Leveraging the Power of JSON/JSONB Fields

Key Considerations for Effective Utilization in PostgreSQL

Key Considerations

- Data type
- Indexing
- Data Size
- Enforcing Schema
- Storage

Some Key Considerations

- Data type
- Indexing
- Data Size
- ~~Enforcing Schema~~
- ~~Storage~~ (medium post)

Data Type - JSON

Introduced in PostgreSQL 9.2 (2012-09-10)

"JSON data types are for storing JSON (JavaScript Object Notation) data, as specified in RFC 7159. Such data can also be stored as text, but the JSON data types have the advantage of enforcing that each stored value is valid according to the JSON rules."

Into the JSON field

```
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

```

Into the JSON field

```
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
```

Into the JSON field

```
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)
```

Into the JSON field

```
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"}
```

TEXT vs JSON

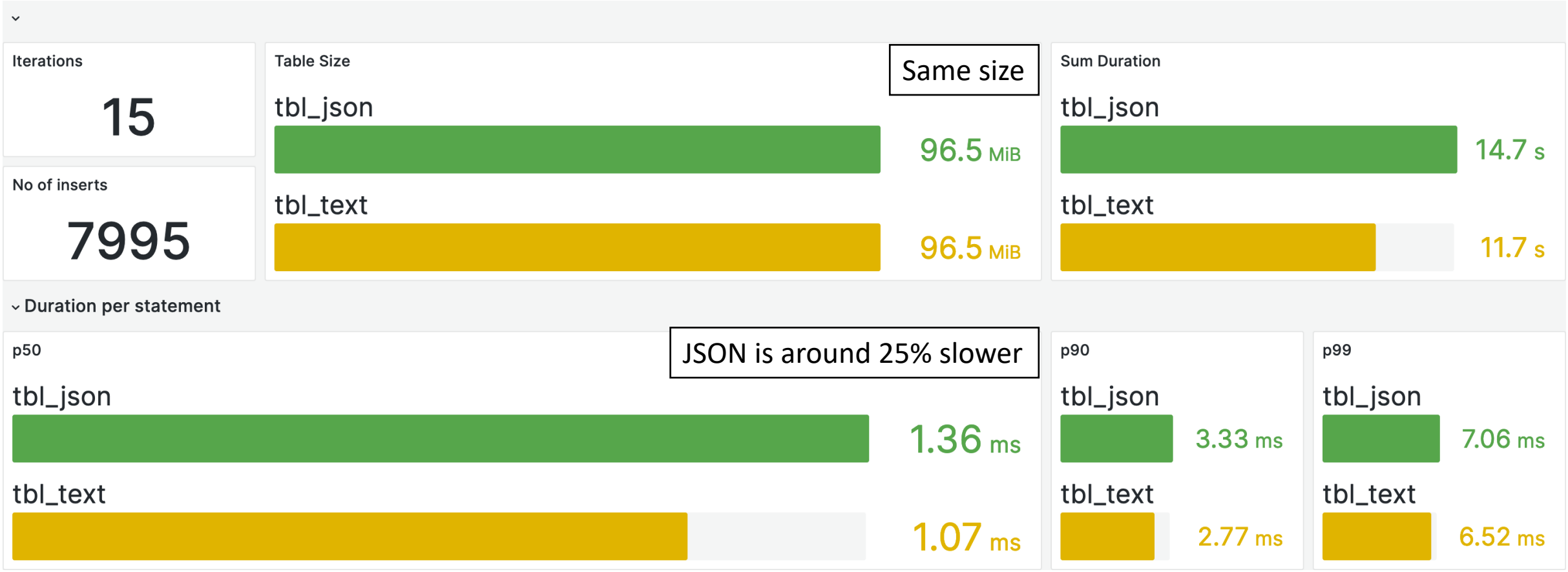
```
CREATE TABLE tbl_json (data json);
```

Insert

- 15 Iterations, intercalating between a table for each type
- Data set from https://github.com/algolia/datasets/movies/records.json
- Each row has one movie
- Total 133,250 records
- Batches of 250 records

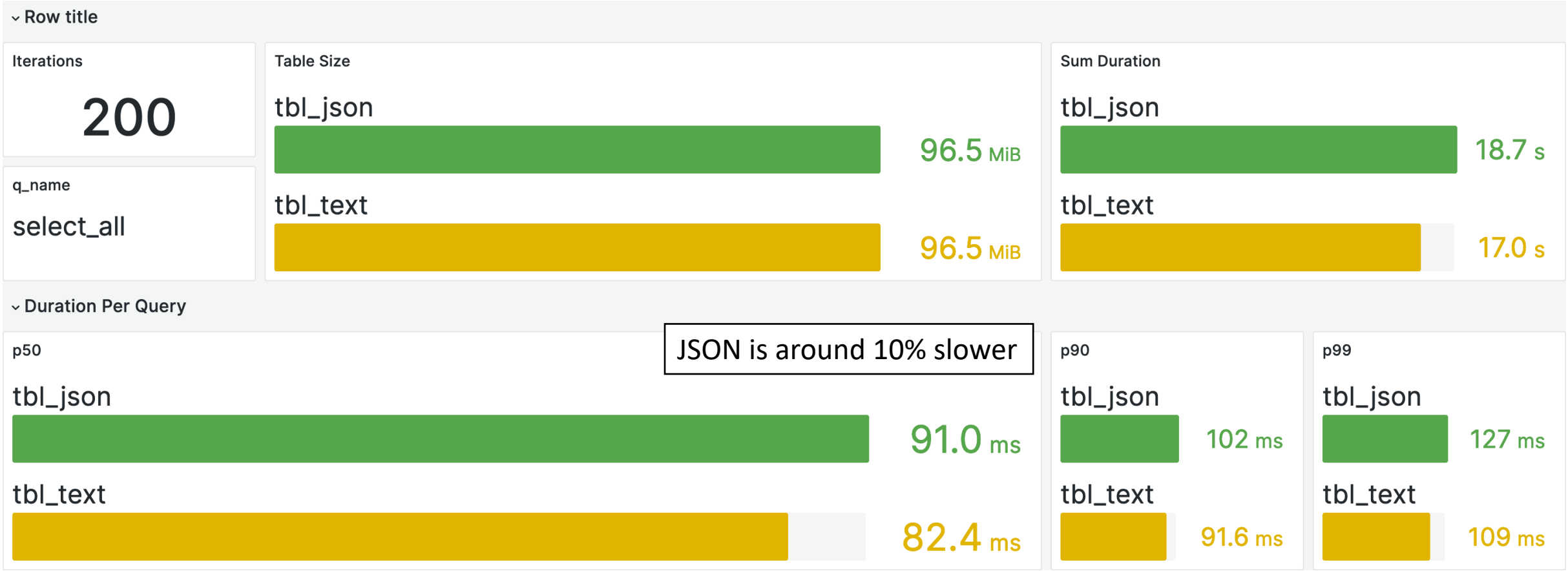
* All tests run from a MacBook Air Apple M2 24 GB

