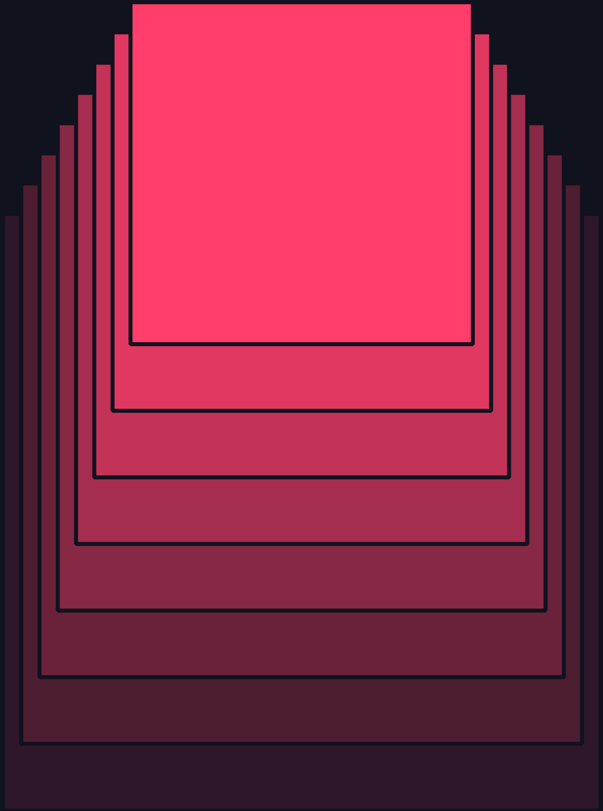
DATA+AI SUMMIT
BY databricks

Variant Data Type

Making Semi-Structured Data Fast and Simple



Gene Pang, Chenhao Li
2024-06-13

OUTLINE

- Motivation
- Variant Data Type Overview
- Using Variant
- Deep Dive: Variant Binary Format
- Performance

Semi-Structured Data in the Lakehouse

- Semi-structured data is partially structured
  - Doesn't fully adhere to relational table model
  - Schema may be unknown, or incompatible, or evolving
- JSON is very popular semi-structured data format
  - Flexible, and supported in most programming languages

How do we store and process semi-structured data in the lakehouse?

Schema Inference

Option 1

- On ingestion, read data and infer schema (structs, arrays, scalars, etc.)
- Read queries use the relational schema
- Performance same as structured/relational data

Challenges with Schema Inference

- Inference must determine a schema that works with all the data
    - If data is diverse, can produce huge, but sparse schemas
- Schema enforcement is strict
    - Incoming data must be compatible with schema
    - Accessing missing field may produce exceptions

TOO STRICT

Treat Data as String

Option 2

- On ingestion, data is stored as string
    - No schema enforcement on ingestion
- Read queries parse the string during execution
- Maximum flexibility for any data

Challenges with using String type

- Parsing String in queries is slow
  - Typically, data is read more than it is written, so expensive parsing is repeated for every query

TOO SLOW

Your Choices

TOO STRICT

TOO SLOW

VARIANT: OPEN, FLEXIBLE, PERFORMANT TYPE FOR SEMI-STRUCTURED DATA

Variant Data Type

Open, Flexible, Performant Data Type for Semi -Structured Data

Open

- Spark & Delta data type
- Spark Variant expressions
- Open-source library for Variant binary encoding

Flexible

- No schema on ingestion
- Schema-on-read access

Performant

- Offset-based binary encoding speeds up navigation

Open Source

- Code merged for Spark 4.0 and Delta Lake 4.0
    - Released in Spark 4.0 PREVIEW and Delta Lake 4.0 PREVIEW
- Open source library for encoding and decoding Variant binary format
    - Make it easier for other projects to support Variant
- Future Variant support for other engines and table formats

Variant Expressions

New/Updated Expressions

- `parse_json`: Constructs a VARIANT from a JSON string
- `to_json`: Converts a VARIANT to a JSON string
- `variant_get`: Extracts the path of a specified type, from the VARIANT
- `cast`: Cast to and from VARIANT
- `schema_of_variant`: Returns the schema string of a VARIANT
- `variant_explode`: Table function for un-nesting a VARIANT

Variant Usage Examples

```sql
-- Create a table with a Variant column
CREATE TABLE T (variant_col Variant)

-- Use PARSE_JSON() to convert JSON string to Variant
SELECT PARSE_JSON(json_str_col) variant_col FROM T

-- Variant path navigation
SELECT variant_col:a.b.c::int, variant_col:arr[1].field::double FROM T

-- Un-nest Variant objects
SELECT key, value FROM T, LATERAL VARIANT_EXPLODE(T.variant_col:obj)
```

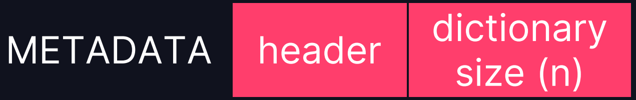
VARIANT BINARY FORMAT DEEP-DIVE

Variant Binary Format

- Binary encoding (instead of plain text) to represent semi-structured data
- Uses offsets to enable skipping for faster navigation
- 2 binary "blobs" used to encode
  - METADATA holds dictionary of keys
  - VALUE holds Variant data and structure, referring to dictionary in METADATA
- On-disk and in-memory binary formats are identical
- Typically smaller size than String representation

