ClickHouse

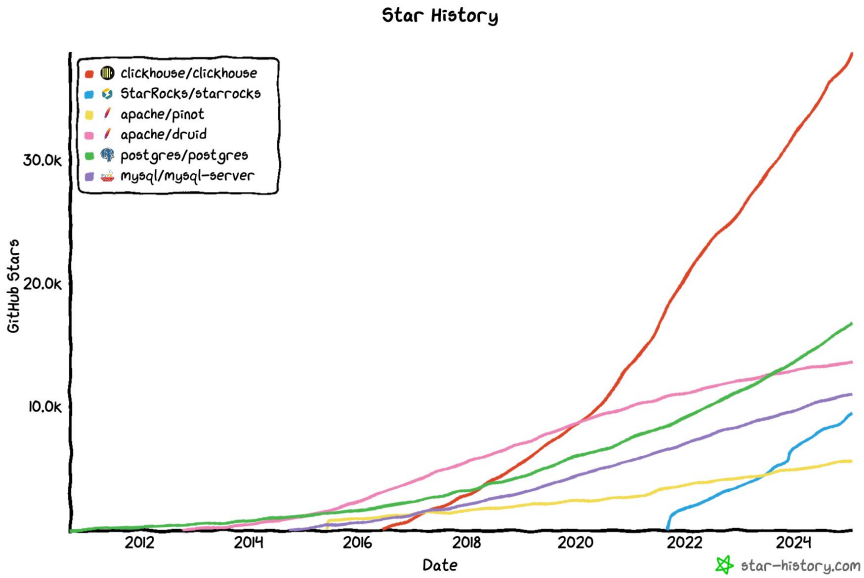
How we Built a New Powerful JSON Data Type for ClickHouse

1 Feb 2025

Pavel Kruglov
Robert Schulze

What is ClickHouse?

- A relational, columnar, shared-nothing, eventually consistent database for analytics.
- Goal: super-fast 🚀 and scalable analytics over tables with trillions of rows.
- Open source (Apache 2.0), built in C++, runs on anything from Raspberry Pi to clusters with hundreds of nodes.
- Self-managed or ClickHouse Cloud, a database-as-a-service (DBaaS).



ClickHouse

Row-wise vs. Column-wise Storage

Country	Product	Sales
GB	Lambda	350
FR	Kappa	400
US	Iota	1300

	Row-wise	Column-wise
		Column 1 GB
	Lambda	FR
	350	US
Row 2	FR	*Column 2* Lambda
	Kappa	Kappa
	400	Iota
Row 3	US	*Column 3* 350
	Iota	400
	1300	1300

Best suited for	Single-row operations	Column scans, aggregation

What is JSON?

- Lingua franca for semi-structured and unstructured data.
- Roots in JavaScript, found broad adoption as a data exchange format.
- Human-readable and machine-parseable.
- Schemaless!

```
{
  "first_name": "Juri",
  "last_name": "Gagarin",
  "is_alive": no,
  "age": 34,
  "born": {
      "country": "Soviet Union",
      "city": "Klushino"
  },
  "awards": [“Hero of the Soviet Union”
            ,”Order of Lenin”,
            ,”Pilot-Cosmonaut of the USSR”]
}
```

Databases vs. JSON

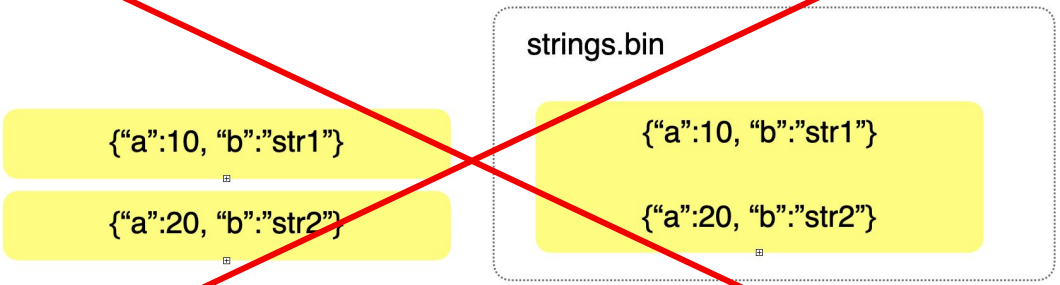
Problem: ⚡ Data models don’t match ⚡

- ClickHouse: Relational tables with a fixed schema (especially: fixed data types).
- JSON: anything and everything (schemaless).

```
{
  "first_name": "Juri",
  "last_name": "Gagarin",
  "is_alive": no,
  "age": 34,
  "born": {
      "country": "Soviet Union",
      "city": "Klushino"
  },
  "awards": [“Hero of the Soviet Union”
            ,”Order of Lenin”,
            ,”Pilot-Cosmonaut of the USSR”]
}
```

How Not to Do it …

- Serialize each JSON document.
- Dump serialized representation into a string column.