TEXT vs JSON - Insert



* The values and percentages from this presentation are based on the tests that I did, this may vary a lot depending on the dataset used

TEXT vs JSON – Select all



JSONB

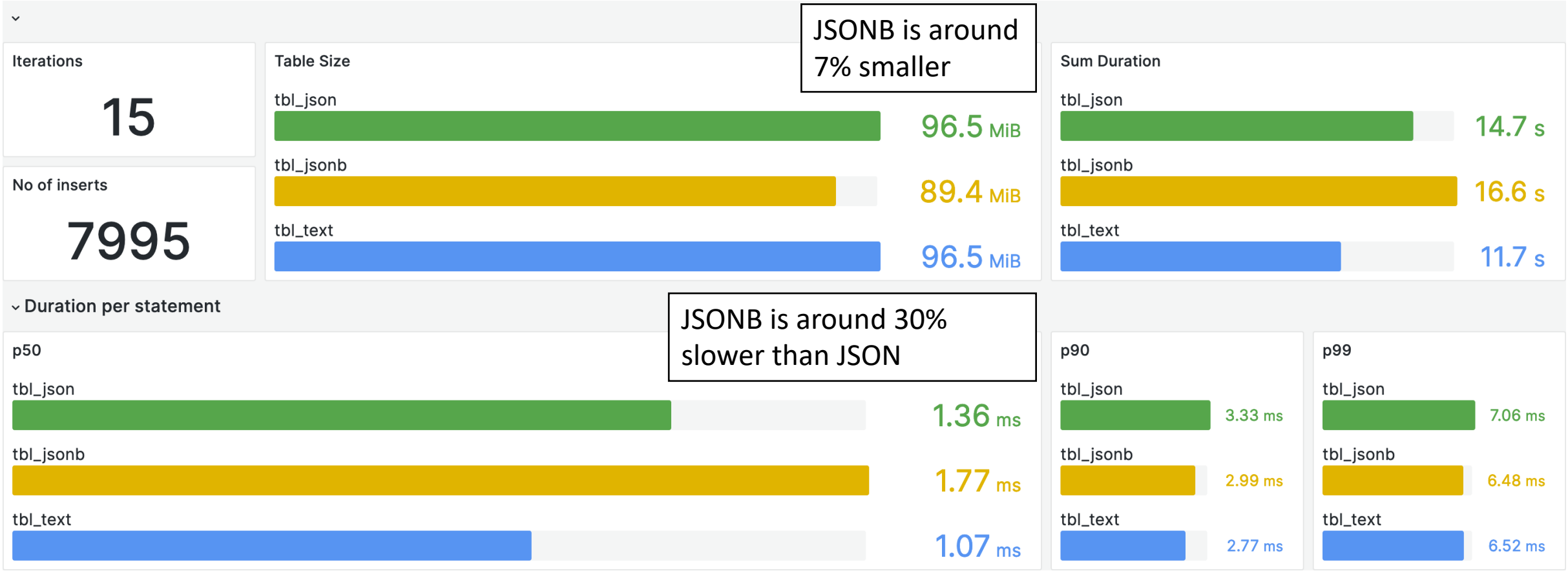
Introduced in PostgreSQL 9.4 (2014-12-18)

“...jsonb data is stored in a decomposed binary format that makes it slightly slower to input due to added conversion overhead, but significantly faster to process, since no reparsing is needed. jsonb also supports indexing, which can be a significant advantage.”