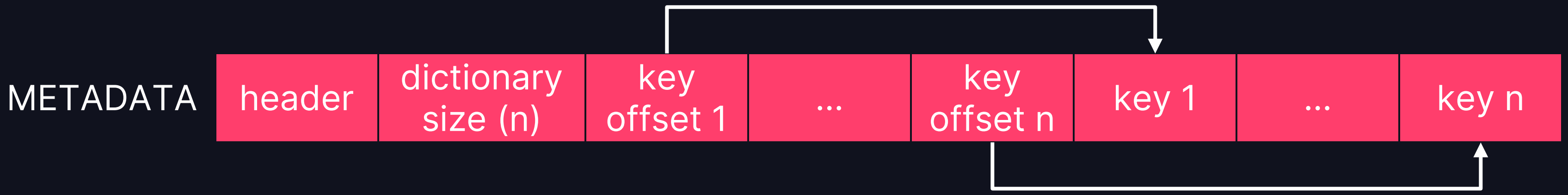
Variant Binary Format

Simplified Example of a Variant Object



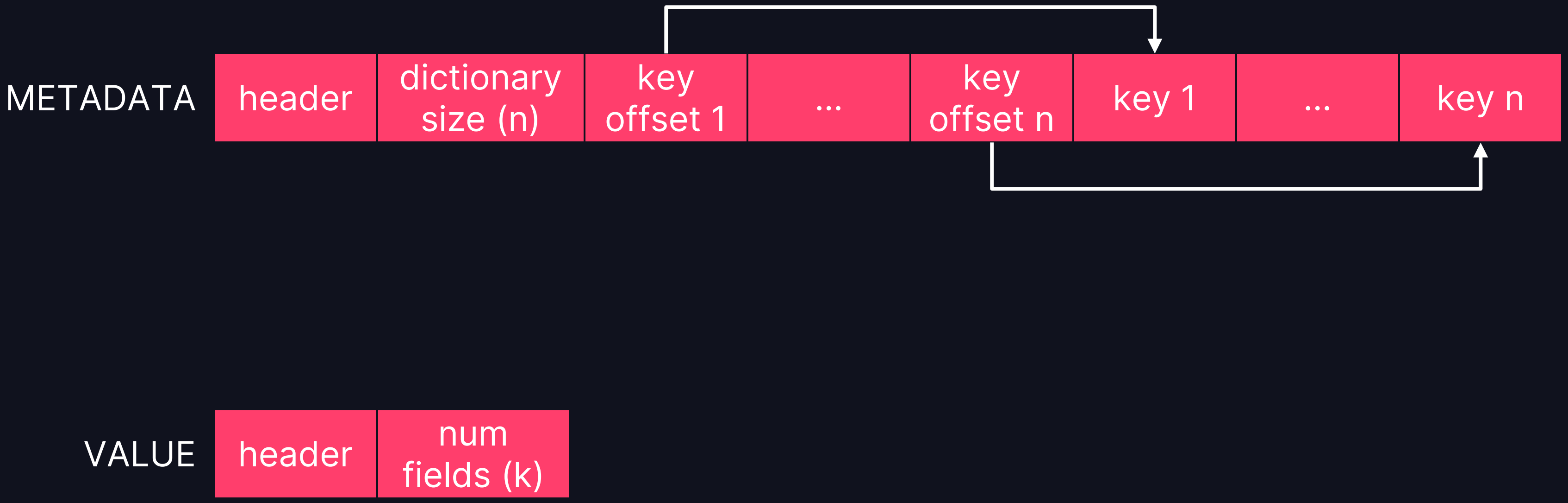
Variant Binary Format

Simplified Example of a Variant Object



Variant Binary Format

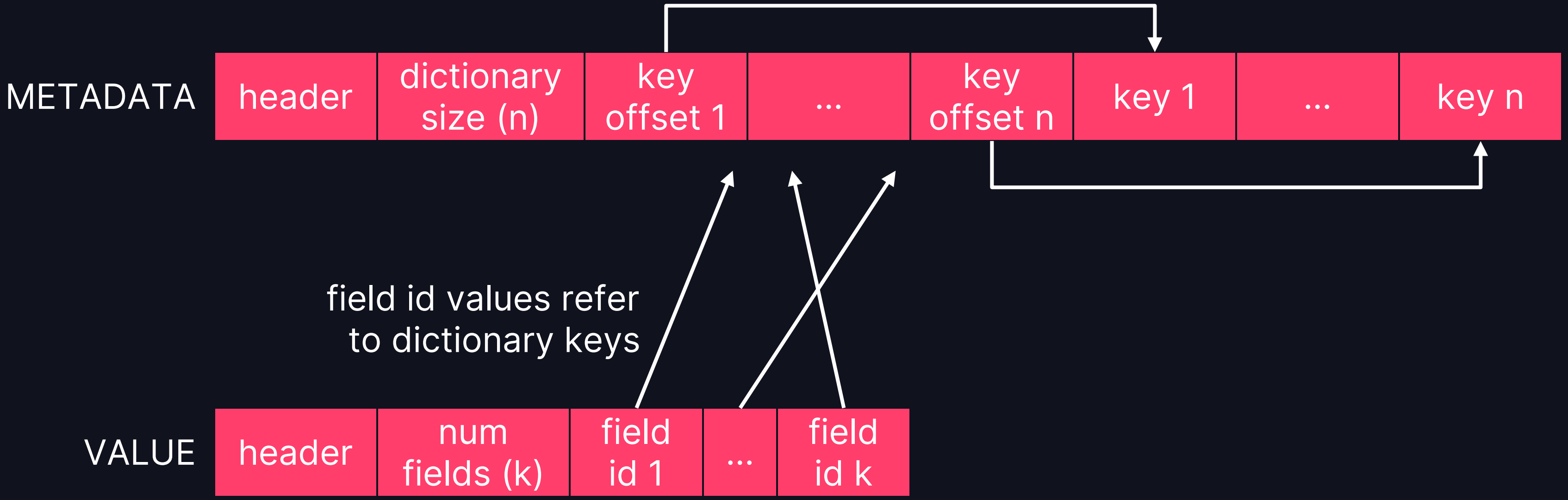
Simplified Example of a Variant Object



METADATA	header	dictionary size (n)	key offset 1	...	key offset n	key 1	...	key n
VALUE	header	num fields (k)						

