

A close-up photograph of various mechanical components, including gears, springs, and metal brackets, arranged in a complex assembly. The lighting is dramatic, highlighting the metallic textures and shadows of the parts. The text is overlaid on the left side of the image.

# All About JSON and ClickHouse

## Tips, Tricks, and New Features

Robert Hodges and Diego Nieto  
26 July 2022

## Let's make some introductions

### Robert Hodges

Database geek with 30+ years  
on DBMS systems. Day job:  
Altinity CEO

### Diego Nieto

Database engineer focused on  
ClickHouse, PostgreSQL, and  
DBMS applications



ClickHouse support and services including [Altinity.Cloud](#)  
Authors of [Altinity Kubernetes Operator for ClickHouse](#)  
and other open source projects

# Reading and writing JSON - the basics

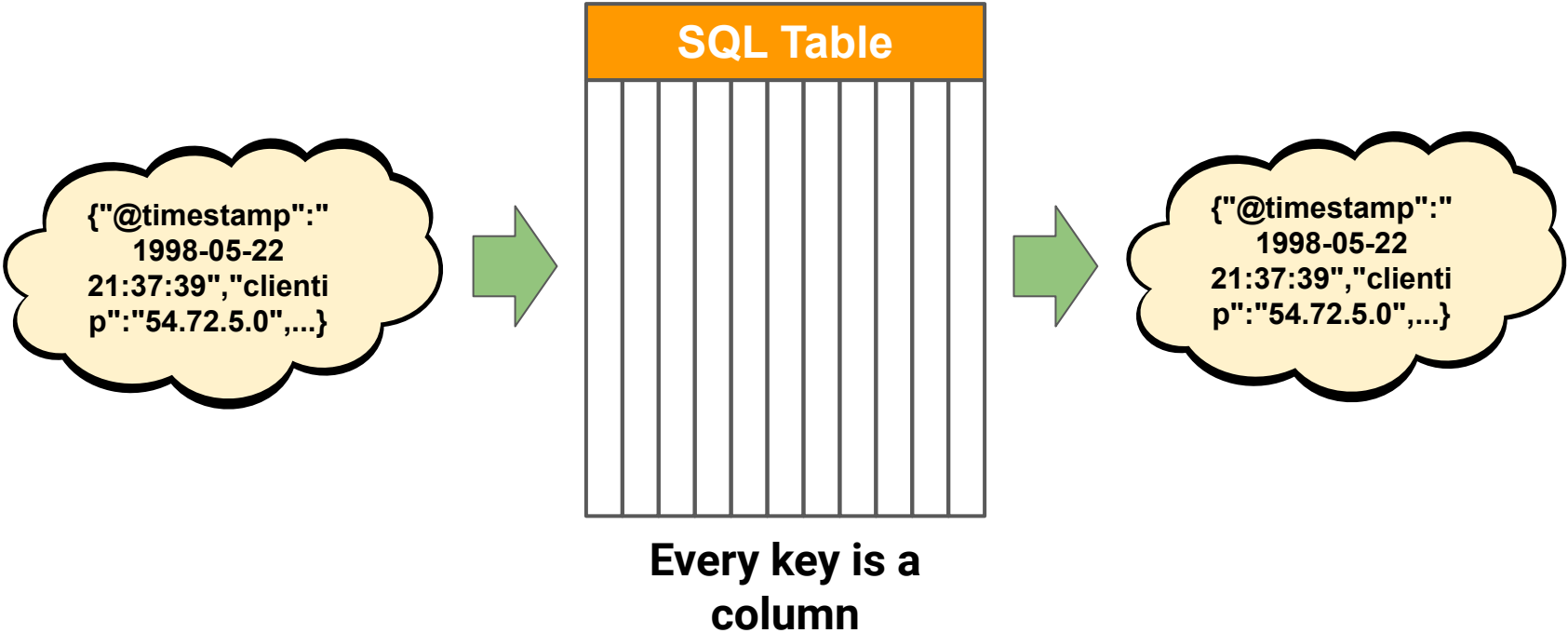
# JSON is pervasive as raw data

```
head http_logs.json
```

```
{"@timestamp": 895873059, "clientip": "54.72.5.0", "request":  
"GET /images/home_bg_stars.gif HTTP/1.1", "status": 200,  
"size": 2557}  
{"@timestamp": 895873059, "clientip": "53.72.5.0", "request":  
"GET /images/home_tool.gif HTTP/1.0", "status": 200, "size":  
327}  
...
```

