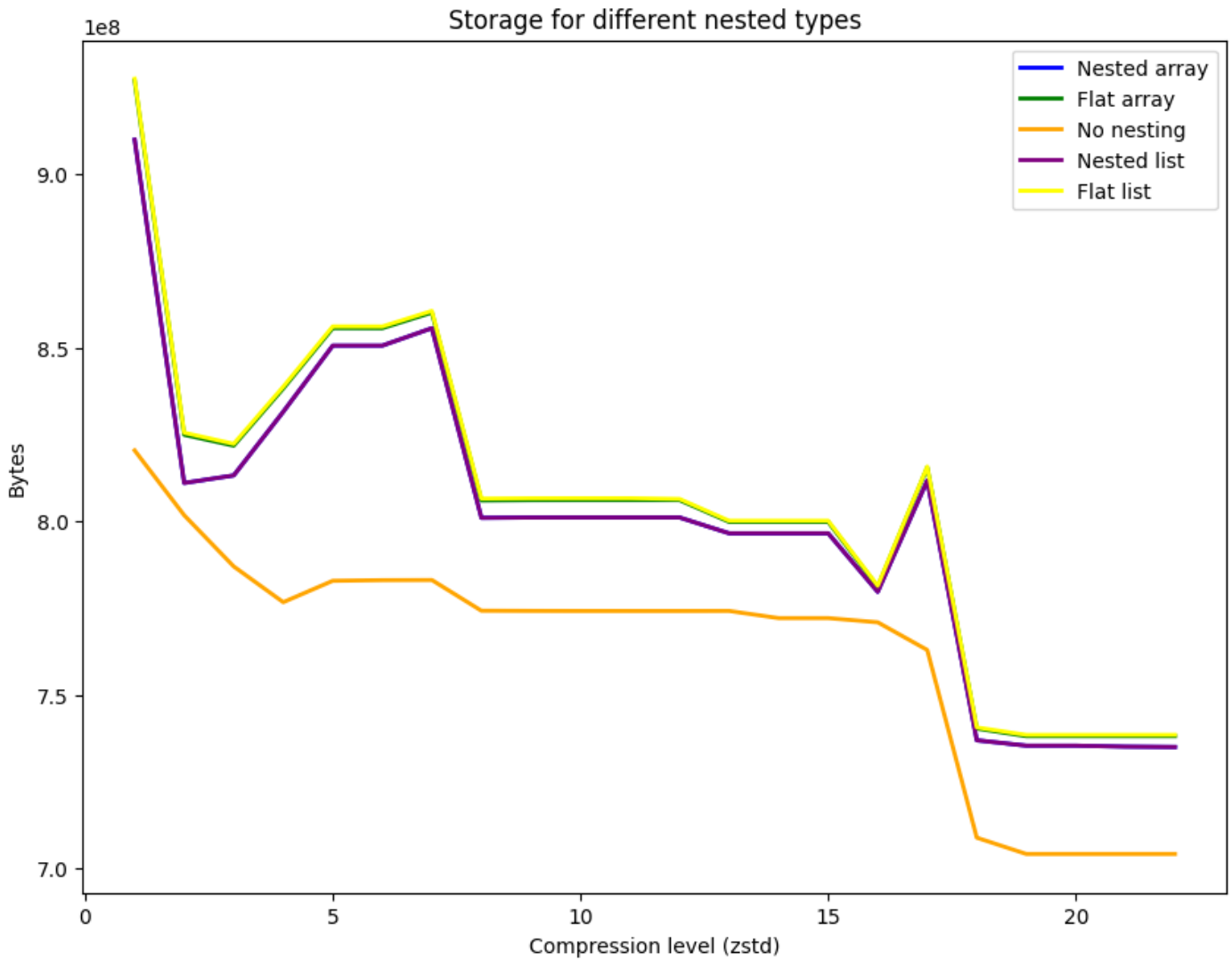
last_update_id	timestamp	bids	asks
i64	i64	list[f64]	list[f64]
55069433	1735686000	[70710.16, 0.004232]	[70762.92, 0.000101]

Nested list

last_update_id	timestamp	bids	asks
i64	i64	list[list[f64]]	list[list[f64]]
55069433	1735686000	[[70710.16, 0.004232], [70710....	[[70762.92, 0.000101], [70762....