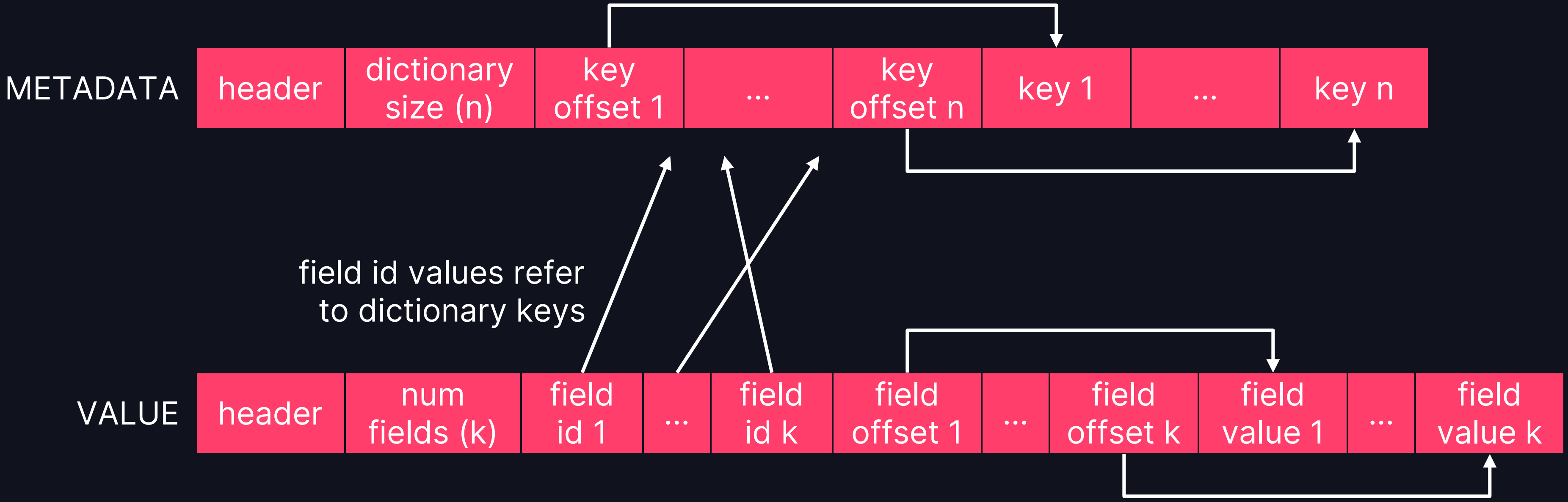
Variant Binary Format

Simplified Example of a Variant Object



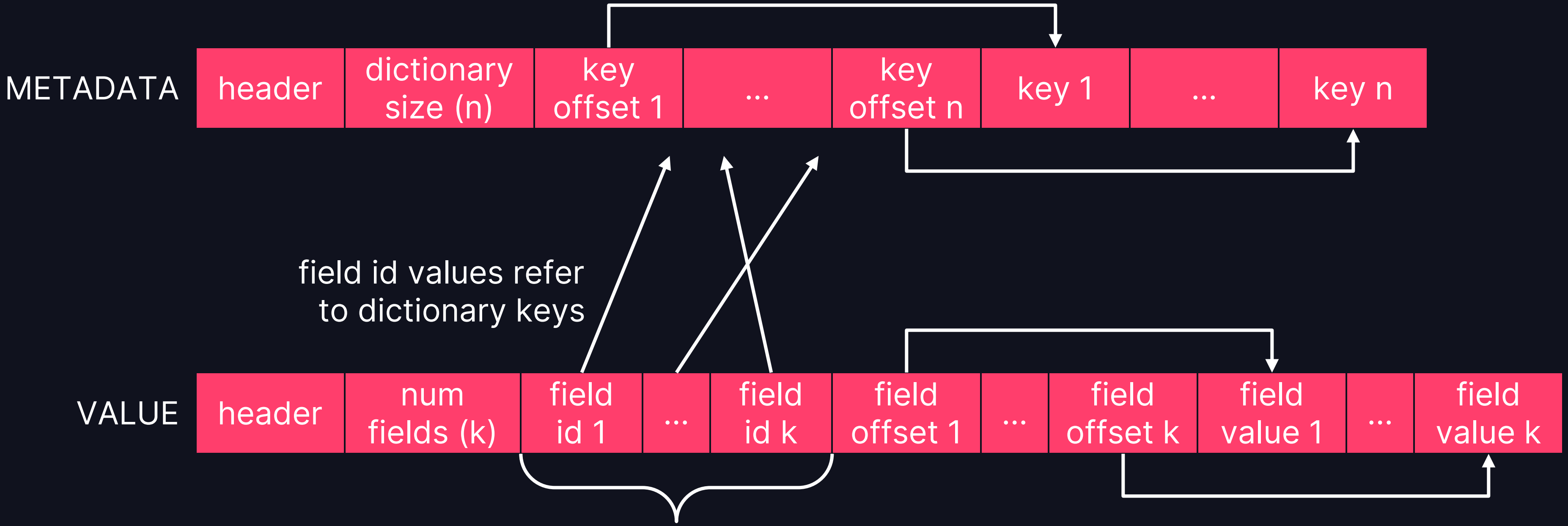
Variant Binary Format

Simplified Example of a Variant Object



Variant Binary Format

Simplified Example of a Variant Object



METADATA

header | dictionary size (n) | key offset 1 | ... | key offset n | key 1 | ... | key n

VALUE

header | num fields (k) | field id 1 | ... | field id k | field offset 1 | ... | field offset k | field value 1 | ... | field value k

field id values refer to dictionary keys

fields are lexicographically ordered by dictionary key

Variant Binary Format

Example of Key Deduplication

JSON String	[{“key1”: 1, “key2”: 2}, {“key1”: 3, “key2”: 4}]
VALUE	[{0: 1, 1: 2}, {0: 3, 1: 4}]
METADATA	[“key1”, “key2”]

Variant Binary Navigation

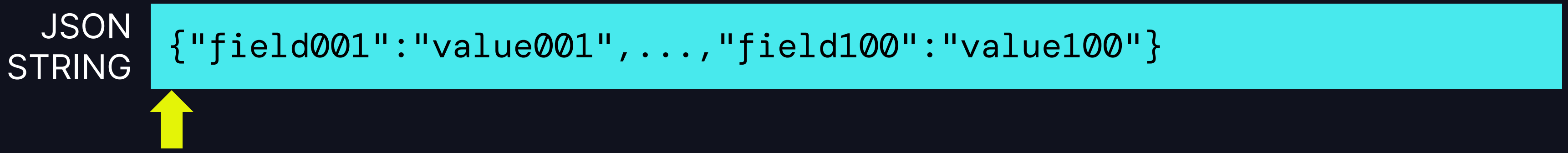
Simplified Example of Navigation

JSON
STRING

```
{"field001":"value001",...,"field100":"value100"}
```

Variant Binary Navigation

Simplified Example of Navigation

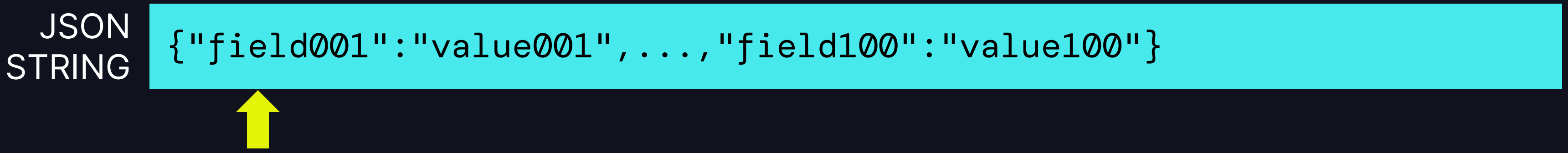


JSON STRING

```
{"field001":"value001",...,"field100":"value100"}
```

Variant Binary Navigation

Simplified Example of Navigation



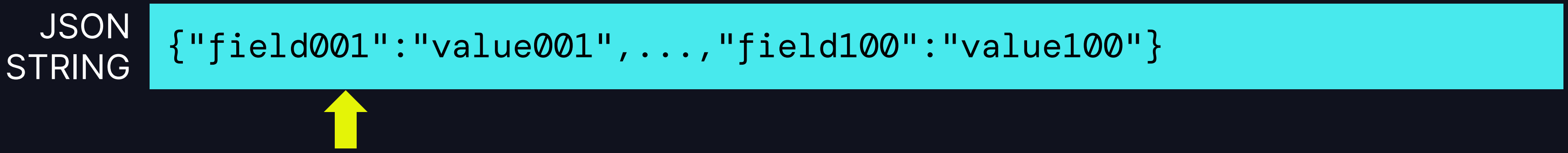
JSON STRING

```
{"field001":"value001",...,"field100":"value100"}
```

Sequential and linear processing of JSON string

Variant Binary Navigation

Simplified Example of Navigation



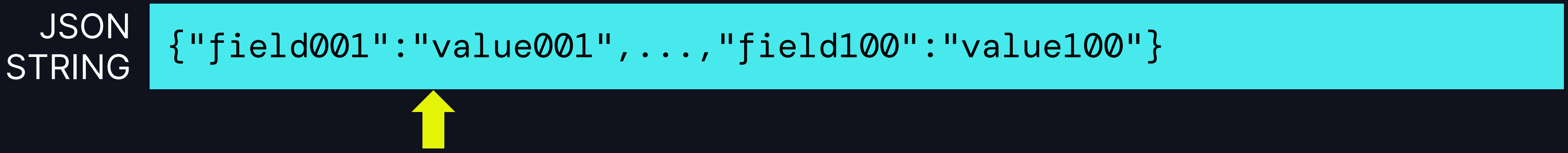
JSON STRING

```
{"field001":"value001",...,"field100":"value100"}
```