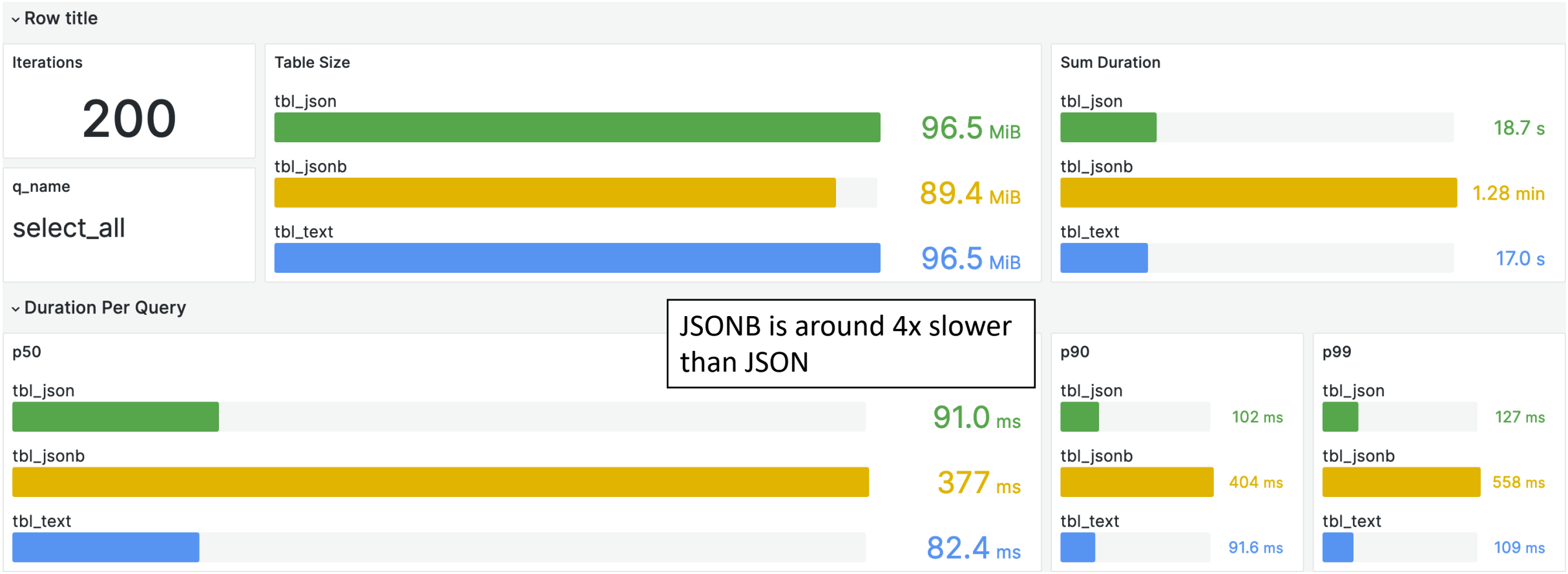
Into the JSONB field

```
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=#
```

TEXT vs JSON vs JSONB - Insert



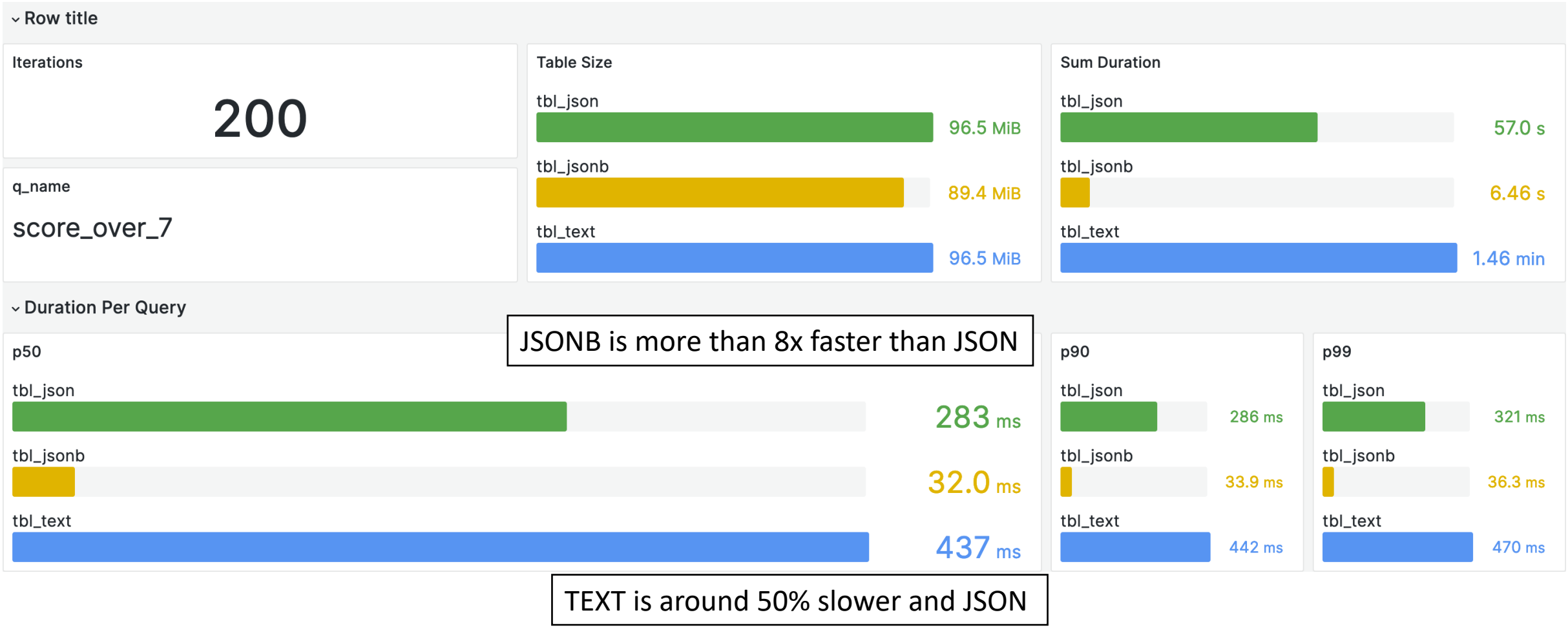
TEXT vs JSON vs JSONB - Select all



Select all where score is over 7.0

```
-- 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
```