Excessive deserialization costs! 🐌

A Better Approach

- View JSON document as a set of paths.
- Store every path as a column.

```
{
  "first_name": "Juri",
  "last_name": "Gagarin",
  "is_alive": no,
  "age": 34,
  "born": {
      "country": "Soviet Union",
      "city": "Klushino"
  },
  "awards": [“Hero of the Soviet Union”
            ,”Order of Lenin”,
            ,”Pilot-Cosmonaut of the USSR”]
}
```

	{
first_name	"first_name": "Juri",
last_name	"last_name": "Gagarin",
is_alive	"is_alive": no,
age	"age": 34,
	"born": {
born.country	"country": "Soviet Union",
born.city	"city": "Klushino"
	},
awards	"awards": [“Hero of the Soviet Union” ,”Order of Lenin”, ,”Pilot-Cosmonaut of the USSR”]
	}

```
{
  "first_name": "Neil",
  "last_name": "Armstrong",
  "is_alive": no,
  "age": 82,
  "born": {
      "country": "United States",
      "city": "Wapakoneta"
  },
  "awards": [“Presidential Medal of Freedom”
            ,”NASA Distinguished Service Medal”]
}
```

- Maps naturally to relational model.
- Apply ClickHouse’s super-fast filtering and aggregation capabilities to JSON paths.

Easier said than done …

Challenge 1:

- Documents share no common paths.
- Exploding path count, column avalanche 😱

Challenge 2:

- Shared path but different types.
- No equivalent database construct. 😰

Challenge 3:

- Paths can vary wildly in their frequency.
- Rare paths cause “sparse” columns. 🫣

```
{
  "first_name": "Juri",
  "last_name": "Gagarin"
}
```

```
{
  "breed": "Husky",
  "activity": "Barking"
  "best_friend": "Human"
}
```

```
{
  "food": "Apple",
  "health_score": 8,
}
```

```
{
  "food": "Icecream",
  "health_score": “unhealthy”,
}
```