**Web server log data**

# Reading and writing JSON data to/from tables



## Loading raw JSON using JSONEachRow input format

```
CREATE TABLE http_logs_tabular (  
    `@timestamp` DateTime,  
    `clientip` IPv4,  
    `status` UInt16,  
    `request` String,  
    `size` UInt32  
) ENGINE = MergeTree  
PARTITION BY toStartOfDay(`@timestamp`)  
ORDER BY `@timestamp`
```

```
clickhouse-client --query \  
    'INSERT INTO http_logs_tabular Format JSONEachRow' \  
    < http_logs_tabular
```

# Writing JSON using JSONEachRow output format

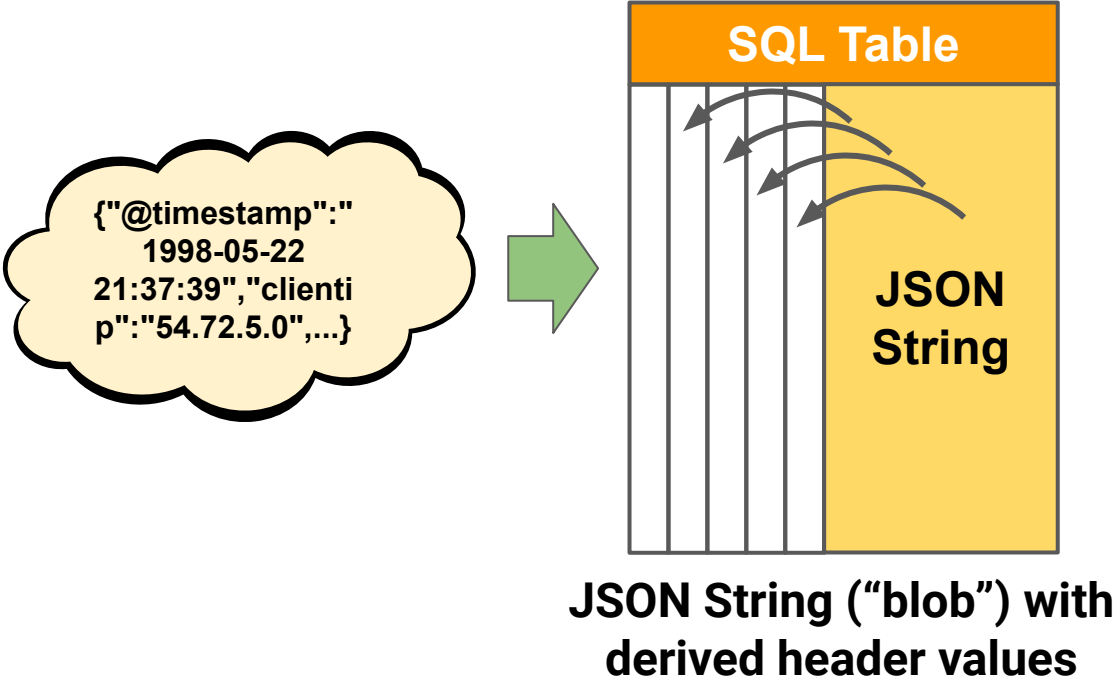
```
SELECT * FROM http_logs_tabular  
LIMIT 2  
FORMAT JSONEachRow
```

```
{"@timestamp":"1998-05-22  
21:37:39","clientip":"54.72.5.0","status":200,"request":"GET  
\images\home_bg_stars.gif HTTP\1.1","size":2557}  
{"@timestamp":"1998-05-22  
21:37:39","clientip":"53.72.5.0","status":200,"request":"GET  
\images\home_tool.gif HTTP\1.0","size":327}
```

# Storing JSON data in Strings



# Mapping JSON to a blob with optional derived columns



## Start by storing the JSON as a String

```
CREATE TABLE http_logs
(
  `file` String,
  `message` String
)
ENGINE = MergeTree
PARTITION BY file
ORDER BY tuple()
SETTINGS index_granularity = 8192
```



“Blob”