Select all where score is over 7.0



Data Type

Use JSONB

- On most applications we need to manipulate the data, and JSONB was made for that

Unless:

- You need to keep details of the document that are not JSON semantics, then use JSON

Indexing

- Index on expression
- JSONB and GIN index
- btree and hash *

* btree and hash are useful only if it's important to check equality of complete JSON documents

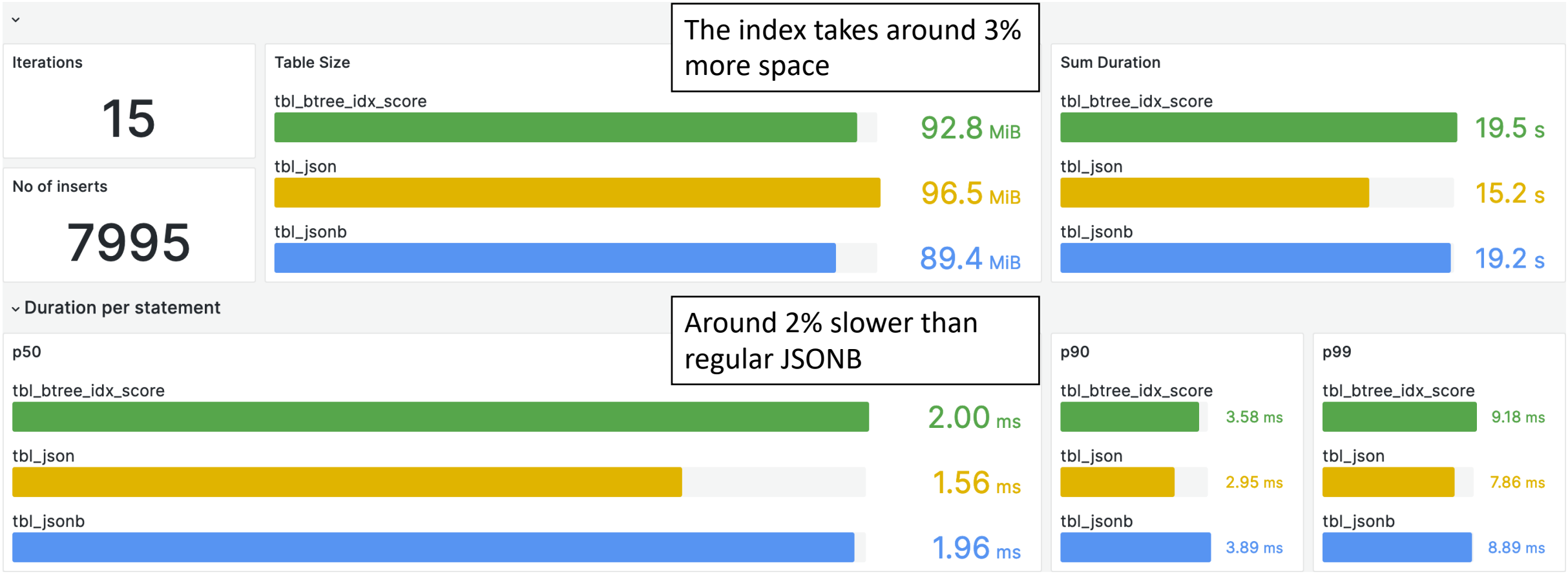
Index on expression

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

test=# CREATE INDEX ON tbl_idx_score (CAST(data->>'score' AS FLOAT));
CREATE INDEX
```

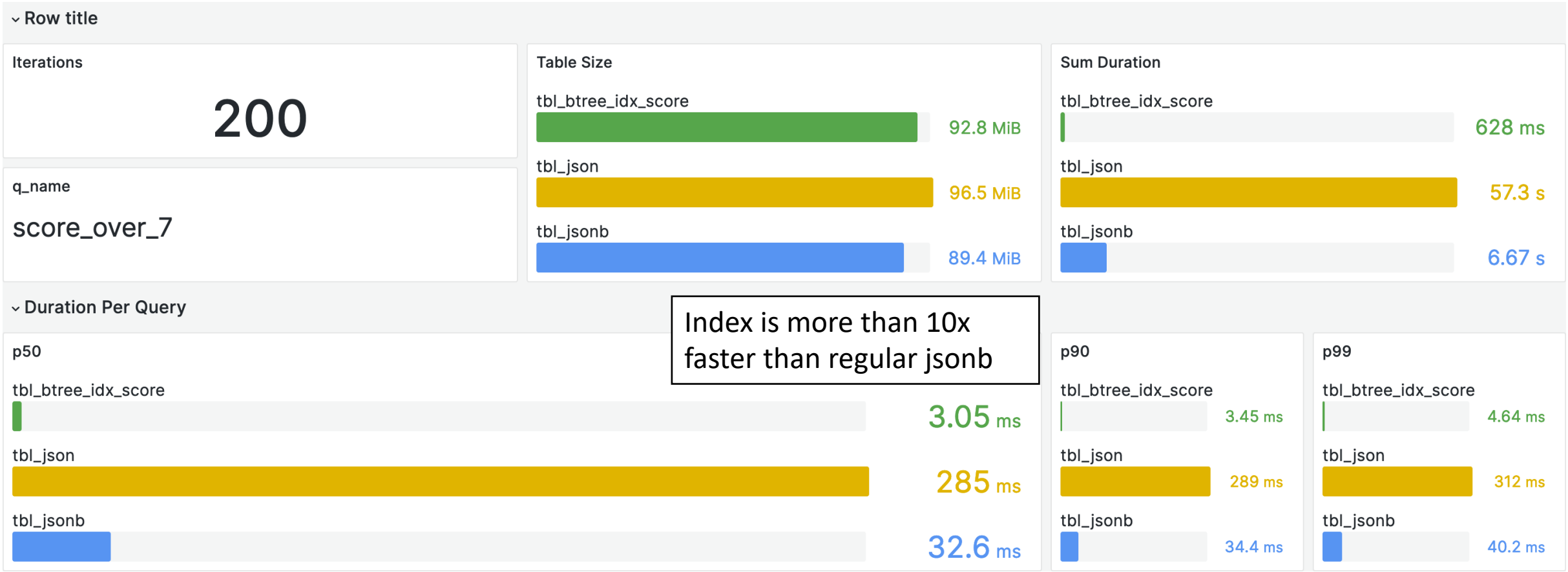
Query:

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

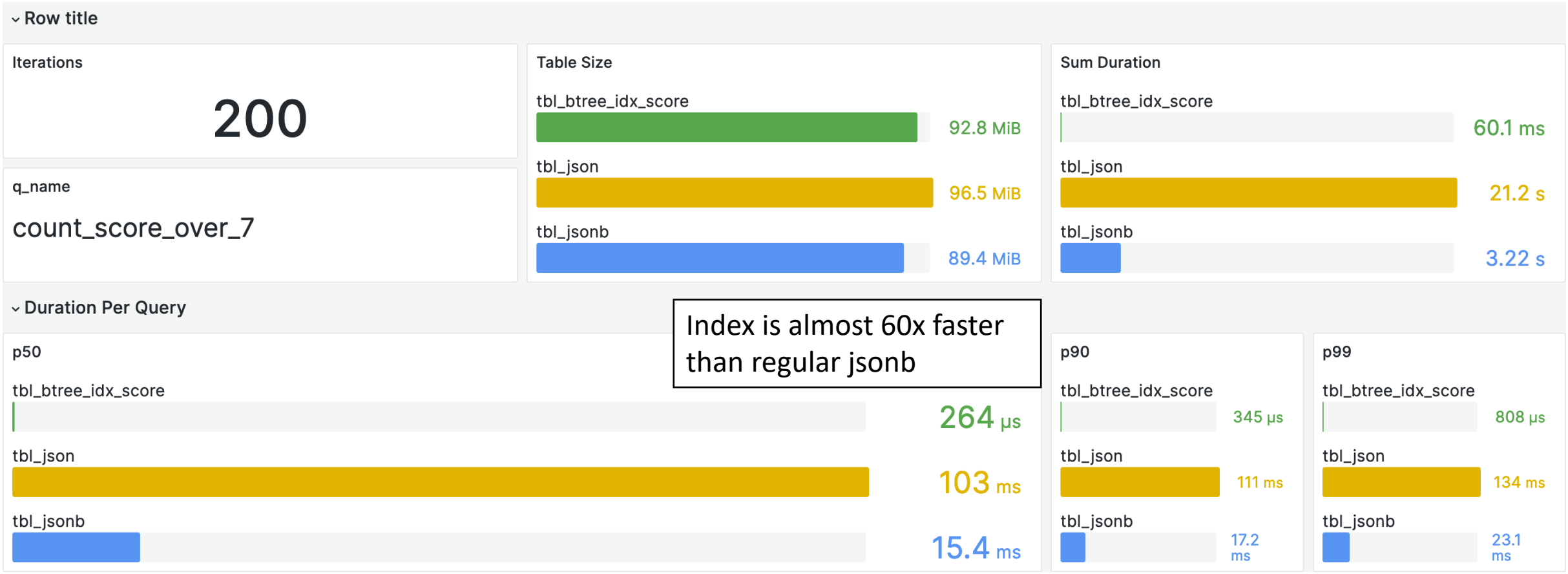
Index on expression - Insert



JSON vs JSONB vs Score Index – Select all where score is over 7.0



JSON vs JSONB vs Score Index – Count where score is over 7.0



JSON vs JSONB vs Score Index – Count where score is over 7.0

```
test=# EXPLAIN (ANALYZE, BUFFERS, COSTS OFF, TIMING OFF, SUMMARY OFF)
SELECT * FROM tbl_jsonb WHERE CAST(data->>'score' AS FLOAT) > 7.0;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on tbl_jsonb (actual rows=354 loops=1)
   Filter: (((data ->> 'score'::text))::double precision > '7'::double precision)
   Rows Removed by Filter: 132896
   Buffers: shared hit=2553 read=9523
   I/O Timings: shared/local read=16.895
(5 rows)
```

JSON vs JSONB vs Score Index – Count where score is over 7.0

```
test=# EXPLAIN (ANALYZE, BUFFERS, COSTS OFF, TIMING OFF, SUMMARY OFF)
SELECT * FROM tbl_btree_idx_score WHERE CAST(data->>'score' AS FLOAT) > 7.0;
                                        QUERY PLAN
-------------------------------------------------------------------------------------
 Index Scan using tbl_btree_idx_score_float8_idx on tbl_btree_idx_score (actual rows=354
loops=1)
   Index Cond: (((data ->> 'score'::text))::double precision > '7'::double precision)
   Buffers: shared hit=90
(3 rows)
```

JSONB and GIN index