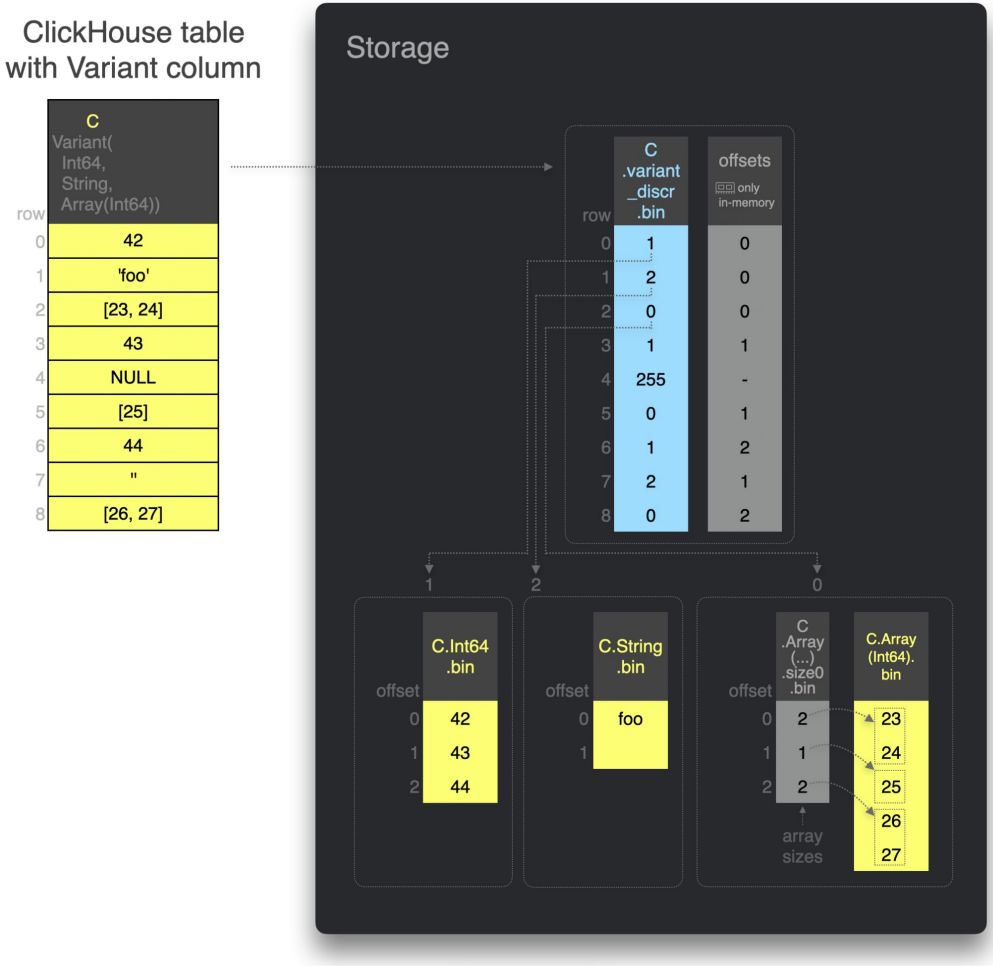
Building Block 1: Variant Data Type

- Formally: `Variant(T1, T2, … Tn)`
- One out of multiple data types or `NULL`.

Implementation based on two structures:

- A *discriminator column*, representing the value's type.
- An *offset structure*, representing the value's position in the sub-type column.

Solves challenges 2 and 3!



ClickHouse

Building Block 2: Dynamic Data Type

Same as `Variant` type BUT without need to specify sub-types in advance.

- Implementation: same as `Variant`.
- Has an additional file `.dynamic_structure`.

ClickHouse

Building Block 2: Dynamic Data Type

- Optionally: upper limit on sub-type columns.
- Remaining values stored as `<data_type><value>` pairs.

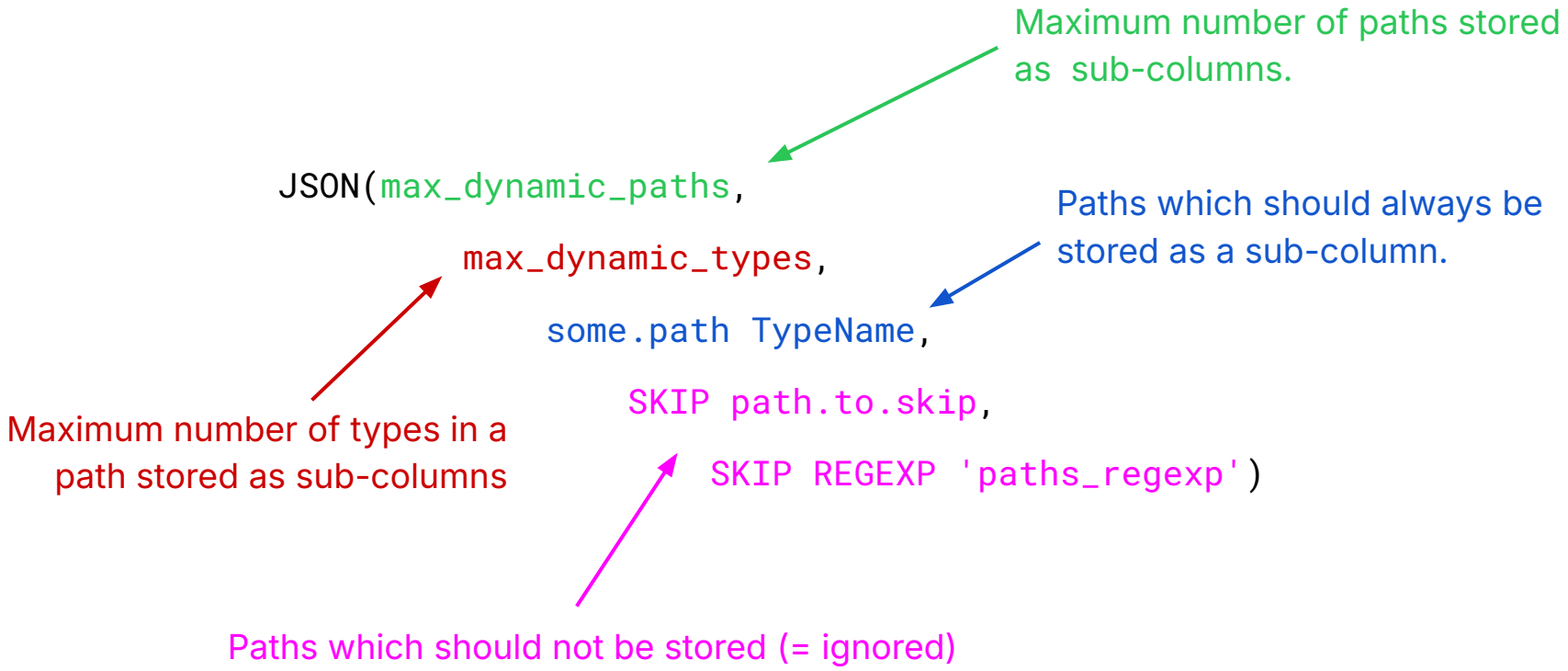
Solves challenge 1!

