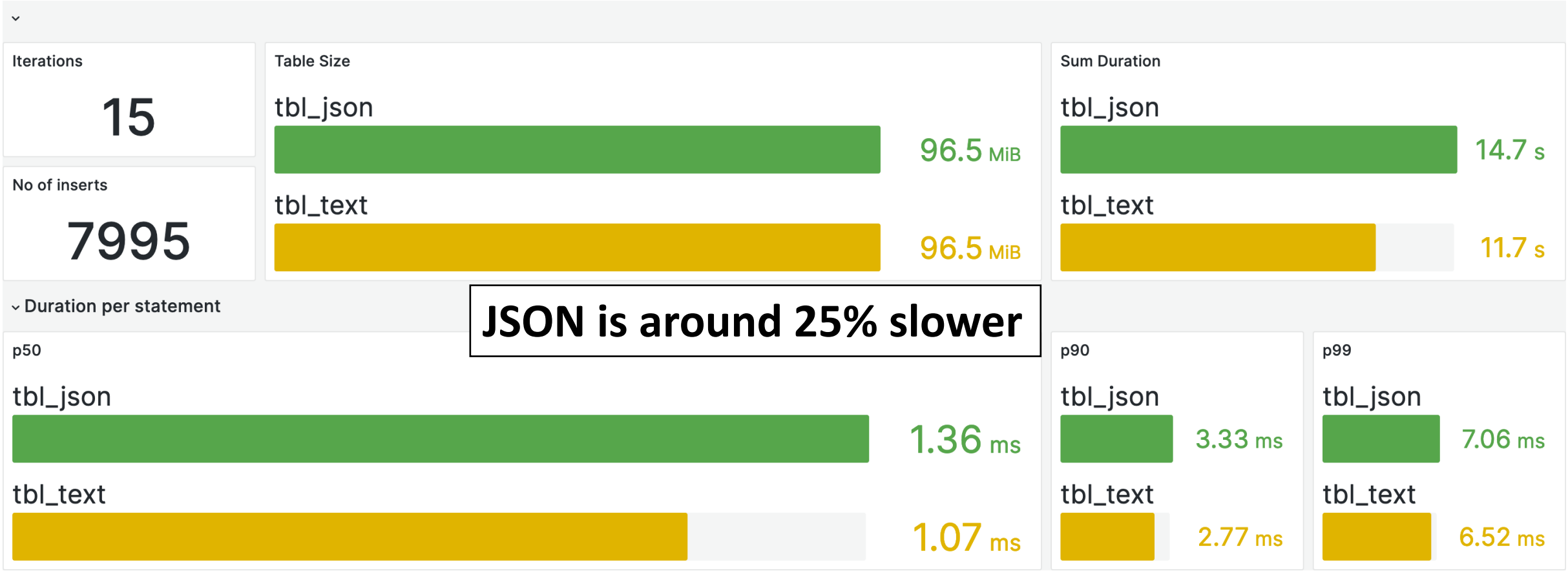
TEXT is faster than JSON

And JSON is faster than JSONB

* Based on some tests I did
https://github.com/waltton/pg_json_bench

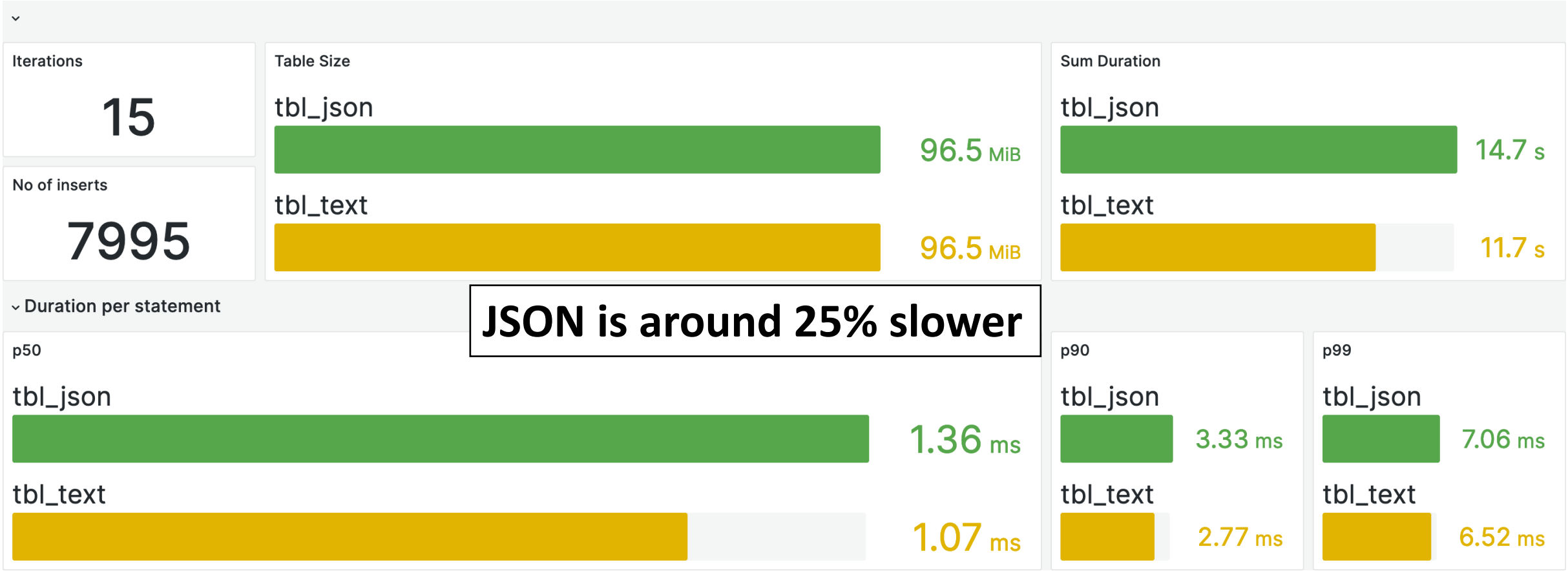
TEXT vs JSON

Insert - TEXT vs JSON

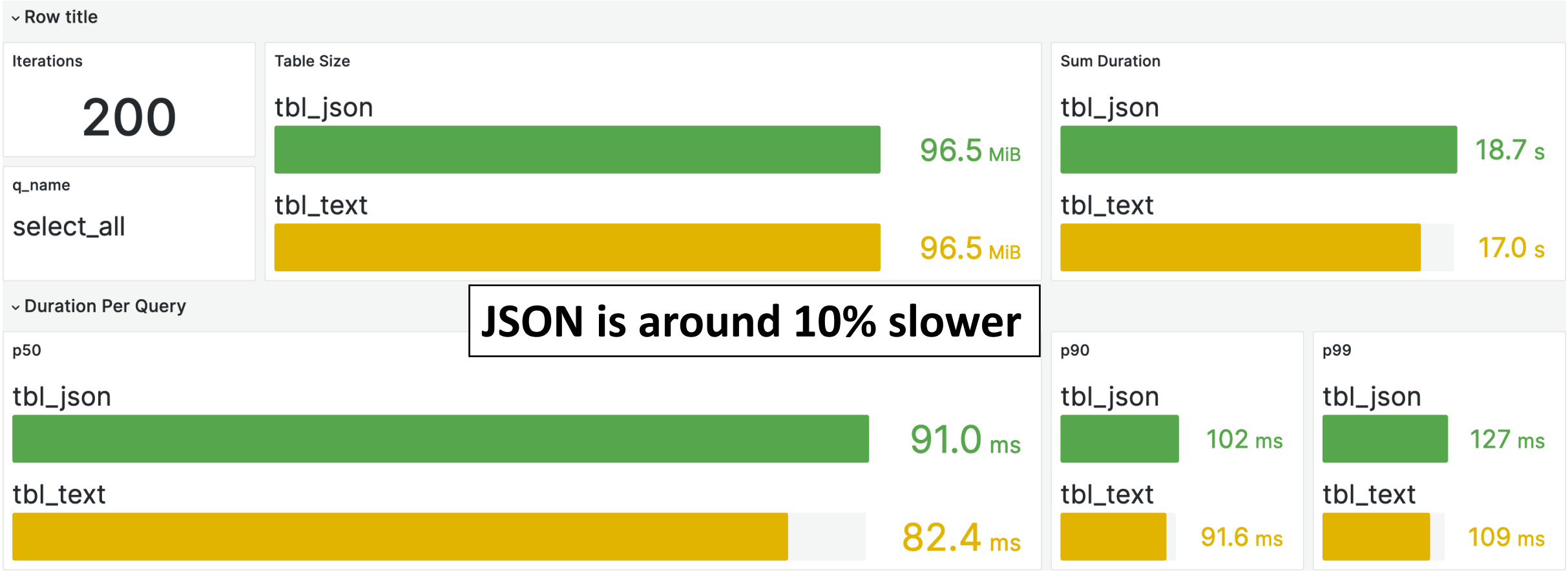


Insert - TEXT vs JSON

SLOWER

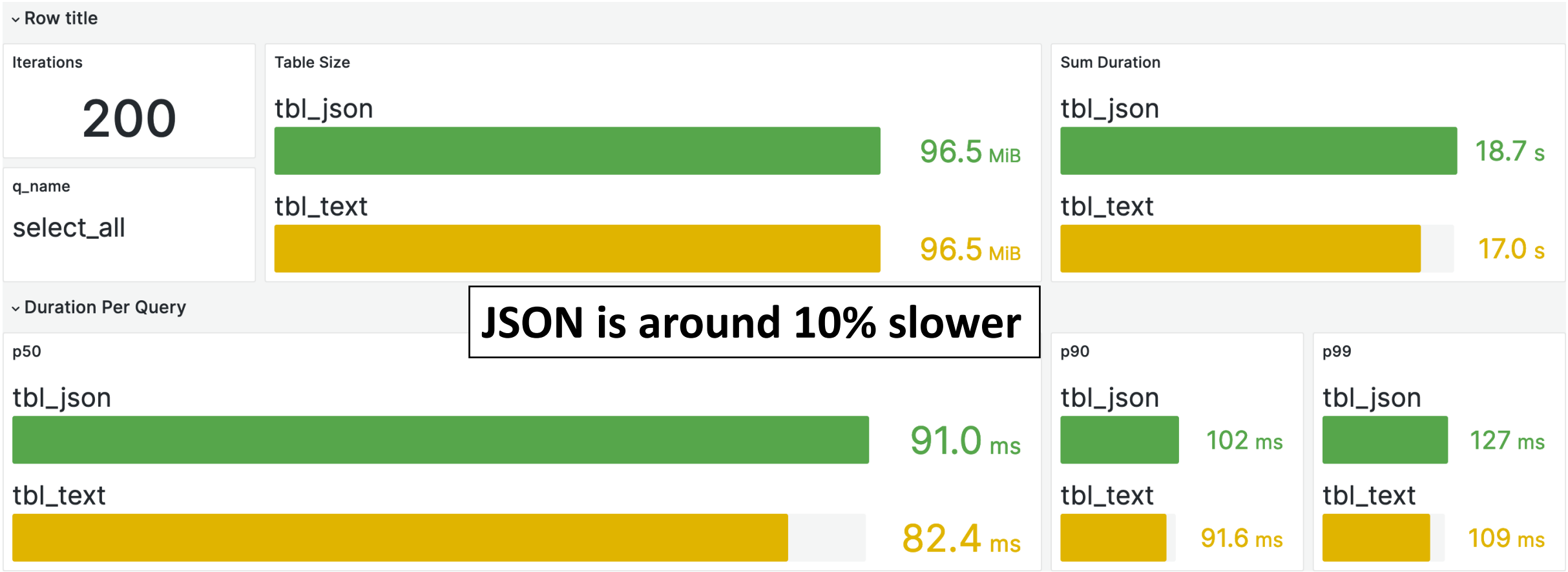


Select - TEXT vs JSON



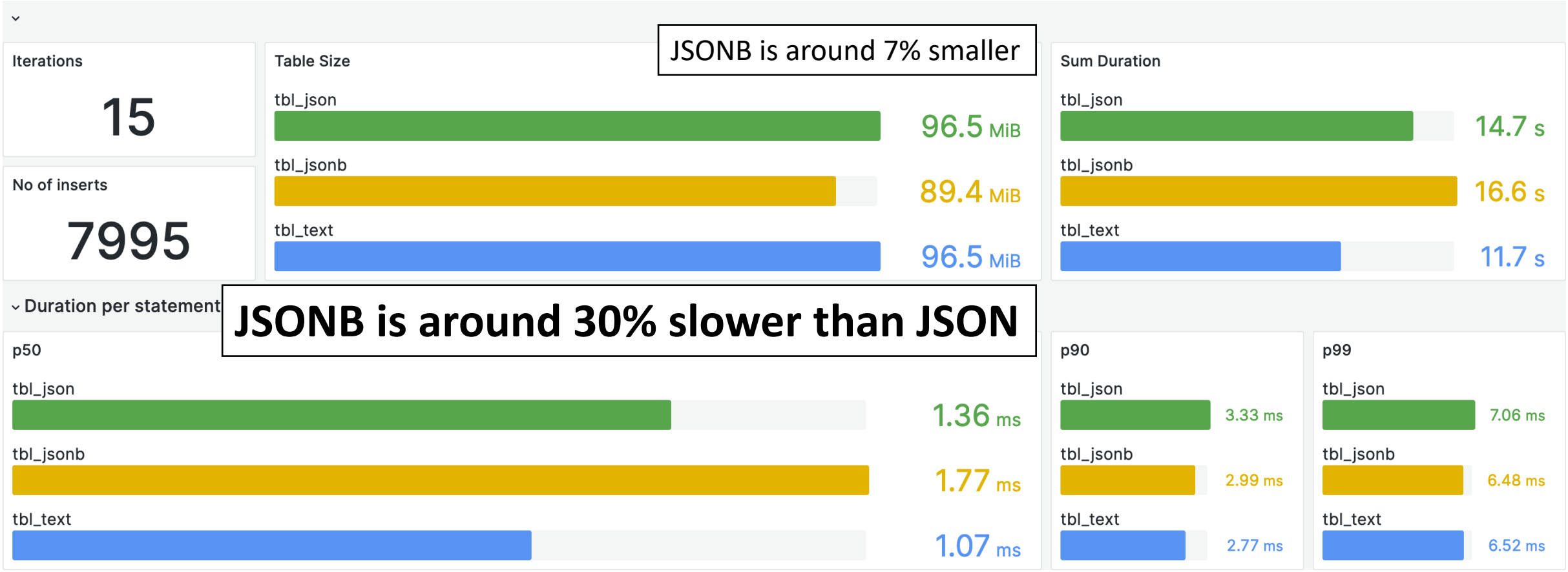
Select - TEXT vs JSON

SLOWER



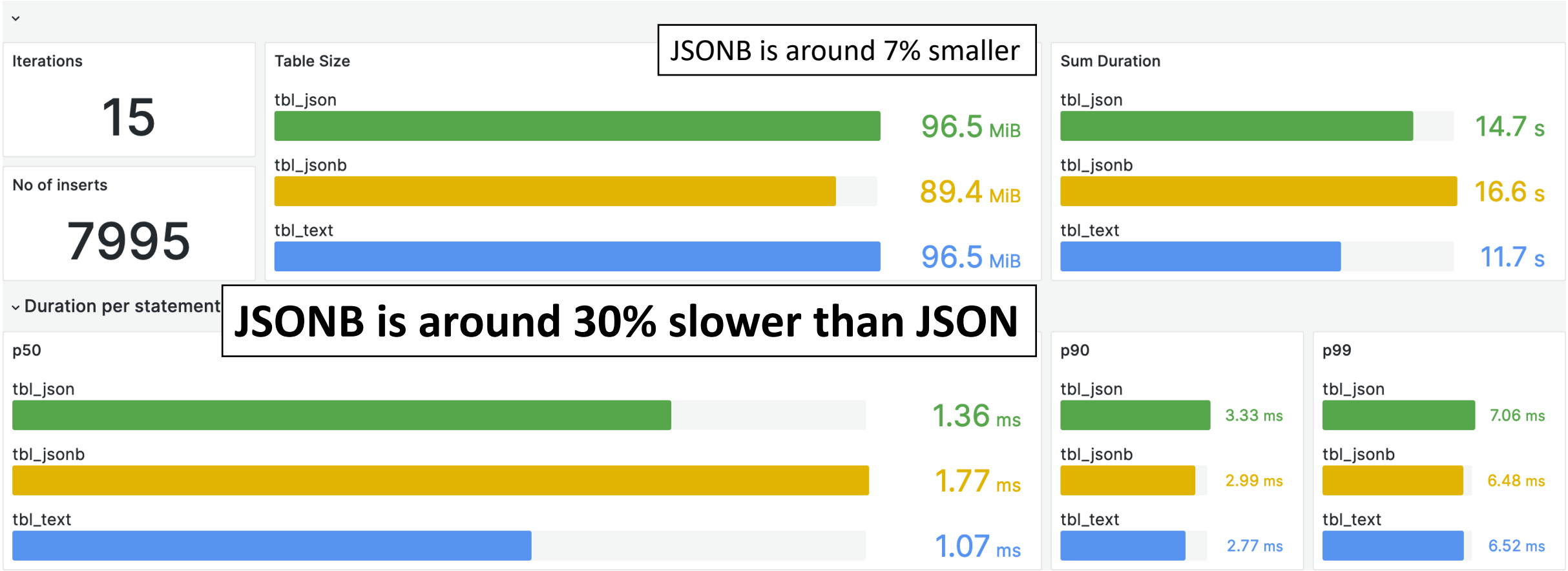
TEXT vs **JSON vs JSONB**

Insert - TEXT vs JSON vs JSONB

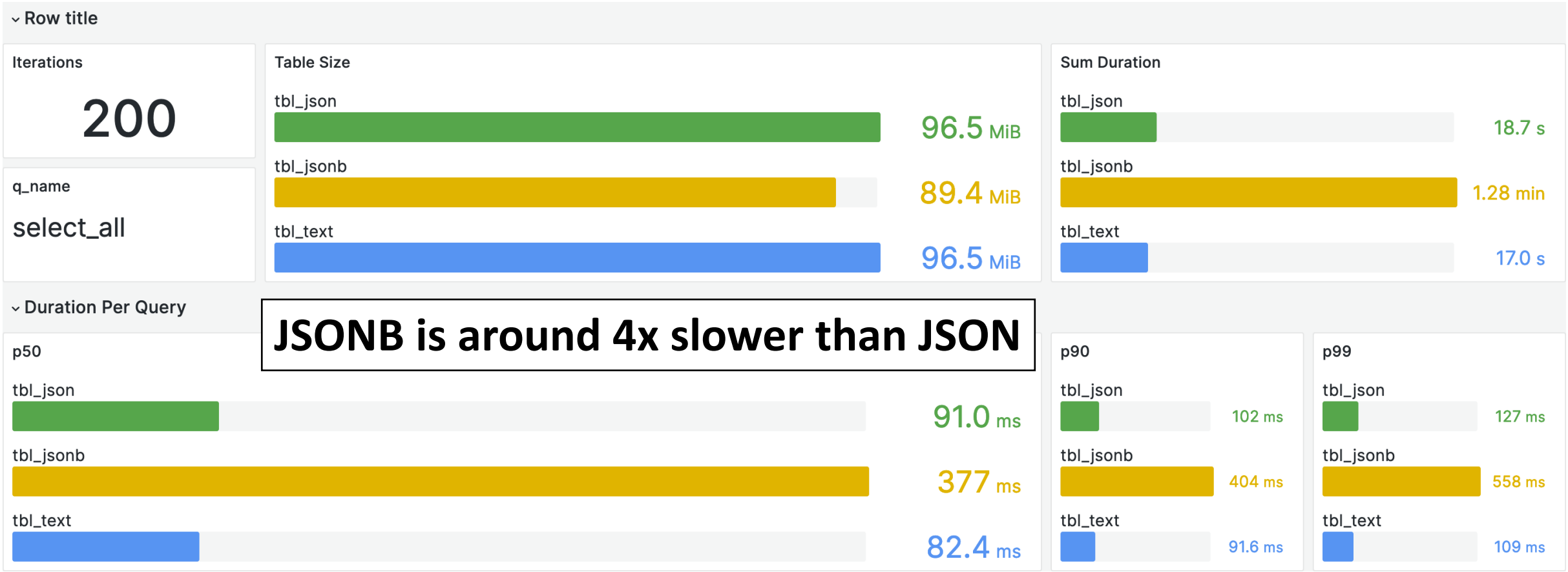


Insert - TEXT vs JSON vs JSONB

SLOWER

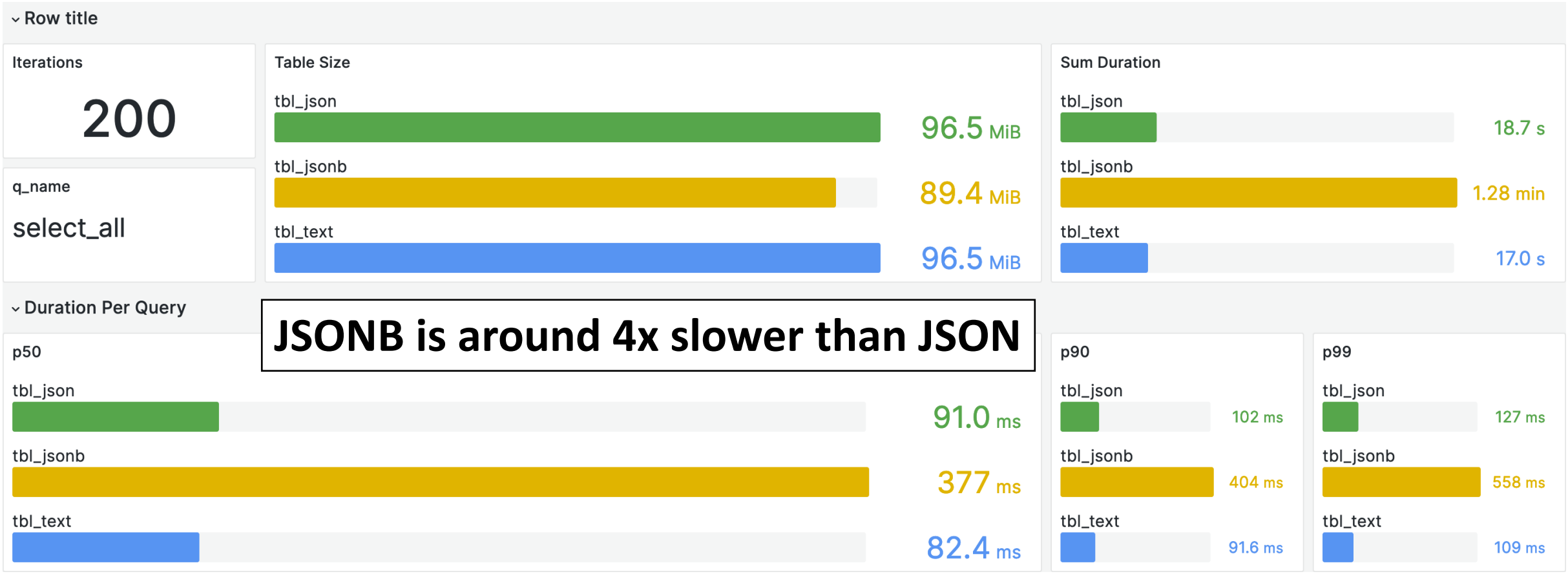