```
test=# CREATE TABLE tbl_gin_idx (data jsonb);
CREATE TABLE
test=# CREATE INDEX ON tbl_gin_idx USING GIN (data);
CREATE INDEX

test=# CREATE TABLE tbl_gin_idx_path (data jsonb);
CREATE TABLE
test=# CREATE INDEX ON tbl_gin_idx_path USING GIN (data jsonb_path_ops);
CREATE INDEX
```

Operator Class jsonb_ops

Operator	Description	Example
?	Does the text string exist as a top-level key or array element within the JSON value?	`'{"a":1, "b":2}'::jsonb ? 'b' → t`
?|	Do any of the strings in the text array exist as top-level keys or array elements?	`'{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'd'] → t`
?&	Do all of the strings in the text array exist as top-level keys or array elements?	`'["a", "b", "c"]'::jsonb ?& array['a', 'b'] → t`
@>	Does the first JSON value contain the second?	`'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb → t`
@?	Does JSON path return any item for the specified JSON value?	`'{"a":[1,2,3,4,5]}'::jsonb @? '$.a[*] ? (@ > 2)' → t`
@@	Returns the result of a JSON path predicate check for the specified JSON value. Only the first item of the result is taken into account. If the result is not Boolean, then NULL is returned.	`'{"a":[1,2,3,4,5]}'::jsonb @@ '$.a[*] > 2' → t`

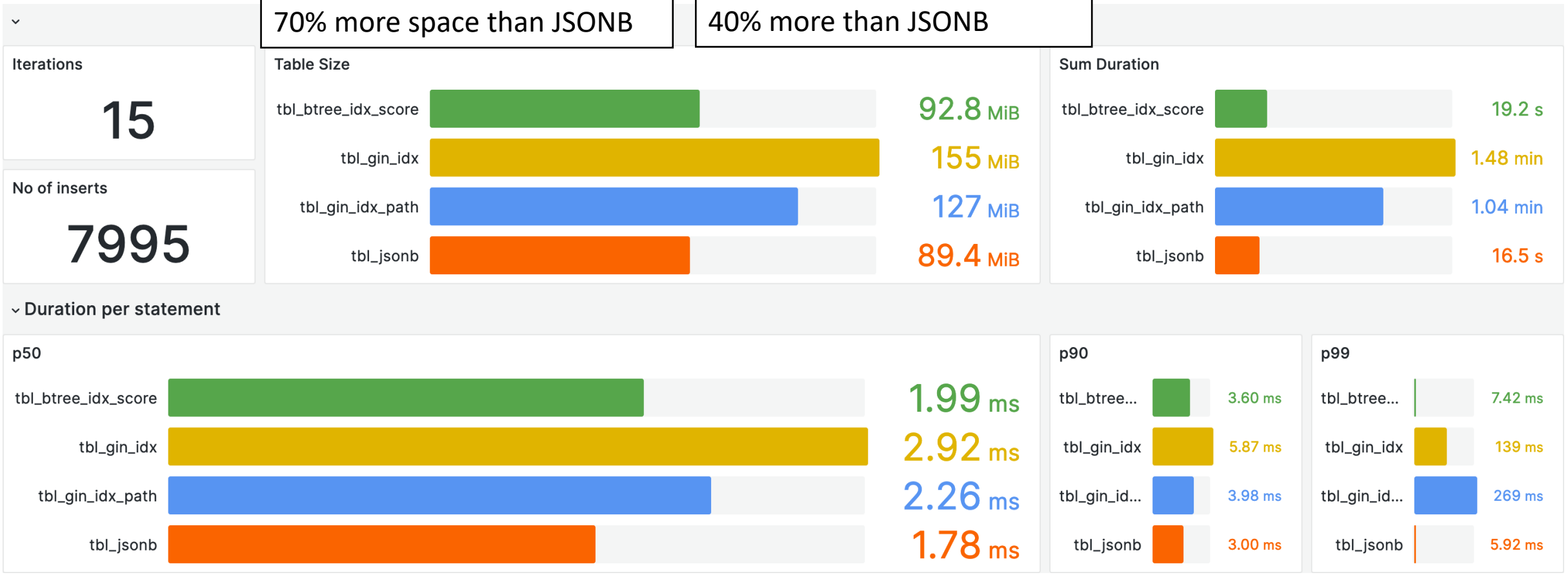
Operator Class jsonb_path_ops

Operator	Description	Example
~~?~~	~~Does the text string exist as a top-level key or array element within the JSON value?~~	~~'{"a":1, "b":2}'::jsonb ? 'b' → t~~