ClickHouse table with Dynamic column and max_types parameter

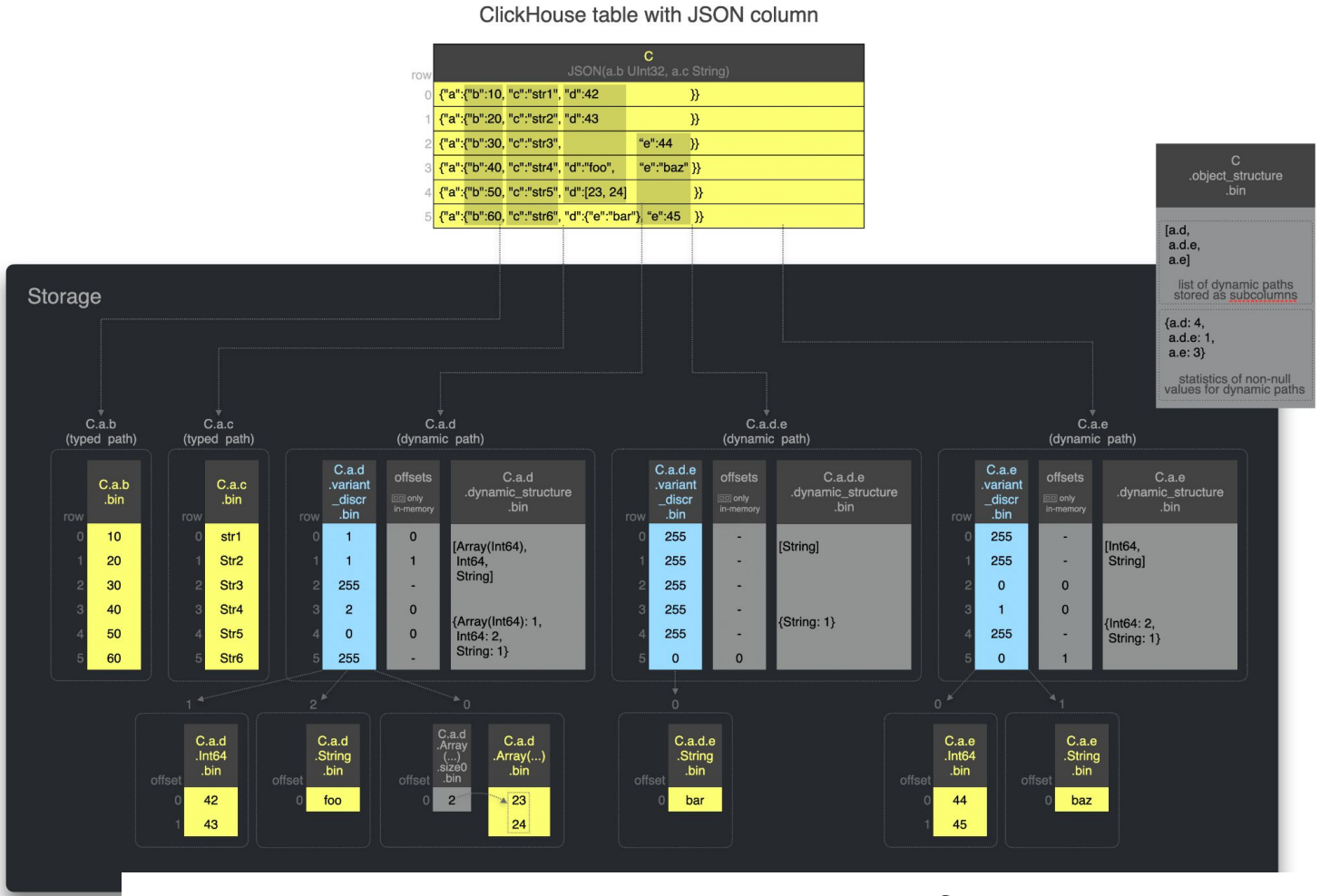
row	C Dynamic (max_types=3)
0	42
1	'foo'
2	[23, 24]
3	'2020-01-01'
4	43
5	NULL
6	FALSE
7	['bar', 'baz']
8	[25]
9	'2020-01-02'
10	44
11	TRUE
12	''
13	[26, 27]