## Load data whatever way is easiest...

### **head http\_logs.csv**

```
"file","message"  
"documents-211998.json","{"@timestamp": 895873059,  
"clientip":"54.72.5.0", "request": "GET  
/images/home_bg_stars.gif HTTP/1.1", "status": 200, "size":  
2557}"  
"documents-211998.json","{"@timestamp": 895873059,  
"clientip":"53.72.5.0", "request": "GET /images/home_tool.gif  
HTTP/1.0", "status": 200, "size": 327}"  
...
```

```
clickhouse-client --query \  
  'INSERT INTO http_logs Format CSVWithNames' \  
  < http_logs.csv
```

## You can query using JSON\* functions

-- Get a JSON string value

```
SELECT JSONExtractString(message, 'request') AS request  
FROM http_logs LIMIT 3
```

-- Get a JSON numeric value

```
SELECT JSONExtractInt(message, 'status') AS status  
FROM http_logs LIMIT 3
```

-- Use values to answer useful questions.

```
SELECT JSONExtractInt(message, 'status') AS status, count() as count  
FROM http_logs WHERE status >= 400  
WHERE toDateTime(JSONExtractUInt32(message, '@timestamp')) BETWEEN  
      '1998-05-20 00:00:00' AND '1998-05-20 23:59:59'  
GROUP BY status ORDER BY status
```

## JSON\* vs visitParam functions

```
-- Get using JSON function  
SELECT JSONExtractString(message, 'request')  
FROM http_logs LIMIT 3
```

**SLOWER**  
**Complete  
JSON parser**

```
-- Get it with proper type.  
SELECT visitParamExtractString(message, 'request')  
FROM http_logs LIMIT 3
```

**FASTER**  
**But cannot distinguish same  
name in different structures**

## We can improve usability by ordering data

```
CREATE TABLE http_logs_sorted (  
  `file` String,  
  `message` String,  
  timestamp DateTime DEFAULT  
    toDateTime(JSONExtractUInt(message, '@timestamp'))  
)  
ENGINE = MergeTree  
PARTITION BY toStartOfMonth(timestamp)  
ORDER BY timestamp  
  
INSERT INTO http_logs_sorted  
  SELECT file, message FROM http_logs
```

## And still further by adding more columns

```
ALTER TABLE http_logs_sorted
  ADD COLUMN `status` Int16 DEFAULT JSONExtractInt(message,
'status') CODEC(ZSTD(1))
```

```
ALTER TABLE http_logs_sorted
  ADD COLUMN `request` String DEFAULT
JSONExtractString(message, 'request')
```

```
-- Force columns to be materialized
ALTER TABLE http_logs_sorted
  UPDATE status=status, request=request
  WHERE 1
```

## Our query is now simpler...

```
SELECT
    status, count() as count
FROM http_logs_sorted WHERE status >= 400 AND
    timestamp BETWEEN
        '1998-05-20 00:00:00' AND '1998-05-20 23:59:59'
GROUP BY status ORDER BY status
```



## And MUCH faster!

```
SELECT
  status, count() as count
FROM http_logs_sorted WHERE status >= 400 AND
  timestamp BETWEEN
    '1998-05-20 00:00:00' AND '1998-05-20 23:59:59'
GROUP BY status ORDER BY status
```

100x less I/O to read

Can use primary  
key index to drop  
blocks

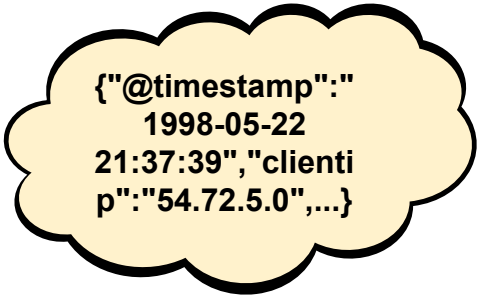
**0.014 seconds vs 9.8 seconds!**

# Using paired arrays and maps for JSON

# Representing JSON as paired arrays and maps



**Map: Header values with mapped key value pairs**



**Arrays: Header values with key-value pairs**

## Storing JSON in paired arrays

```
CREATE TABLE http_logs_arrays (  
  `file` String,  
  `keys` Array(String),  
  `values` Array(String),  
  timestamp DateTime CODEC(Delta, ZSTD(1))  
)  
ENGINE = MergeTree  
PARTITION BY toStartOfMonth(timestamp)  
ORDER BY timestamp
```