~~?|~~	~~Do any of the strings in the text array exist as top-level keys or array elements?~~	~~'{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'd'] → t~~
~~?&~~	~~Do all of the strings in the text array exist as top-level keys or array elements?~~	~~'["a", "b", "c"]'::jsonb ?& array['a', 'b'] → t~~
@>	Does the first JSON value contain the second?	'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb → t
@?	Does JSON path return any item for the specified JSON value?	'{"a":[1,2,3,4,5]}'::jsonb @? '$.a[*] ? (@ > 2)' → t
@@	Returns the result of a JSON path predicate check for the specified JSON value. Only the first item of the result is taken into account. If the result is not Boolean, then NULL is returned.	'{"a":[1,2,3,4,5]}'::jsonb @@ '$.a[*] > 2' → t

JSONB and GIN index - Insert

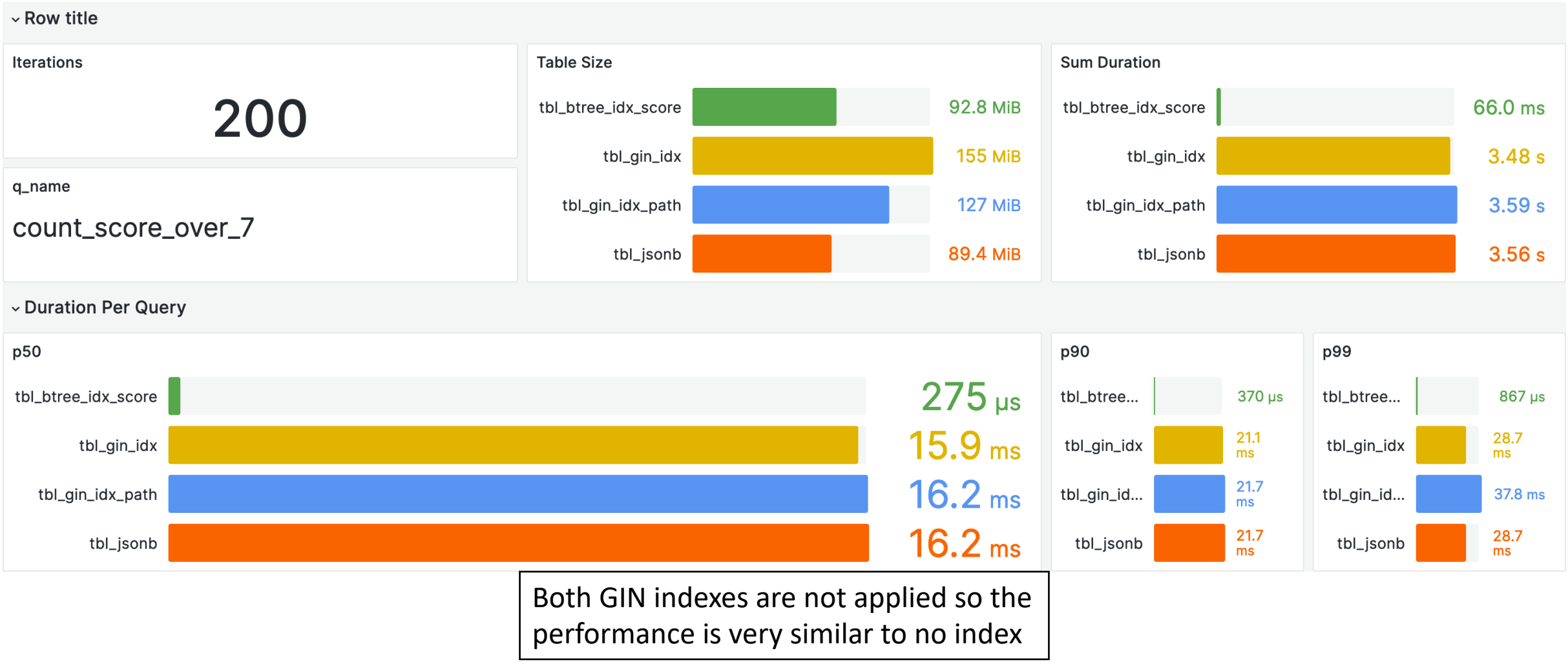
Default GIN Index takes over 70% more space than JSONB

jsonb_path_ops takes “only” 40% more than JSONB



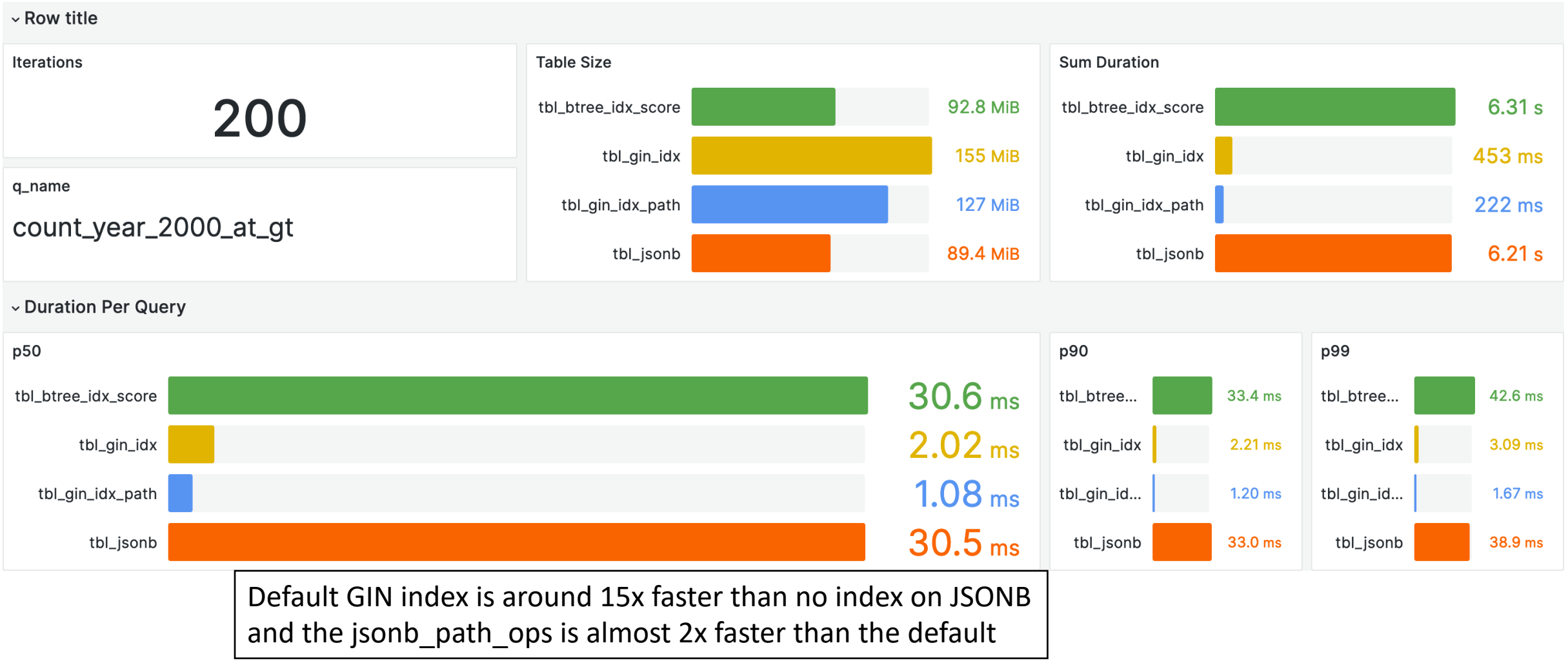
Default GIN Index takes over 60% more than JSONB, jsonb_path_ops takes a bit over 25%

JSONB and GIN index - Count where score is over 7.0



Both GIN indexes are not applied so the performance is very similar to no index

JSONB and GIN index - Count year is 2000 using @>



Default GIN index is around 15x faster than no index on JSONB and the jsonb_path_ops is almost 2x faster than the default

JSONB and GIN index – without index