Sequential and linear processing of JSON string

Variant Binary Navigation

Simplified Example of Navigation



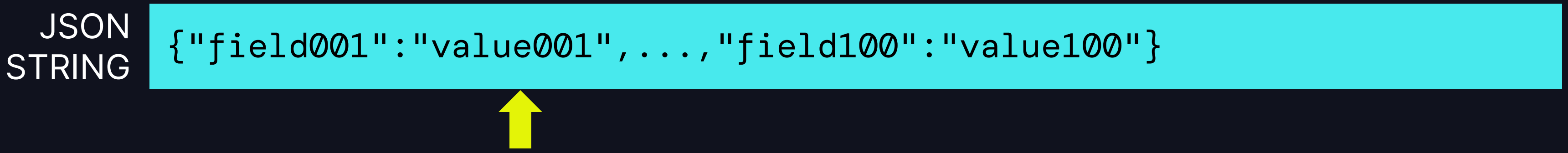
JSON STRING

```
{"field001":"value001",...,"field100":"value100"}
```

Sequential and linear processing of JSON string

Variant Binary Navigation

Simplified Example of Navigation



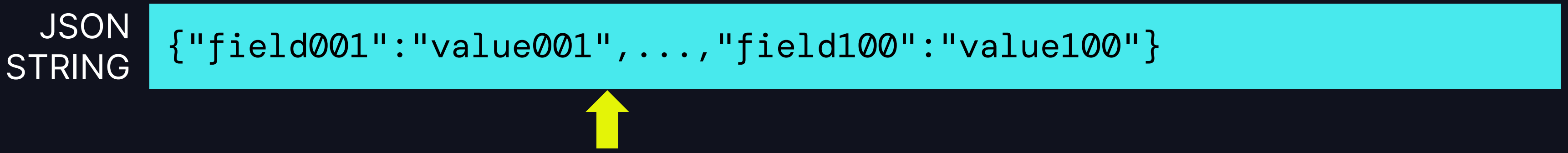
JSON STRING

```
{"field001":"value001",...,"field100":"value100"}
```

Sequential and linear processing of JSON string

Variant Binary Navigation

Simplified Example of Navigation



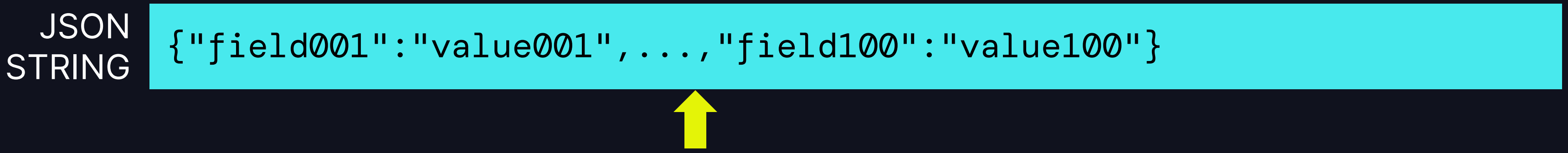
JSON STRING

```
{"field001":"value001",...,"field100":"value100"}
```

Sequential and linear processing of JSON string

Variant Binary Navigation

Simplified Example of Navigation



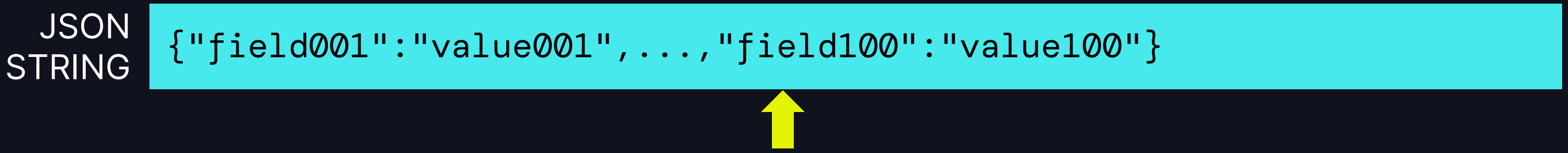
JSON STRING

```
{"field001":"value001",...,"field100":"value100"}
```

Sequential and linear processing of JSON string

Variant Binary Navigation

Simplified Example of Navigation



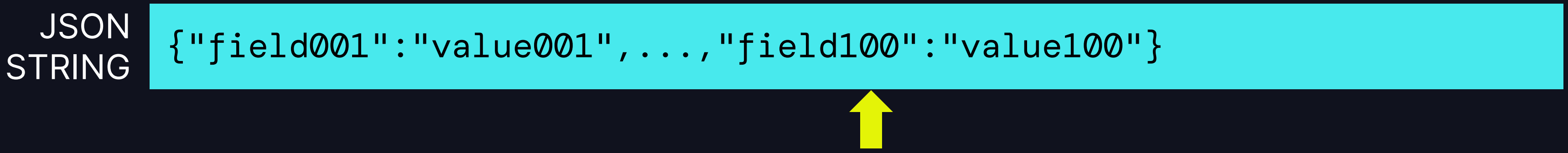
JSON STRING

```
{"field001":"value001",...,"field100":"value100"}
```

Sequential and linear processing of JSON string

Variant Binary Navigation

Simplified Example of Navigation



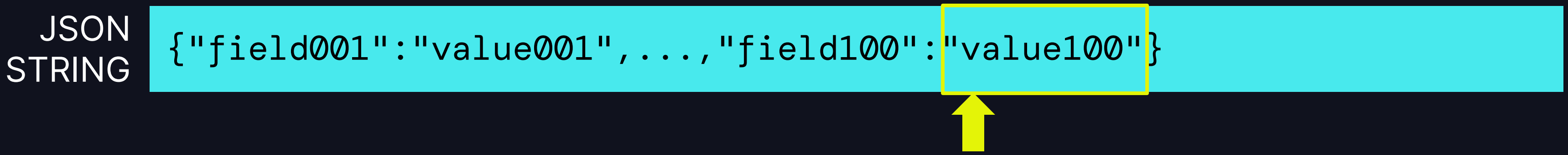
JSON STRING

```
{"field001":"value001",...,"field100":"value100"}
```

Sequential and linear processing of JSON string

Variant Binary Navigation

Simplified Example of Navigation



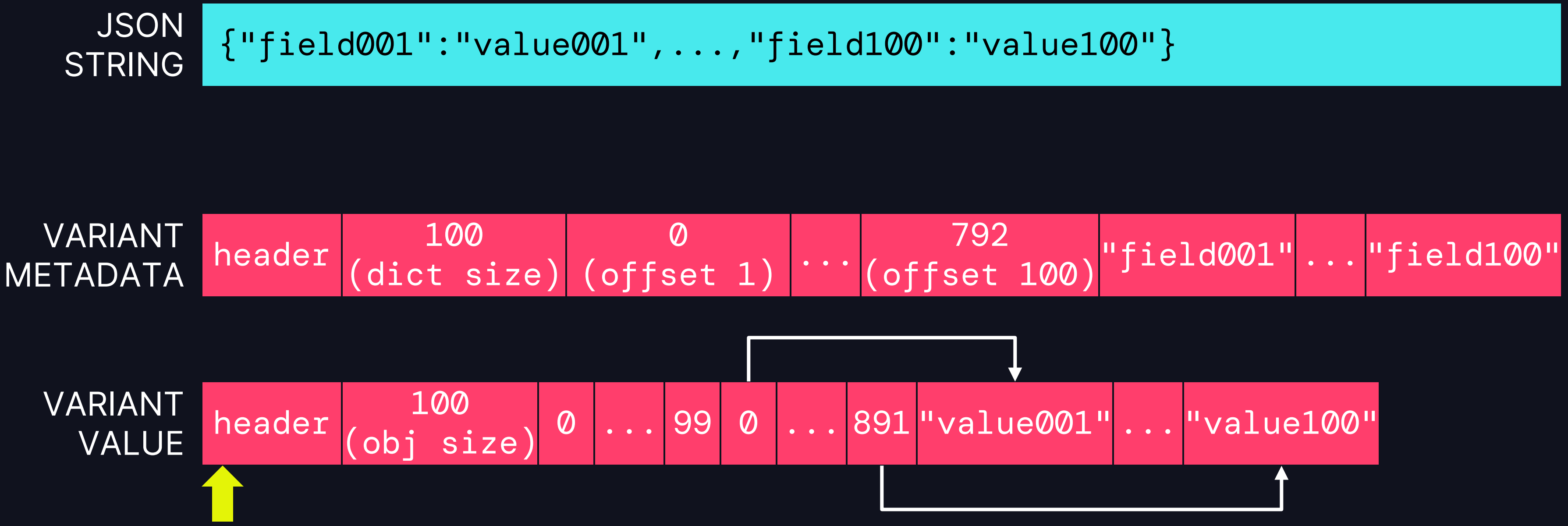
JSON STRING

```
{"field001":"value001",...,"field100":"value100"}
```