## Loading JSON to paired arrays

```
-- Load data. Might be better to format outside ClickHouse.  
INSERT into http_logs_arrays(file, keys, values, timestamp)  
  SELECT file,  
         arrayMap(x -> x.1,  
                 JSONExtractKeysAndValues(message, 'String')) keys,  
         arrayMap(x -> x.2,  
                 JSONExtractKeysAndValues(message, 'String')) values,  
         toDateTime(JSONExtractUInt(message, '@timestamp'))  
timestamp  
FROM http_logs limit 30000000
```

# Querying values in arrays

-- Run a query.

```
SELECT values[indexOf(keys, 'status')] status, count()  
FROM http_logs_arrays  
GROUP BY status ORDER BY status
```

status	count()
200	24917090
206	64935
302	1941
304	4899616
400	888
404	115005
500	525

**4-5x faster than accessing  
JSON string objects**

## Another way to store JSON objects: Maps

```
CREATE TABLE http_logs_map (  
  `file` String, `message` Map(String, String),  
  timestamp DateTime  
    DEFAULT toDateTime(toUInt32(message['@timestamp']))  
    CODEC(Delta, ZSTD(1))  
)  
ENGINE = MergeTree  
PARTITION BY toStartOfMonth(timestamp)  
ORDER BY timestamp
```

# Loading and querying JSON in Maps

```
-- Load data
INSERT into http_logs_map(file, message)
  SELECT file,
    JSONExtractKeysAndValues(message, 'String') message
  FROM http_logs

-- Run a query.
SELECT message['status'] status, count()
FROM http_logs_map
GROUP BY status ORDER BY status
```

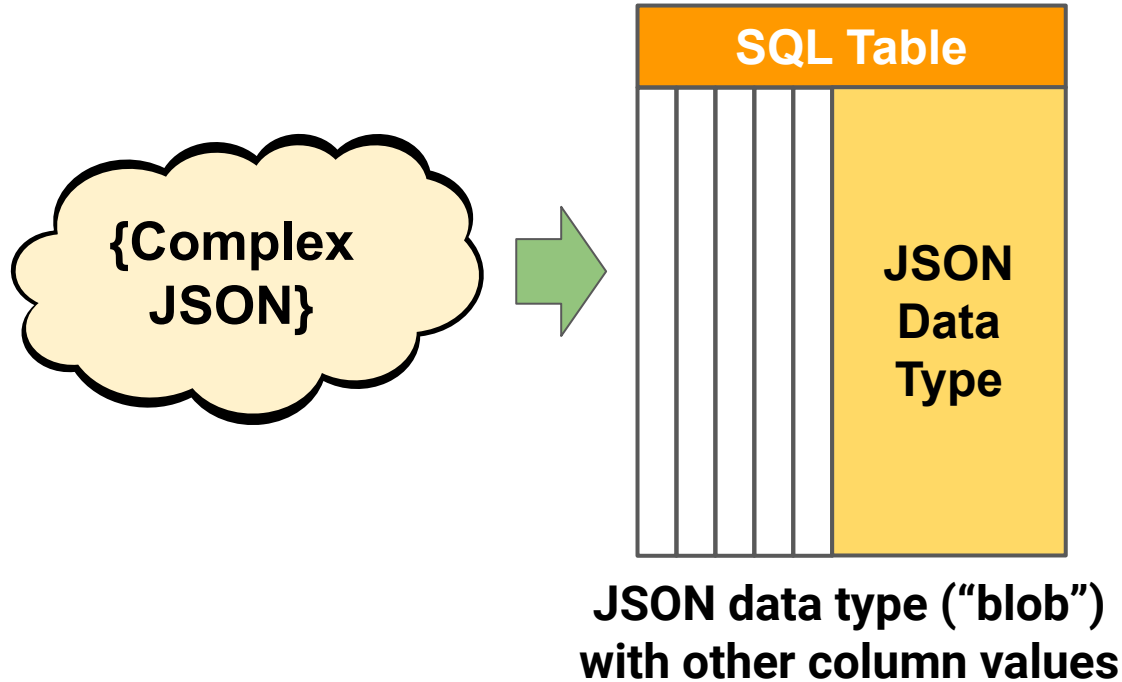
**4-5x faster than accessing  
JSON string objects**





# The JSON Data Type

# Mapping complex data to a JSON data type column



# How did JSON work until now?

- Storing JSON using *String* datatypes
- 2 Parsers:
  - Simple parser
  - Full-fledged parser
- 2-set functions for each parser:
  - Family of *simpleJSON* functions that only work for simple non-nested JSON files
    - *visitParamExtractUInt = simpleJSONExtractUInt*
  - Family of *JSONExtract\** functions that can parse any JSON object completely.
    - *JSONExtractUInt, JSONExtractString, JSONExtractRawArray ...*

Query Time!