Select - TEXT vs JSON vs JSONB



Select - TEXT vs JSON vs JSONB

SLOWER



Why?

JSON

Is basically TEXT…

```
test=# CREATE TABLE what_is_inside_a_json_field(data json);
CREATE TABLE

test=# INSERT INTO what_is_inside_a_json_field(data) VALUES ('{"postgresql": "json"}'::JSON);
INSERT 0 1

test=# CREATE EXTENSION pageinspect;
CREATE EXTENSION

test=# SELECT tuple_data_split('what_is_inside_a_json_field'::regclass, t_data, t_infomask,
t_infomask2, t_bits) FROM heap_page_items(get_raw_page('what_is_inside_a_json_field', 0));
                   tuple_data_split
-------------------------------------------------------
 {"\\x2f7b22706f737467726573716c223a20226a736f6e227d"}
(1 row)

test=# \! echo '2f7b22706f737467726573716c223a20226a736f6e227d' | xxd -r -p && echo ""
/{"postgresql": "json"}
```

JSON
Is basically TEXT…

TEXT

```
test=# CREATE TABLE what_is_inside_a_json_field(data json);
CREATE TABLE

test=# INSERT INTO what_is_inside_a_json_field(data) VALUES ('{"postgresql": "json"}'::JSON);
INSERT 0 1

test=# CREATE EXTENSION pageinspect;
CREATE EXTENSION

test=# SELECT tuple_data_split('what_is_inside_a_json_field'::regclass, t_data, t_infomask,
t_infomask2, t_bits) FROM heap_page_items(get_raw_page('what_is_inside_a_json_field', 0));
                   tuple_data_split
-------------------------------------------------------
 {"\\x2f7b22706f737467726573716c223a20226a736f6e227d"}
(1 row)