Sequential and linear processing of JSON string

Variant Binary Navigation

Simplified Example of Navigation



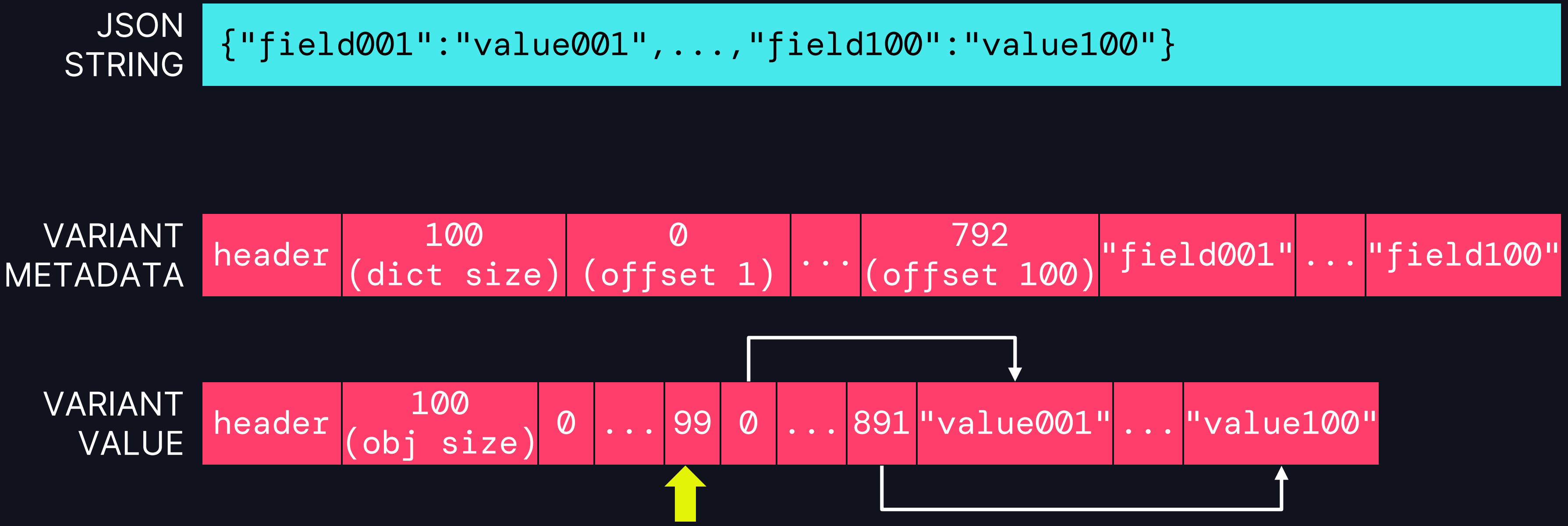
JSON STRING	{"field001":"value001",...,"field100":"value100"}

VARIANT METADATA	header	100 (dict size)	0 (offset 1)	...	792 (offset 100)	"field001"	...	"field100"

VARIANT VALUE	header	100 (obj size)	0	...	99	0	...	891	"value001"	...	"value100"

Variant Binary Navigation

Simplified Example of Navigation



JSON STRING

{"field001":"value001",...,"field100":"value100"}

VARIANT METADATA

header	100 (dict size)	0 (offset 1)	...	792 (offset 100)	"field001"	...	"field100"

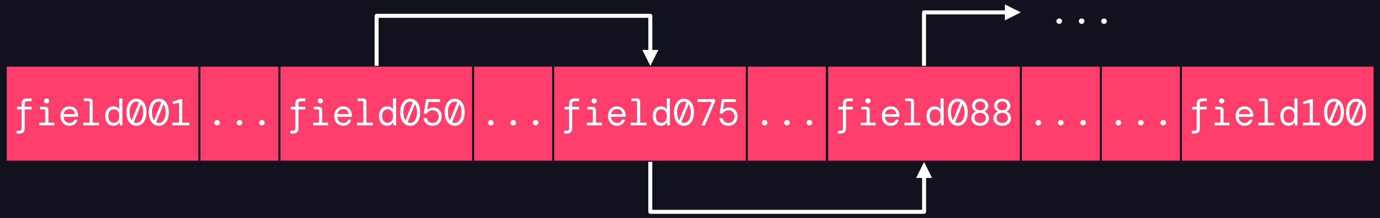
VARIANT VALUE

header	100 (obj size)	0	...	99	0	...	891	"value001"	...	"value100"