Storage

"The gigabyte scrooge"



Queries

Straightforward calculations for each observation:

1. Mid-price & spread - *mean of max bid & min ask, difference between max bid & min ask*
2. Total queue imbalance - *total volume of bids & asks weighted, indicating buying/selling pressure*
3. Bid/ask depth at given level (ask @ 500) - *sum of volume within interval*

Query 1: Mid-price & spread

Nested list & nested array

```
select()...arr.agg(pl.element().arr.get()...)
```

Flat list & flat array

```
group_by().agg()...arr.get()
```

No nesting

```
group_by().agg()
```

Note: list & array queries are the same, except for the namespace

Query 2: Total queue imbalance

Nested list & nested array

```
select()...arr.agg()...arr.get()
with_columns()
```

Flat list & flat array

```
group_by().agg()...arr.get()
select()
```

No nesting

```
group_by()...agg()
select()
```

Query 3: Ask depth at 500 level

Nested list & nested array

```
with_columns()...arr().agg()...arr().get()...
with_columns()...
explode()...filter()...arr.get()...
group_by()...agg()...arr.get()
```

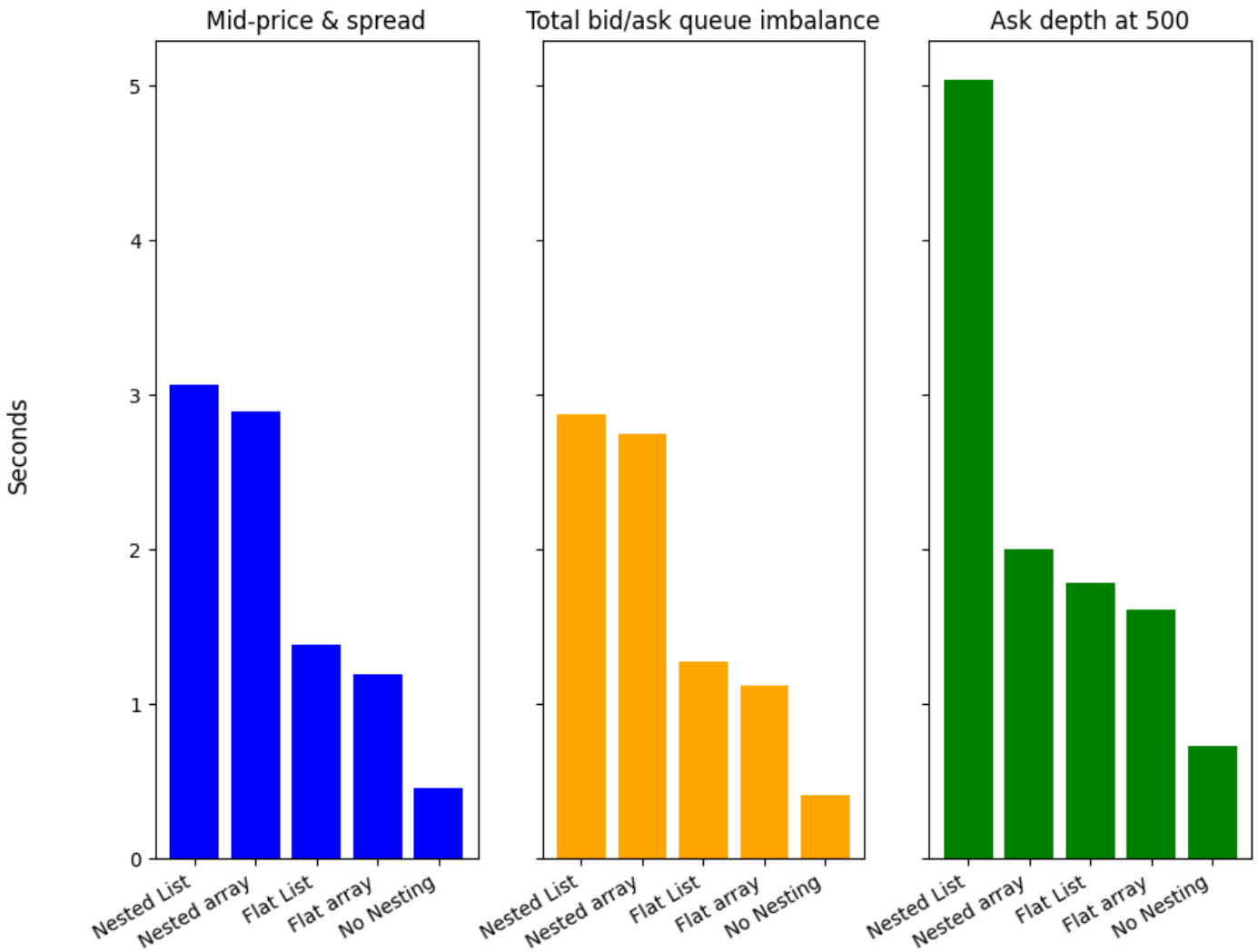
Flat list & flat array

```
with_columns()...arr.get()...
with_columns()...
filter()...arr.get()
group_by()...agg()...arr.get()
```

No nesting

```
group_by()...agg()...
with_columns()...
join()...
filter()...
group_by()...agg()...
```

Performance



Benchmarks: HP laptop with Intel i5 12th generation CPU, 16GB RAM, SSD, 100 repetitions, min. value

Conclusion

- Nested types add storage overhead
- Query structure can become inscrutable
- Not nesting has much better performance, despite needing to use `group_by()` & `join()`
- Are the gains in relational structure worth it?
- Would further development of `list` & `arr` namespaces make any difference?
- User-defined Python functions, e.g. via `Numba`, or Rust plugins, offer an alternative