```
test=# EXPLAIN (ANALYZE, BUFFERS, COSTS OFF, TIMING OFF, SUMMARY OFF)
SELECT COUNT(*) FROM tbl_jsonb WHERE data @> '{"year": 2000}';
                        QUERY PLAN
-----------------------------------------------------------
 Aggregate (actual rows=1 loops=1)
   Buffers: shared hit=4177 read=7899
   I/O Timings: shared/local read=19.720
   ->  Seq Scan on tbl_jsonb (actual rows=2348 loops=1)
         Filter: (data @> '{"year": 2000}'::jsonb)
         Rows Removed by Filter: 130902
         Buffers: shared hit=4177 read=7899
         I/O Timings: shared/local read=19.720
(8 rows)
```

JSONB and GIN index – GIN jsonb_ops

```
test=# EXPLAIN (ANALYZE, BUFFERS, COSTS OFF, TIMING OFF, SUMMARY OFF)
SELECT COUNT(*) FROM tbl_gin_idx WHERE data @> '{"year": 2000}';
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate (actual rows=1 loops=1)
   Buffers: shared hit=596
   ->  Bitmap Heap Scan on tbl_gin_idx (actual rows=2348 loops=1)
         Recheck Cond: (data @> '{"year": 2000}'::jsonb)
         Heap Blocks: exact=539
         Buffers: shared hit=596
         ->  Bitmap Index Scan on tbl_gin_idx_data_idx (actual rows=2348 loops=1)
               Index Cond: (data @> '{"year": 2000}'::jsonb)
               Buffers: shared hit=36
 Planning:
   Buffers: shared hit=1
(11 rows)
```

JSONB and GIN index – GIN jsonb_path_ops

```
test=# EXPLAIN (ANALYZE, BUFFERS, COSTS OFF, TIMING OFF, SUMMARY OFF)
SELECT COUNT(*) FROM tbl_gin_idx_path WHERE data @> '{"year": 2000}';
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Aggregate (actual rows=1 loops=1)
   Buffers: shared hit=565
   ->  Bitmap Heap Scan on tbl_gin_idx_path (actual rows=2348 loops=1)
         Recheck Cond: (data @> '{"year": 2000}'::jsonb)
         Heap Blocks: exact=539
         Buffers: shared hit=565
         ->  Bitmap Index Scan on tbl_gin_idx_path_data_idx (actual rows=2348 loops=1)
               Index Cond: (data @> '{"year": 2000}'::jsonb)
               Buffers: shared hit=5
 Planning:
   Buffers: shared hit=1
(11 rows)
```

Indexing

Take advantage of the different option of indexing and fit them to your use case and remember that they have cost.

Index on expression

- (+) Good when the schema is well defined, and the most frequent searches are well known and require operators not supported by GIN.
- (-) Need to define a different index for each expression used in searches

Indexing

Take advantage of the different option of indexing and fit them to your use case and remember that they have cost.

JSONB GIN index

- (+) A single index will cover all attributes.
- (-) Operations like > and < common for numeric values are not supported

Indexing

Take advantage of the different option of indexing and fit them to your use case and remember that they have cost.

jsonb_ops vs jsonb_path_ops

- The non-default option “jsonb_path_ops“ is actually a very good option when you have a more consistent schema and key exists operators are not required

Data Size Limits

TEXT and JSON fields:

“…the longest possible character string that can be stored is about 1 GB”

JSONB fields:

The limit for a jsonb field is 255MB.

JSONB max size allowed

```
// src/include/utils/jsonb.h:138
#define JENTRY_OFFLENMASK       0x0FFFFFFF

// src/backend/utils/adt/jsonb.c:281
static bool
checkStringLen(size_t len, Node *escontext)
{
    if (len > JENTRY_OFFLENMASK)
        ereturn(escontext, false,
                (errcode(ERRCODE_PROGRAM_LIMIT_EXCEEDED),
                 errmsg("string too long to represent as jsonb string"),
                 errdetail("Due to an implementation restriction, jsonb strings cannot exceed %d bytes.",
                           JENTRY_OFFLENMASK)));

    return true;
}
```

Q&A