ClickHouse

Bringing it all together: The JSON Data Type

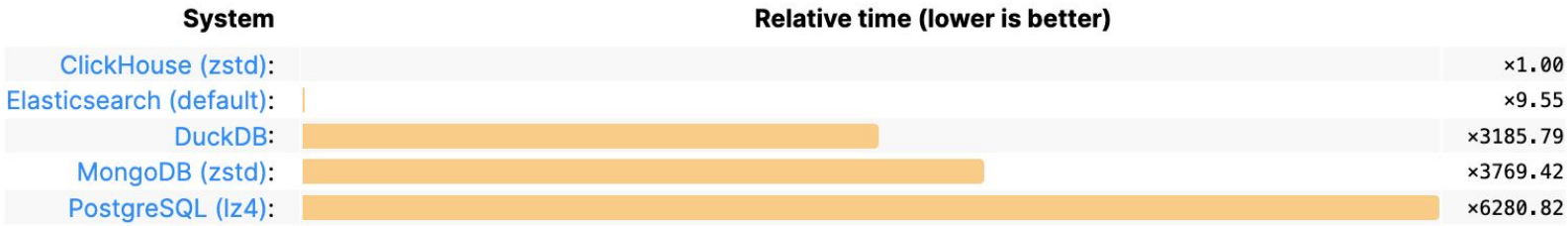


ClickHouse



https://clickhouse.com/blog/a-new-powerful-json-data-type-for-clickhouse

JSONBench (jsonbench.com)



Detailed Comparison

	ClickHouse (zstd)	Elasticsearch (default)	DuckDB	MongoDB (zstd)	PostgreSQL (lz4)
Data size:	92.42 GiB (×1.00)	454.54 GiB (×4.92)	440.14 GiB (×4.76)	144.72 GiB (×1.57)	579.23 GiB (×6.27)
Data quality:	999999258	999999153	974400000	893632990	804000000
☑ Q1.	0.394s (×1.00)	3.026s (×7.51)	3731.713s (×9236.94)	1664.900s (×4121.06)	3907.110s (×9671.09)
☑ Q2.	5.632s (×1.00)	27.380s (×4.85)	3737.248s (×662.40)	☠	32621.100s (×5781.83)
☑ Q3.	2.466s (×1.00)	24.787s (×10.01)	3729.542s (×1506.28)	5451.420s (×2201.71)	4274.220s (×1726.26)
☑ Q4.	0.596s (×1.00)	8.731s (×14.42)	3735.854s (×6164.79)	1674.570s (×2763.33)	6292.720s (×10384.04)
☑ Q5.	0.637s (×1.00)	9.733s (×15.06)	3736.934s (×5775.80)	1698.160s (×2624.68)	6309.170s (×9751.44)

ClickHouse

Summary

- ClickHouse now speaks JSON!
- The implementation is easy to use, flexible and extremely fast 🚀.
- JSON support is beta and will be GA in 2025.

ClickHouse

Dinner & Drinks

FOSDEM Dinner with ClickHouse

Brussels Belgium, Saturday Feb 1st from 19:00-23:00pm

L'Ultime Atome
Rue Saint-Boniface 14, 1050 Ixelles, Belgium