Binary search over the fields to find desired field

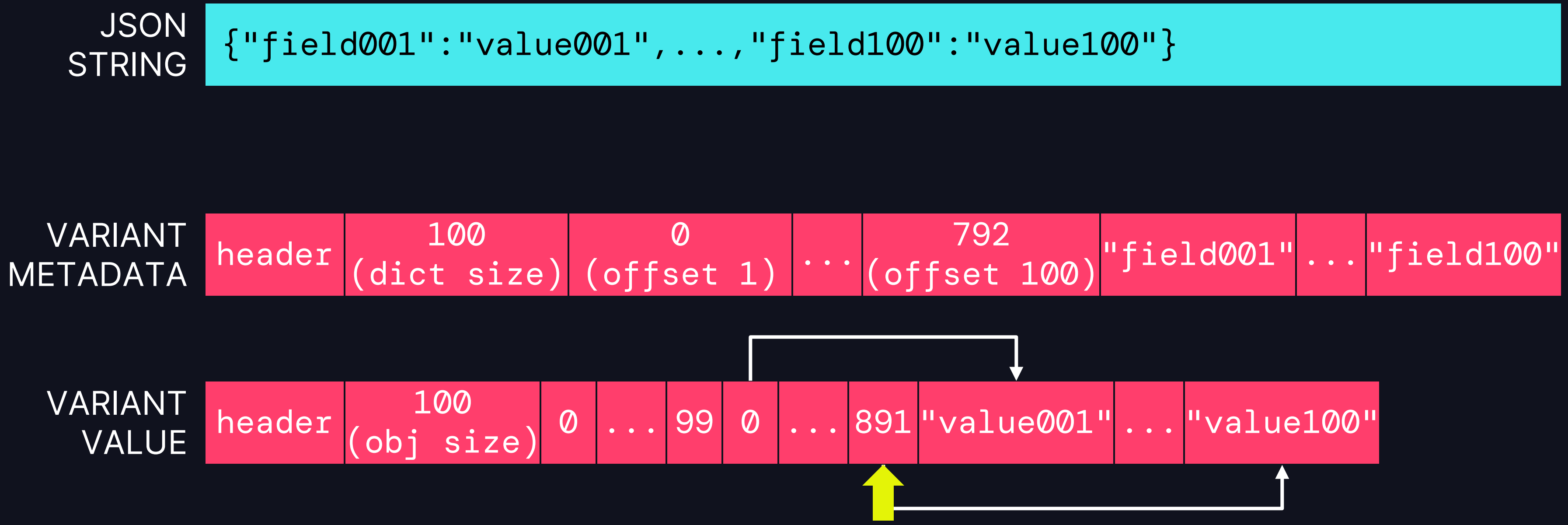
Variant Binary Navigation

Example of Binary Search



Variant Binary Navigation

Simplified Example of Navigation



JSON STRING

`{"field001":"value001",...,"field100":"value100"}`

VARIANT METADATA

header	100 (dict size)	0 (offset 1)	...	792 (offset 100)	"field001"	...	"field100"

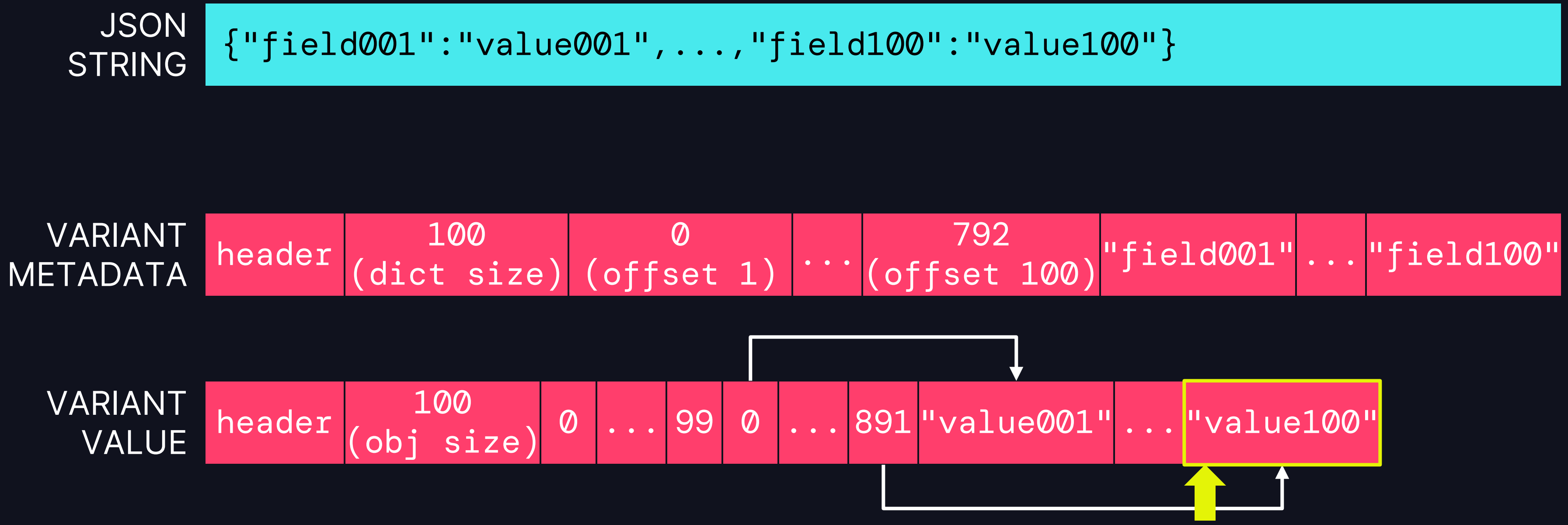
VARIANT VALUE

header	100 (obj size)	0	...	99	0	...	891	"value001"	...	"value100"

Jump to the desired field's offset value

Variant Binary Navigation

Simplified Example of Navigation



JSON STRING

{"field001":"value001",...,"field100":"value100"}

VARIANT METADATA

header	100 (dict size)	0 (offset 1)	...	792 (offset 100)	"field001"	...	"field100"

VARIANT VALUE

header	100 (obj size)	0	...	99	0	...	891	"value001"	...	"value100"

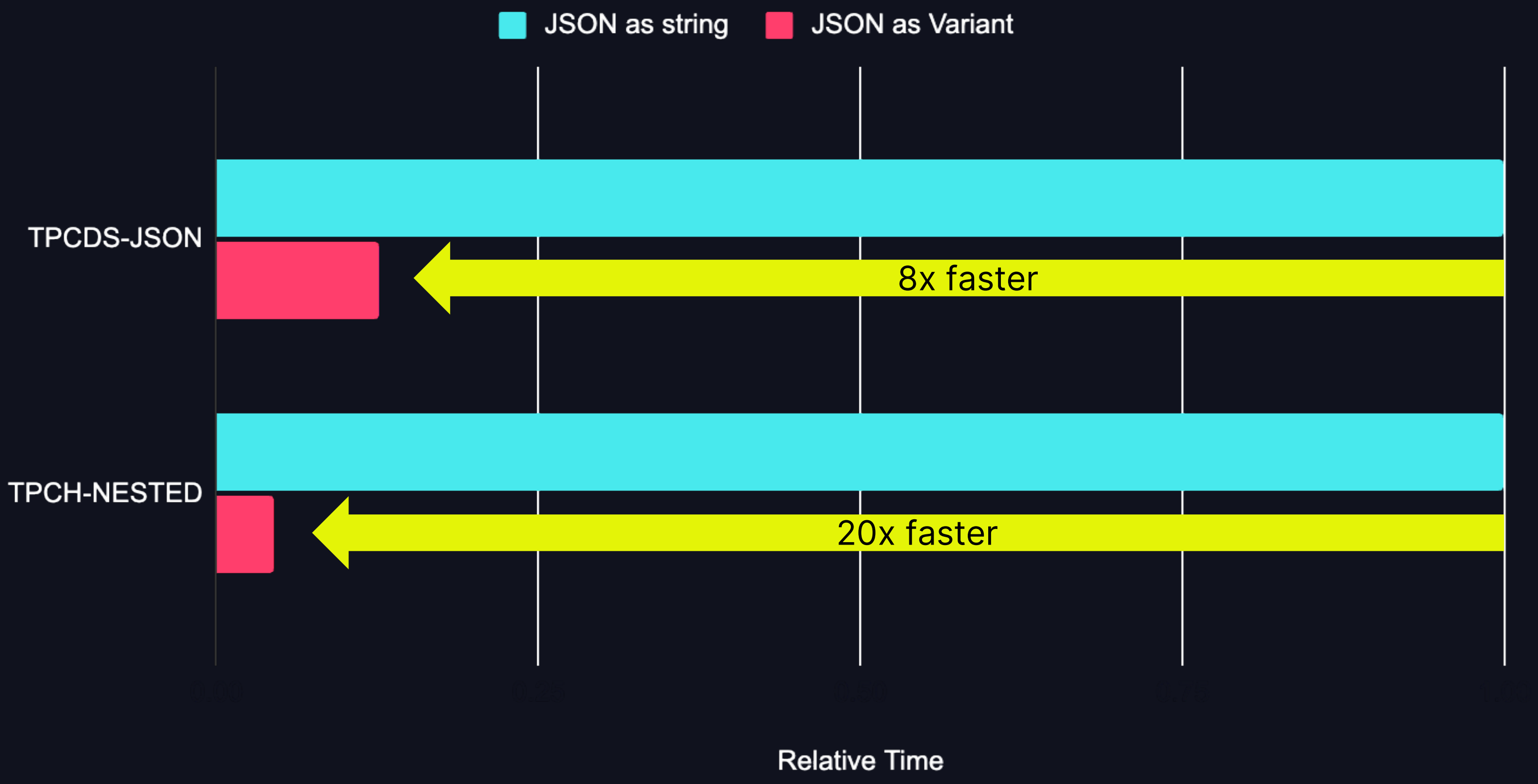
Jump to the desired field's value data

VARIANT PERFORMANCE

Performance Benchmarks

- TPCDS-JSON
  - Rows of each table is converted to flat JSON records or Variant records
- TPCH-NESTED
  - Dataset is denormalized to nested JSON records or nested Variant records