# How did JSON work until now?

```
WITH JSONExtract(json, 'Tuple(a UInt32, b UInt32, c Nested(d UInt32, e String))') AS parsed_json
```

```
SELECT JSONExtractUInt(json, 'a') AS a, JSONExtractUInt(json, 'b') AS b,  
JSONExtractArrayRaw(json, 'c') AS array_c, tupleElement(parsed_json, 'a')  
AS a_tuple, tupleElement(parsed_json, 'b') AS b_tuple,  
tupleElement(parsed_json, 'c') AS array_c_tuple,  
tupleElement(tupleElement(parsed_json, 'c'), 'd') AS `c.d`,  
tupleElement(tupleElement(parsed_json, 'c'), 'e') AS `c.e`
```

```
FROM ( SELECT '{"a":1,"b":2,"c":[{"d":3,"e":"str_1"},  
{ "d":4,"e":"str_2"}, {"d":3,"e":"str_1"}, {"d":4,"e":"str_1"},  
{ "d":7,"e":"str_9"}]}' AS json )
```

FORMAT Vertical

## Let's dive in!

# How did JSON work until now?

## 1. Approach A: Using tuples

- 1.1. Get the structure of the json parsing it using the *JSONExtract* function and generate a tuple structure using a CTE (WITH clause)
- 1.2. Use *tupleElement* function to extract the tuples: `tupleElement->tupleElement` for getting nested fields

## 2. Approach B: Direct

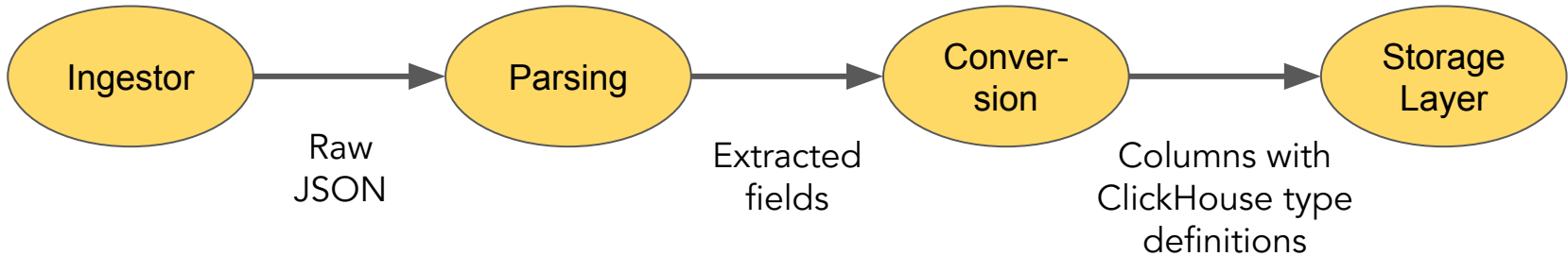
- 2.1. Use *JSONExtractUInt/Array* to extract the values directly

Both require multiple passes:

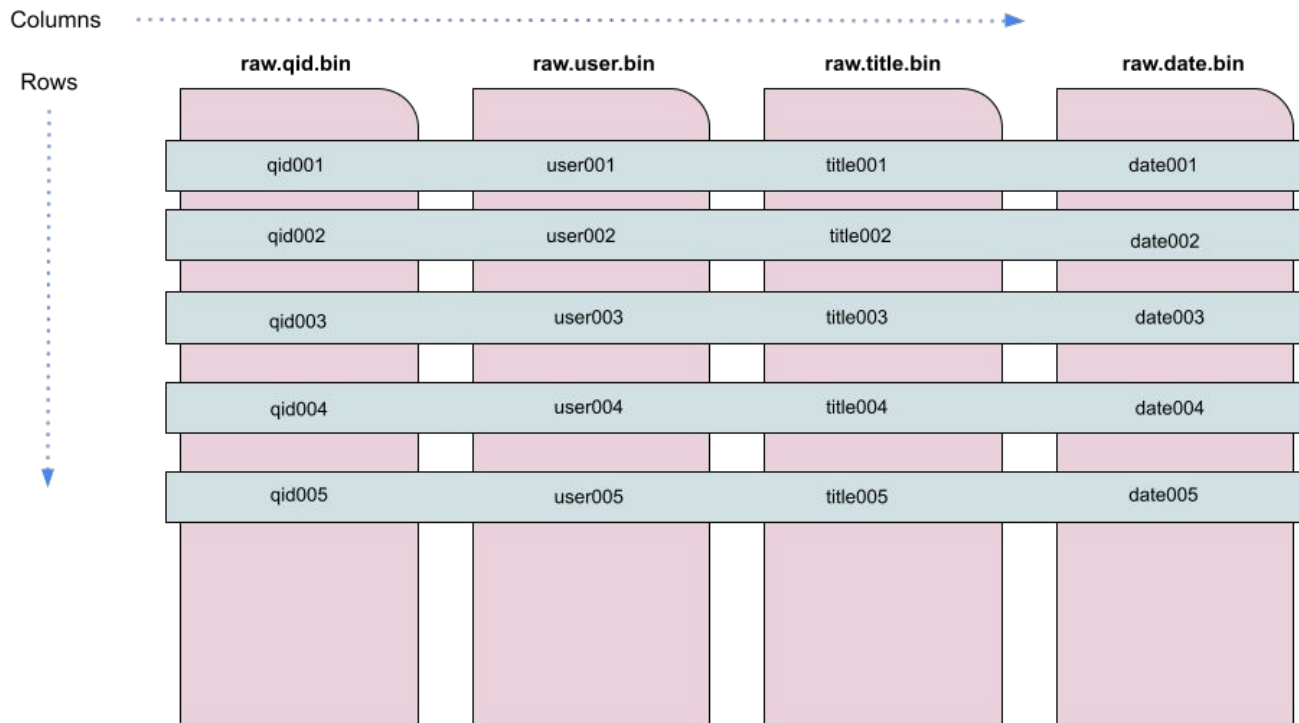
- Tuple approach= 2 pass (CTE + Query)
- Direct approach= 3 pass two ints (a and b) and an array (array\_c).

# New JSON

- ClickHouse parses JSON data at INSERT time.
- Automatic inference and creation of the underlying table structure
- JSON object stored in a columnar ClickHouse native format
- Named tuple and array notation to query JSON objects: *array[x] | tuple.element*



# New JSON storage format



# New JSON

```
SET allow_experimental_object_type = 1;

CREATE TABLE json_test.stack_overflow_js (`raw` JSON)

ENGINE = MergeTree ORDER BY tuple();

INSERT INTO stack_overflow_js
  SELECT json
     FROM file('stack_overflow_nested.json.gz', JSONAsObject);

SELECT count(*) FROM stack_overflow_js;
```

```
11203029 rows in set. Elapsed: 2.323 sec. Processed 11.20 million rows, 3.35 GB (4.82
million rows/s., 1.44 GB/s.)
```



# New JSON useful settings

```
SET describe_extend_object_types = 1;
```

```
DESCRIBE TABLE stack_overflow_js;
```

```
--Basic structure
```

```
SET describe_include_subcolumns = 1;
```

```
DESCRIBE TABLE stack_overflow_js FORMAT Vertical;
```

```
--Columns included
```

```
SET output_format_json_named_tuples_as_objects = 1;
```

```
SELECT raw FROM stack_overflow_js LIMIT 1 FORMAT JSONEachRow;
```

```
--JSON full structure
```

# New vs Old-school

stack\_overflow\_js vs stack\_overflow\_str:

```
CREATE TABLE nested_json.stack_overflow_js (`raw` JSON)
ENGINE = MergeTree ORDER BY tuple();
```

```
CREATE TABLE nested_json.stack_overflow_str (`raw` String)
ENGINE = MergeTree ORDER BY tuple();
```

- *topK stack\_overflow\_str:*

```
SELECT topK(100) (arrayJoin(JSONExtract(raw, 'tag', 'Array(String)')))
FROM stack_overflow_str;
```

1 rows in set. Elapsed: 2.101 sec. Processed 11.20 million rows, 3.73 GB (5.33 million rows/s., 1.77 GB/s.)

- *topK stack\_overflow\_js:*

```
SELECT topK(100) (arrayJoin(raw.tag)) FROM stack_overflow_js
```

1 rows in set. Elapsed: 0.331 sec. Processed 11.20 million rows, 642.07 MB (33.90 million rows/s., 1.94 GB/s.)