test=# \! echo '2f7b22706f737467726573716c223a20226a736f6e227d' | xxd -r -p && echo ""
/{"postgresql": "json"}
```

JSON

With check for JSON rules, ...

```
test=# CREATE TABLE tbl_text(value TEXT);
CREATE TABLE

test=# INSERT INTO tbl_text VALUES ('{"broken", "json');
INSERT 0 1

-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

test=# CREATE TABLE tbl_json(value JSON);
CREATE TABLE

test=# INSERT INTO tbl_json VALUES ('{"broken", "json');
ERROR:  invalid input syntax for type json
LINE 1: INSERT INTO tbl_json VALUES ('{"broken", "json');
                                     ^
DETAIL:  Expected ":", but found ",".
CONTEXT:  JSON data, line 1: {"broken",...
```

JSON

With check for JSON rules, ...

INTEGRITY

```
test=# CREATE TABLE tbl_text(value TEXT);
CREATE TABLE

test=# INSERT INTO tbl_text VALUES ('{"broken", "json');
INSERT 0 1

-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

test=# CREATE TABLE tbl_json(value JSON);
CREATE TABLE

test=# INSERT INTO tbl_json VALUES ('{"broken", "json');
ERROR:  invalid input syntax for type json
LINE 1: INSERT INTO tbl_json VALUES ('{"broken", "json');
                                     ^
DETAIL:  Expected ":", but found ",".
CONTEXT:  JSON data, line 1: {"broken",...
```

JSON
And functions and operators

9.15. JSON Functions and Operators

Table 9-40 shows the operators that are available for use with JSON (see Section 8.14) data.