Variant vs JSON String Performance



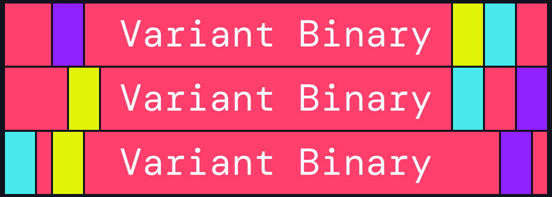
Sneak Peak: Variant Shredding

Work-in-Progress: Performance Optimizations

- Certain paths are stored in separate columns in file
- Shredded paths are removed from binary representation
- Faster to access shredded paths
    - Less IO required to fetch path
    - Less CPU required to decode values
    - min/max statistics available for data skipping
- Performance nearly equivalent to fully structured, relational data

Variant Shredding Storage

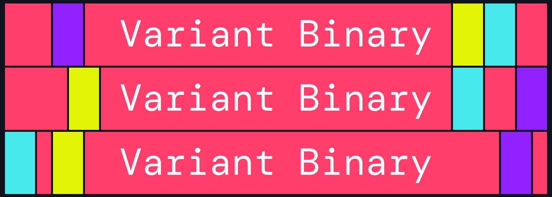
Without Shredding

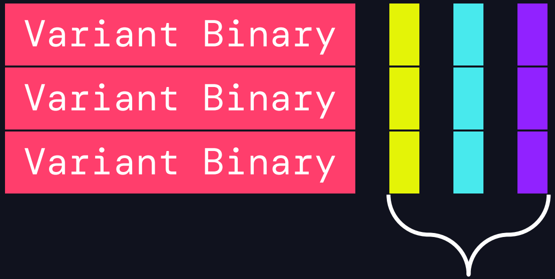


Variant Shredding Storage

Without Shredding

With Shredding



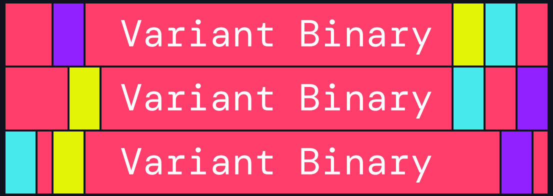


Shredded paths are removed from binary and stored in separate columns

Querying Variant Shredded Data