# Limitations:

- What happens if there are schema changes?:
  - column type changes, new keys, deleted keys ....
  - Insert a new json like this `{ "foo": "10", "bar": 10 }`:
    - CH will create a new part for this json
    - CH will create a tuple structure: `raw.foo` and `raw.bar`
    - OPTIMIZE TABLE FINAL
      - New mixed tuple = stack\_overflow tuple + foobar tuple
- Problems:
  - No errors or warnings during insertions
  - Malformed JSON will pollute our data
  - We cannot select slices like `raw.answers.*`
  - CH creates a dynamic column per json key (our JSON has 1K keys so 1K columns)

## Check tuple structure:

```
INSERT INTO stack_overflow_js VALUES ('{ "bar": "hello", "foo": 1 }');
```

```
SELECT table,  
       column,  
       name AS part_name,  
       type,  
       subcolumns.names,  
       subcolumns.type  
FROM system.parts_columns  
WHERE table = 'stack_overflow_js'  
FORMAT Vertical
```

# Check tuple structure:

Row 1:

```
table:          stack_overflow_js
column:         raw
part_name:      all_12_22_5
type:           Tuple(answers Nested(date String, user String), creationDate String, qid String, tag
Array(String), title String, user String)
subcolumns.names:
['answers', 'answers.size0', 'answers.date', 'answers.user', 'creationDate', 'qid', 'tag', 'tag.size0', 'title', 'user']
subcolumns.types:
['Nested(date String, user
String)', 'UInt64', 'Array(String)', 'Array(String)', 'String', 'String', 'Array(String)', 'UInt64', 'String', 'String']
subcolumns.serializations:
['Default', 'Default', 'Default', 'Default', 'Default', 'Default', 'Default', 'Default', 'Default', 'Default', 'Default']
```

Row 2:

```
table:          stack_overflow_js
column:         raw
part_name:      all_23_23_0
type:           Tuple(Bar String, foo Int8)
subcolumns.names:
['foo', 'foo']
subcolumns.types:
['String', 'String']
subcolumns.serializations: ['Default', 'Default']
```

# Improvements:

- CODEC Changes: LZ4 vs ZSTD

table	column	LZ4	ZSTD	uncompressed
stack_overflow_str	raw	1.73 GiB	1.23 GiB	3.73 GiB
stack_overflow_json	raw	1.30 GiB	886.77 GiB	2.29 GiB

```
SELECT table, column,  
       formatReadableSize(sum(column_data_compressed_bytes)) AS compressed,  
       formatReadableSize(sum(column_data_uncompressed_bytes)) AS uncompressed  
FROM system.parts_columns  
WHERE table IN ('stack_overflow_js', 'stack_overflow_str') AND column IN ('raw')  
GROUP BY table, column
```

- ALTER TABLEs

```
ALTER TABLE stack_overflow_str MODIFY COLUMN raw CODEC(ZSTD(3));  
ALTER TABLE stack_overflow_js MODIFY COLUMN raw CODEC(ZSTD(3));
```

# Improvements

- Query times: LZ4 vs ZSTD
  - LZ4
    - 0.3s New vs 2.1s Old
  - ZSTD
    - 0.4s New vs 2.8s Old

table	column	LZ4	ZSTD	comp.ratio
stack_overflow_str	raw	0.3s	0.4s	12%
stack_overflow_json	raw	2.1s	2.8s	10%

# Wrap-up and References



## Secrets to JSON happiness in ClickHouse

- Use JSON formats to read and write JSON data
- Fetch JSON String data with JSONExtract\*/JSONVisitParam\* functions
- Store JSON in paired arrays or maps
- **(NEW)** The new JSON data type stores data efficiently and offers convenient query syntax
  - It's still **experimental**

## More things to look at by yourself

- Using materialized views to populate JSON data
- Indexing JSON data
  - Indexes on JSON data type columns
  - Bloom filters on blobs
- More compression and codec tricks

# Where to get more information

ClickHouse Docs: <https://clickhouse.com/docs/>

Altinity Knowledge Base: <https://kb.altinity.com/>

Altinity Blog: <https://altinity.com>

ClickHouse Source Code and Tests: <https://github.com/ClickHouse/ClickHouse>

- Especially [tests](#)



Thank you!

Questions?

<https://altinity.com>

Altinity.Cloud

Altinity Support

Altinity Stable  
Builds

We're hiring!