Table 9-40. JSON Operators

Operator	Right Operand Type	Description	Example
->	int	Get JSON array element	`'[1,2,3]'::json->2`
->	text	Get JSON object field	`'{"a":1,"b":2}'::json->'b'`
->>	int	Get JSON array element as text	`'[1,2,3]'::json->>2`
->>	text	Get JSON object field as text	`'{"a":1,"b":2}'::json->>'b'`
#>	array of text	Get JSON object at specified path	`'{"a":[1,2,3],"b":[4,5,6]}'::json#>'{a,2}'`
#>>	array of text	Get JSON object at specified path as text	`'{"a":[1,2,3],"b":[4,5,6]}'::json#>>'{a,2}'`

Table 9-41 shows the functions that are available for creating and manipulating JSON (see Section 8.14) data.

Table 9-41. JSON Support Functions

Function	Return Type	Description	Example	Example Result
`array_to_json(anyarray [, pretty_bool])`	**`json`**	Returns the array as JSON. A PostgreSQL multidimensional array becomes a JSON array of arrays. Line feeds will be added between dimension 1 elements if **`pretty_bool`** is true.	`array_to_json('{{1,5}, {99,100}}'::int[])`	`[[1,5],[99,100]]`
`row_to_json(record [, pretty_bool])`	**`json`**	Returns the row as JSON. Line feeds will be added between level 1 elements if **`pretty_bool`** is true.	`row_to_json(row(1,'foo'))`	`{"f1":1,"f2":"foo"}`
`to_json(anyelement)`	**`json`**	Returns the value as JSON. If the data type is not built in, and there is a cast from the type to **`json`**, the cast function will be used to perform the conversion. Otherwise, for any value other than a number, a Boolean, or a null value, the text representation will be used, escaped and quoted so that it is	`to_json('Fred said "Hi."'::text)`	`"Fred said \"Hi.\""`

*Docs for Postgres 9.3

JSON
And functions and operators

MORE USEFUL

9.15. JSON Functions and Operators

Table 9-40 shows the operators that are available for use with JSON (see Section 8.14) data.

Table 9-40. JSON Operators

Operator	Right Operand Type	Description	Example
->	int	Get JSON array element	`'[1,2,3]'::json->2`
->	text	Get JSON object field	`'{"a":1,"b":2}'::json->'b'`
->>	int	Get JSON array element as text	`'[1,2,3]'::json->>2`
->>	text	Get JSON object field as text	`'{"a":1,"b":2}'::json->>'b'`
#>	array of text	Get JSON object at specified path	`'{"a":[1,2,3],"b":[4,5,6]}'::json#>'{a,2}'`
#>>	array of text	Get JSON object at specified path as text	`'{"a":[1,2,3],"b":[4,5,6]}'::json#>>'{a,2}'`

Table 9-41 shows the functions that are available for creating and manipulating JSON (see Section 8.14) data.

Table 9-41. JSON Support Functions

Function	Return Type	Description	Example	Example Result
`array_to_json(anyarray [, pretty_bool])`	**`json`**	Returns the array as JSON. A PostgreSQL multidimensional array becomes a JSON array of arrays. Line feeds will be added between dimension 1 elements if **`pretty_bool`** is true.	`array_to_json('{{1,5}, {99,100}}'::int[])`	`[[1,5],[99,100]]`
`row_to_json(record [, pretty_bool])`	**`json`**	Returns the row as JSON. Line feeds will be added between level 1 elements if **`pretty_bool`** is true.	`row_to_json(row(1,'foo'))`	`{"f1":1,"f2":"foo"}`
`to_json(anyelement)`	**`json`**	Returns the value as JSON. If the data type is not built in, and there is a cast from the type to **`json`**, the cast function will be used to perform the conversion. Otherwise, for any value other than a number, a Boolean, or a null value, the text representation will be used, escaped and quoted so that it is	`to_json('Fred said "Hi."'::text)`	`"Fred said \"Hi.\""`

*Docs for Postgres 9.3

JSONB

B for binary, also…

```
test=# CREATE TABLE what_is_inside_a_jsonb_field(data jsonb);
CREATE TABLE

test=# INSERT INTO what_is_inside_a_jsonb_field(data) VALUES ('{"postgresql":
"json"}'::JSONB);
INSERT 0 1

test=# SELECT tuple_data_split('what_is_inside_a_jsonb_field'::regclass, t_data, t_infomask,
t_infomask2, t_bits) FROM heap_page_items(get_raw_page('what_is_inside_a_jsonb_field', 0));
                   tuple_data_split
-----------------------------------------------------
 {"\\x37010000200a000080040000000706f737467726573716c6a736f6e"}
(1 row)

test=# \! echo '37010000200a000080040000000706f737467726573716c6a736f6e' | xxd -r -p && echo ""
7
postgresqljson
test=#
```