Query wants 🟨

Without Shredding



Variant Binary

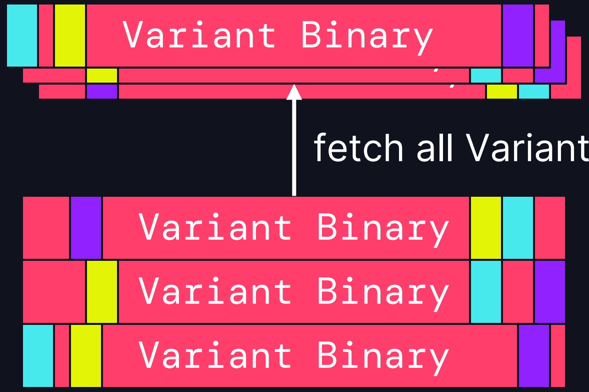
Variant Binary

Variant Binary

Querying Variant Shredded Data

Query wants 🟨

Without Shredding



Variant Binary

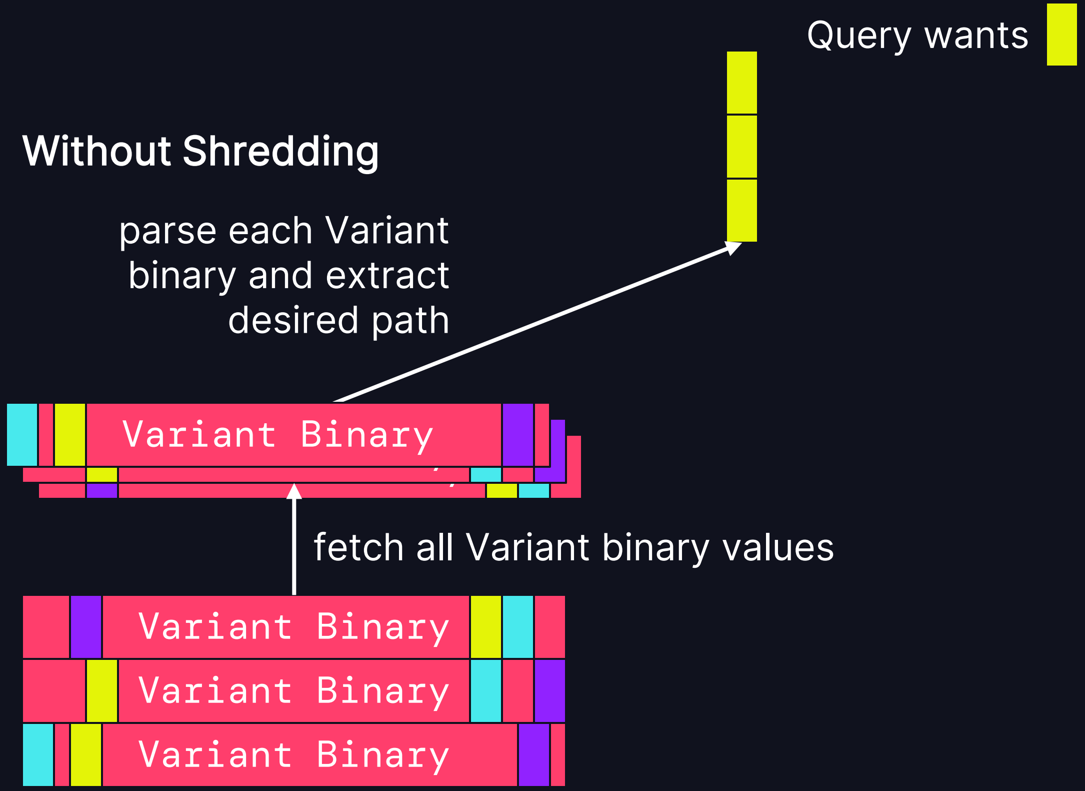
fetch all Variant binary values

Variant Binary

Variant Binary

Variant Binary

Querying Variant Shredded Data



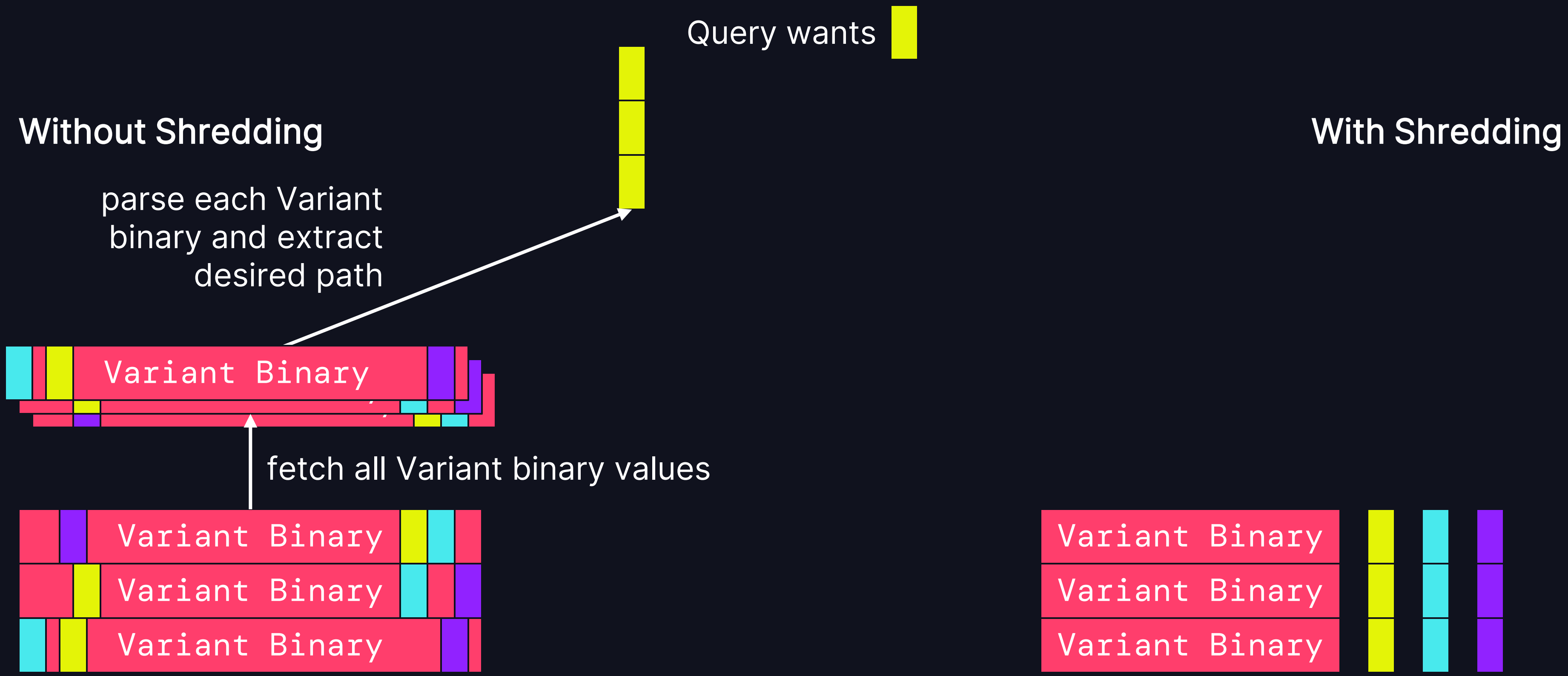
Query wants 🟨

Without Shredding

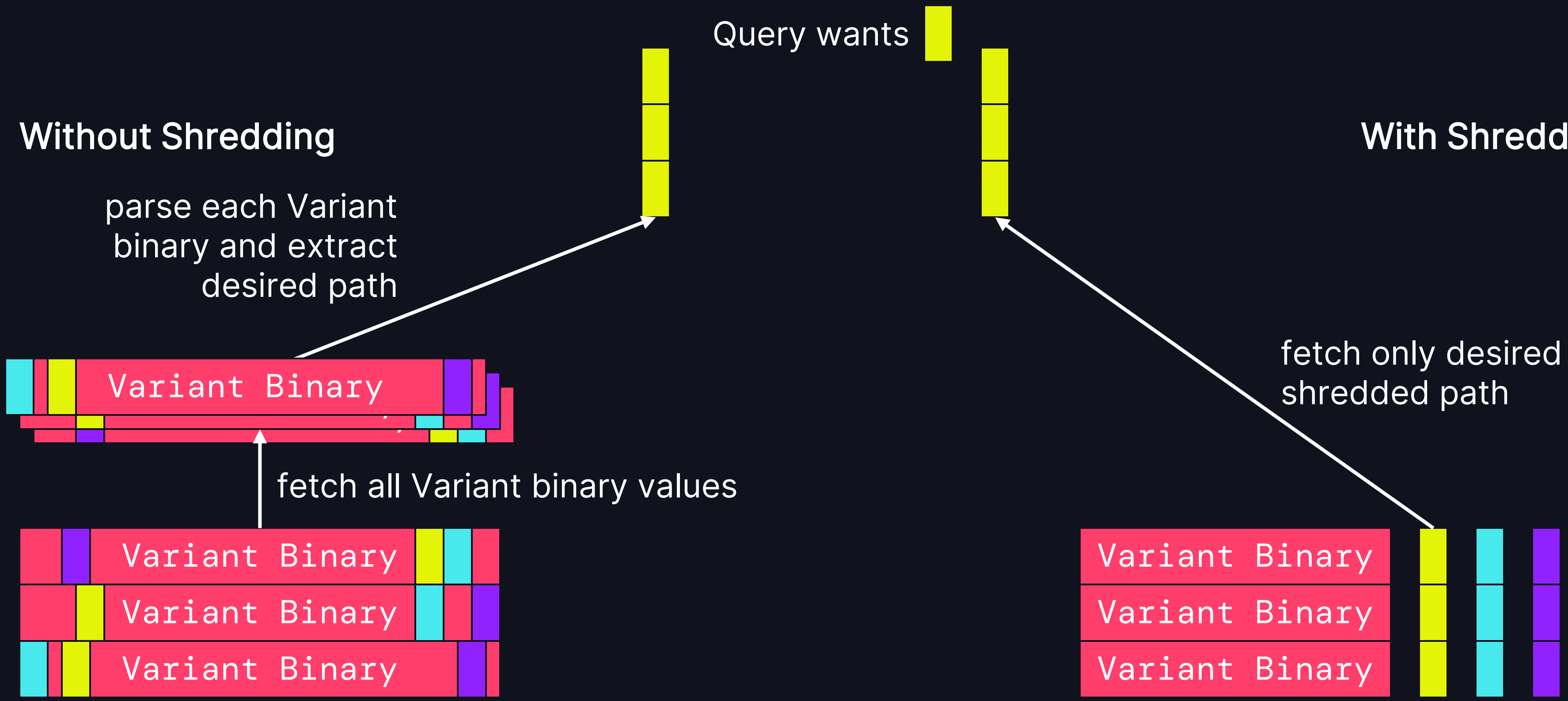
parse each Variant binary and extract desired path

fetch all Variant binary values

Querying Variant Shredded Data



Querying Variant Shredded Data



Query wants

Without Shredding

parse each Variant binary and extract desired path

fetch all Variant binary values

With Shredding

fetch only desired shredded path

Variant Data Type

Open

Performant

Flexible