JSONB

B for binary, also…

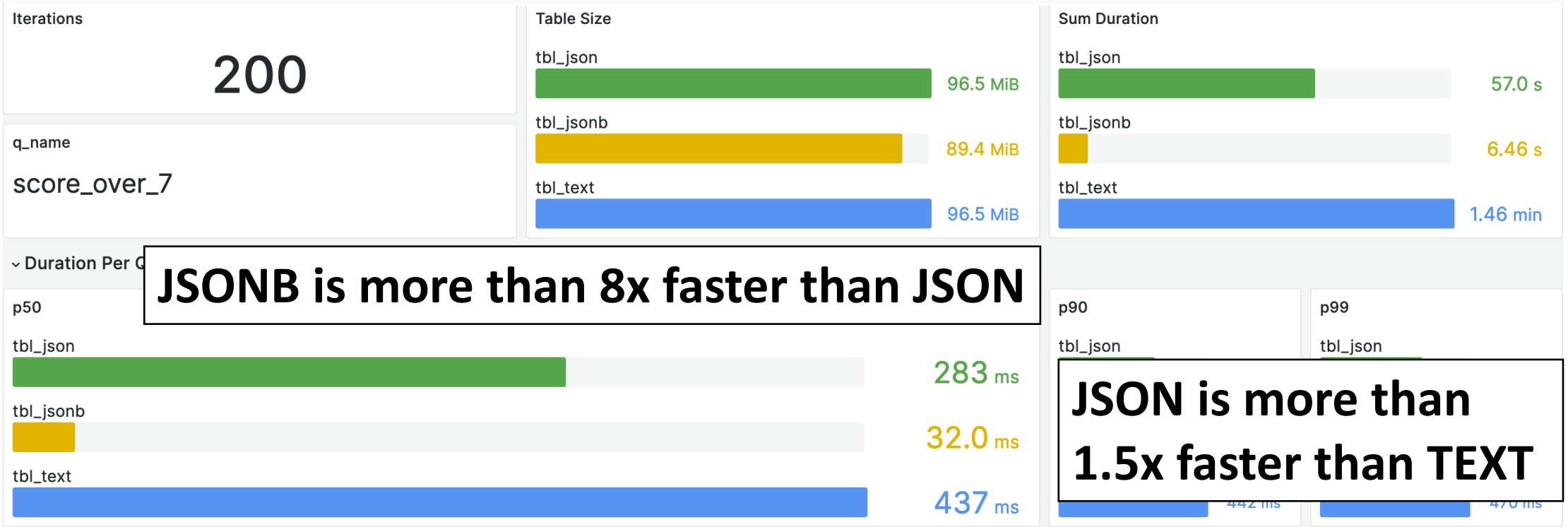
BINARY

```
test=# CREATE TABLE what_is_inside_a_jsonb_field(data jsonb);
CREATE TABLE

test=# INSERT INTO what_is_inside_a_jsonb_field(data) VALUES ('{"postgresql":
"json"}'::JSONB);
INSERT 0 1

test=# SELECT tuple_data_split('what_is_inside_a_jsonb_field'::regclass, t_data, t_infomask,
t_infomask2, t_bits) FROM heap_page_items(get_raw_page('what_is_inside_a_jsonb_field', 0));
                   tuple_data_split
-----------------------------------------------------
 {"\\x37010000200a000080040000000706f737467726573716c6a736f6e"}
(1 row)

test=# \! echo '37010000200a000080040000000706f737467726573716c6a736f6e' | xxd -r -p && echo ""
7
postgresqljson
test=#
```

JSONB

Faster processing

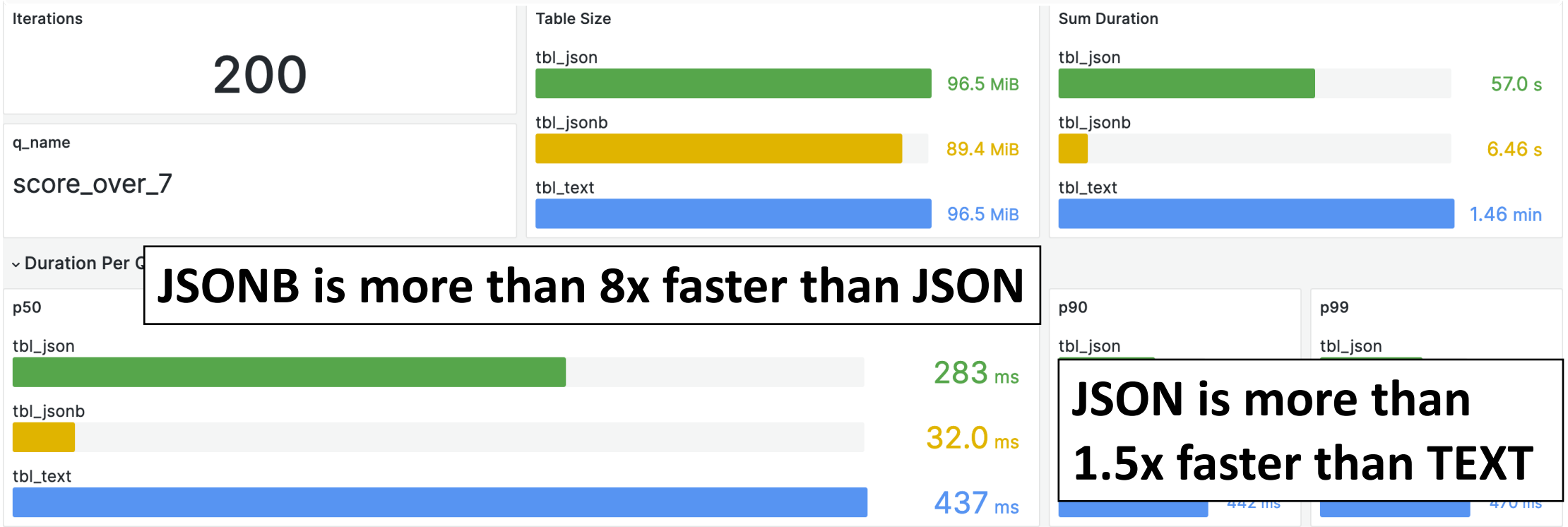
```sql
-- TEXT field
SELECT * FROM %s WHERE CAST(CAST(data AS JSON)->>'score' AS FLOAT) > 7.0

-- JSON and JSONB fields
SELECT * FROM %s WHERE CAST(data->>'score' AS FLOAT) > 7.0
```



JSONB
Faster processing

FASTER!!!

```
1 -- TEXT field
2 SELECT * FROM %s WHERE CAST(CAST(data AS JSON)->>'score' AS FLOAT) > 7.0
3
4 -- JSON and JSONB fields
5 SELECT * FROM %s WHERE CAST(data->>'score' AS FLOAT) > 7.0
```



JSONB
Indexing support

```
1 SELECT COUNT(*) FROM %s WHERE data @> '{"year": 2000}'
```

Iterations

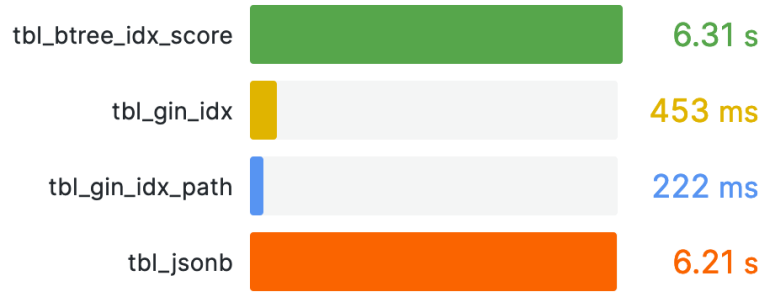
200

q_name

count_year_2000_at_gt

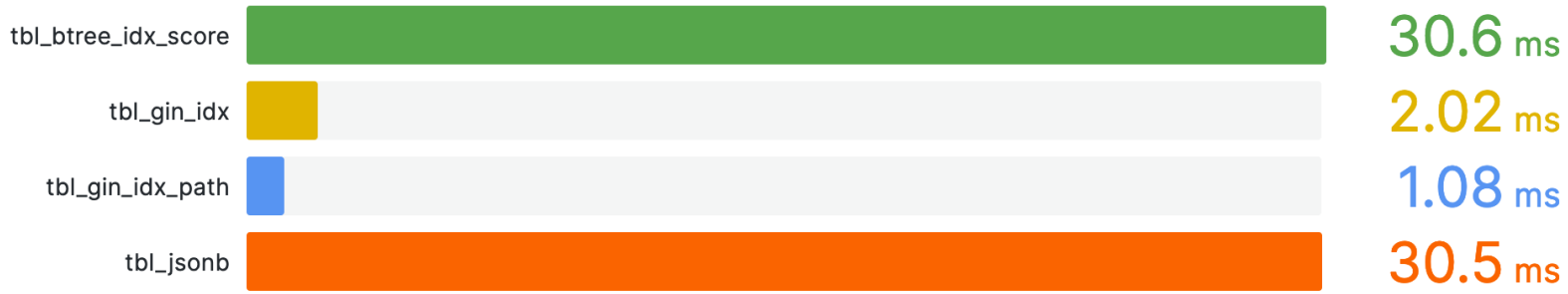
Table Size

Sum Duration



Duration Per Query

p50



Default GIN index is around 15x faster than no index on JSONB

jsonb_path_ops is almost 2x faster than the default

JSONB
Indexing support

FASTER!!!

```
1 SELECT COUNT(*) FROM %s WHERE data @> '{"year": 2000}'
```

Iterations

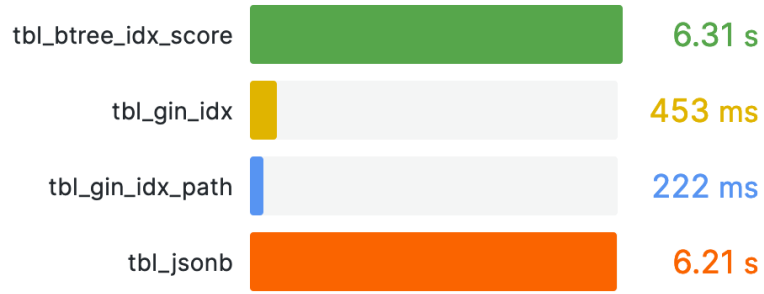
200

q_name

count_year_2000_at_gt

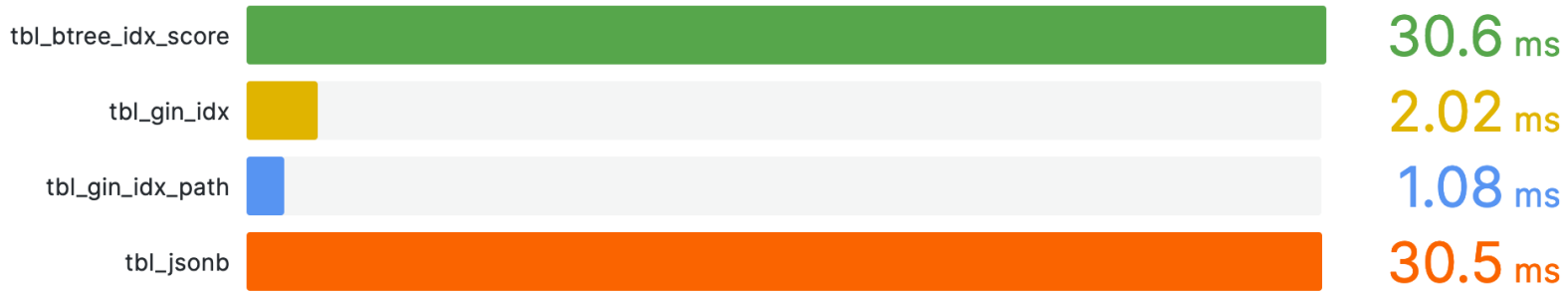
Table Size

Sum Duration



Duration Per Query

p50



Default GIN index is around 15x faster than no index on JSONB

jsonb_path_ops is almost 2x faster than the default

Thanks

waltton/pg_json_bench